SQL Anywhere® Server
SQL Reference

February 2009

Version 11.0.1

Copyright and trademarks

Contents

About this book

Subject

This book provides reference information for system procedures, and the catalog (system tables and views). It also provides an explanation of the SQL Anywhere implementation of the SQL language (search conditions, syntax, data types, and functions).

Audience

This book is for all users of SQL Anywhere. It includes material of particular interest to users of MobiLink, UltraLite and SQL Remote. It is to be used in conjunction with other books in the documentation set.

About the SQL Anywhere documentation

The complete SQL Anywhere documentation is available in four formats that contain identical information.

- **HTML Help** The online Help contains the complete SQL Anywhere documentation, including the books and the context-sensitive help for SQL Anywhere tools.

  If you are using a Microsoft Windows operating system, the online Help is provided in HTML Help (CHM) format. To access the documentation, choose **Start** » **Programs** » **SQL Anywhere 11** » **Documentation** » **Online Books**.

  The administration tools use the same online documentation for their Help features.

- **Eclipse** On Unix platforms, the complete online Help is provided in Eclipse format. To access the documentation, run *sadoc* from the *bin32* or *bin64* directory of your SQL Anywhere 11 installation.

- **DocCommentXchange** DocCommentXchange is a community for accessing and discussing SQL Anywhere documentation.

  Use DocCommentXchange to:

  - View documentation
  - Check for clarifications users have made to sections of documentation
  - Provide suggestions and corrections to improve documentation for all users in future releases

  Visit http://dcx.sybase.com.

- **PDF** The complete set of SQL Anywhere books is provided as a set of Portable Document Format (PDF) files. You must have a PDF reader to view information. To download Adobe Reader, visit http://get.adobe.com/reader/.

  To access the PDF documentation on Microsoft Windows operating systems, choose **Start** » **Programs** » **SQL Anywhere 11** » **Documentation** » **Online Books - PDF Format**.

  To access the PDF documentation on Unix operating systems, use a web browser to open *install-dir/documentation/en/pdf/index.html*.

About the books in the documentation set

The SQL Anywhere documentation consists of the following books:

- **SQL Anywhere 11 - Introduction** This book introduces SQL Anywhere 11, a comprehensive package that provides data management and data exchange, enabling the rapid development of database-powered applications for server, desktop, mobile, and remote office environments.
- **SQL Anywhere 11 - Changes and Upgrading** This book describes new features in SQL Anywhere 11 and in previous versions of the software.
- **SQL Anywhere Server - Database Administration** This book describes how to run, manage, and configure SQL Anywhere databases. It describes database connections, the database server, database

files, backup procedures, security, high availability, replication with the Replication Server, and administration utilities and options.

- **SQL Anywhere Server - Programming** This book describes how to build and deploy database applications using the C, C++, Java, PHP, Perl, Python, and .NET programming languages such as Visual Basic and Visual C#. A variety of programming interfaces such as ADO.NET and ODBC are described.
- **SQL Anywhere Server - SQL Reference** This book provides reference information for system procedures, and the catalog (system tables and views). It also provides an explanation of the SQL Anywhere implementation of the SQL language (search conditions, syntax, data types, and functions).
- **SQL Anywhere Server - SQL Usage** This book describes how to design and create databases; how to import, export, and modify data; how to retrieve data; and how to build stored procedures and triggers.
- **MobiLink - Getting Started** This book introduces MobiLink, a session-based relational-database synchronization system. MobiLink technology allows two-way replication and is well suited to mobile computing environments.
- **MobiLink - Client Administration** This book describes how to set up, configure, and synchronize MobiLink clients. MobiLink clients can be SQL Anywhere or UltraLite databases. This book also describes the Dbmlsync API, which allows you to integrate synchronization seamlessly into your C++ or .NET client applications.
- **MobiLink - Server Administration** This book describes how to set up and administer MobiLink applications.
- **MobiLink - Server-Initiated Synchronization** This book describes MobiLink server-initiated synchronization, a feature that allows the MobiLink server to initiate synchronization or perform actions on remote devices.
- **QAnywhere** This book describes QAnywhere, which is a messaging platform for mobile, wireless, desktop, and laptop clients.
- **SQL Remote** This book describes the SQL Remote data replication system for mobile computing, which enables sharing of data between a SQL Anywhere consolidated database and many SQL Anywhere remote databases using an indirect link such as email or file transfer.
- **UltraLite - Database Management and Reference** This book introduces the UltraLite database system for small devices.
- **UltraLite - C and C++ Programming** This book describes UltraLite C and C++ programming interfaces. With UltraLite, you can develop and deploy database applications to handheld, mobile, or embedded devices.
- **UltraLite - M-Business Anywhere Programming** This book describes UltraLite for M-Business Anywhere. With UltraLite for M-Business Anywhere you can develop and deploy web-based database applications to handheld, mobile, or embedded devices, running Palm OS, Windows Mobile, or Windows.
- **UltraLite - .NET Programming** This book describes UltraLite.NET. With UltraLite.NET you can develop and deploy database applications to computers, or handheld, mobile, or embedded devices.
- **UltraLiteJ** This book describes UltraLiteJ. With UltraLiteJ, you can develop and deploy database applications in environments that support Java. UltraLiteJ supports BlackBerry smartphones and Java SE environments. UltraLiteJ is based on the iAnywhere UltraLite database product.

- **Error Messages** This book provides a complete listing of SQL Anywhere error messages together with diagnostic information.

Documentation conventions

This section lists the conventions used in this documentation.

Operating systems

SQL Anywhere runs on a variety of platforms. In most cases, the software behaves the same on all platforms, but there are variations or limitations. These are commonly based on the underlying operating system (Windows, Unix), and seldom on the particular variant (AIX, Windows Mobile) or version.

To simplify references to operating systems, the documentation groups the supported operating systems as follows:

- **Windows** The Microsoft Windows family includes Windows Vista and Windows XP, used primarily on server, desktop, and laptop computers, and Windows Mobile used on mobile devices.

  Unless otherwise specified, when the documentation refers to Windows, it refers to all Windows-based platforms, including Windows Mobile.
- **Unix** Unless otherwise specified, when the documentation refers to Unix, it refers to all Unix-based platforms, including Linux and Mac OS X.

Directory and file names

In most cases, references to directory and file names are similar on all supported platforms, with simple transformations between the various forms. In these cases, Windows conventions are used. Where the details are more complex, the documentation shows all relevant forms.

These are the conventions used to simplify the documentation of directory and file names:

- **Uppercase and lowercase directory names** On Windows and Unix, directory and file names may contain uppercase and lowercase letters. When directories and files are created, the file system preserves letter case.

  On Windows, references to directories and files are *not* case sensitive. Mixed case directory and file names are common, but it is common to refer to them using all lowercase letters. The SQL Anywhere installation contains directories such as *Bin32* and *Documentation*.

  On Unix, references to directories and files *are* case sensitive. Mixed case directory and file names are not common. Most use all lowercase letters. The SQL Anywhere installation contains directories such as *bin32* and *documentation*.

  The documentation uses the Windows forms of directory names. In most cases, you can convert a mixed case directory name to lowercase for the equivalent directory name on Unix.
- **Slashes separating directory and file names** The documentation uses backslashes as the directory separator. For example, the PDF form of the documentation is found in *install-dir\Documentation\en \PDF* (Windows form).

On Unix, replace the backslash with the forward slash. The PDF documentation is found in *install-dir/documentation/en/pdf*.

- **Executable files** The documentation shows executable file names using Windows conventions, with a suffix such as *.exe* or *.bat*. On Unix, executable file names have no suffix.

  For example, on Windows, the network database server is *dbsrv11.exe*. On Unix, it is *dbsrv11*.

- ***install-dir*** During the installation process, you choose where to install SQL Anywhere. The environment variable SQLANY11 is created and refers to this location. The documentation refers to this location as *install-dir*.

  For example, the documentation may refer to the file *install-dir*\readme.txt. On Windows, this is equivalent to *%SQLANY11%\readme.txt*. On Unix, this is equivalent to *$SQLANY11/readme.txt* or *${SQLANY11}/readme.txt*.

  For more information about the default location of *install-dir*, see "SQLANY11 environment variable" [*SQL Anywhere Server - Database Administration*].

- ***samples-dir*** During the installation process, you choose where to install the samples included with SQL Anywhere. The environment variable SQLANYSAMP11 is created and refers to this location. The documentation refers to this location as *samples-dir*.

  To open a Windows Explorer window in *samples-dir*, from the **Start** menu, choose **Programs** » **SQL Anywhere 11** » **Sample Applications And Projects**.

  For more information about the default location of *samples-dir*, see "SQLANYSAMP11 environment variable" [*SQL Anywhere Server - Database Administration*].

Command prompts and command shell syntax

Most operating systems provide one or more methods of entering commands and parameters using a command shell or command prompt. Windows command prompts include Command Prompt (DOS prompt) and 4NT. Unix command shells include Korn shell and bash. Each shell has features that extend its capabilities beyond simple commands. These features are driven by special characters. The special characters and features vary from one shell to another. Incorrect use of these special characters often results in syntax errors or unexpected behavior.

The documentation provides command line examples in a generic form. If these examples contain characters that the shell considers special, the command may require modification for the specific shell. The modifications are beyond the scope of this documentation, but generally, use quotes around the parameters containing those characters or use an escape character before the special characters.

These are some examples of command line syntax that may vary between platforms:

- **Parentheses and curly braces** Some command line options require a parameter that accepts detailed value specifications in a list. The list is usually enclosed with parentheses or curly braces. The documentation uses parentheses. For example:

  ```
  -x tcpip(host=127.0.0.1)
  ```

  Where parentheses cause syntax problems, substitute curly braces:

  ```
  -x tcpip{host=127.0.0.1}
  ```

If both forms result in syntax problems, the entire parameter should be enclosed in quotes as required by the shell:

```
-x "tcpip(host=127.0.0.1)"
```

- **Quotes** If you must specify quotes in a parameter value, the quotes may conflict with the traditional use of quotes to enclose the parameter. For example, to specify an encryption key whose value contains double-quotes, you might have to enclose the key in quotes and then escape the embedded quote:

  ```
  -ek "my \"secret\" key"
  ```

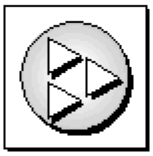
  In many shells, the value of the key would be `my "secret" key`.

- **Environment variables** The documentation refers to setting environment variables. In Windows shells, environment variables are specified using the syntax *%ENVVAR%*. In Unix shells, environment variables are specified using the syntax *$ENVVAR* or *${ENVVAR}*.

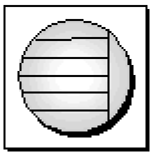
Graphic icons

The following icons are used in this documentation.

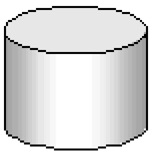
- A client application.



- A database server, such as Sybase SQL Anywhere.



- A database. In some high-level diagrams, the icon may be used to represent both the database and the database server that manages it.



- Replication or synchronization middleware. These assist in sharing data among databases. Examples are the MobiLink server and the SQL Remote Message Agent.

- A programming interface.

Interface

Contacting the documentation team

We would like to receive your opinions, suggestions, and feedback on this Help.

To submit your comments and suggestions, send an email to the SQL Anywhere documentation team at iasdoc@sybase.com. Although we do not reply to emails, your feedback helps us to improve our documentation, so your input is welcome.

DocCommentXchange

You can also leave comments directly on help topics using DocCommentXchange. DocCommentXchange (DCX) is a community for accessing and discussing SQL Anywhere documentation. Use DocCommentXchange to:

- View documentation
- Check for clarifications users have made to sections of documentation
- Provide suggestions and corrections to improve documentation for all users in future releases

Visit http://dcx.sybase.com.

Finding out more and requesting technical support

Additional information and resources are available at the Sybase iAnywhere Developer Community at http://www.sybase.com/developer/library/sql-anywhere-techcorner.

If you have questions or need help, you can post messages to the Sybase iAnywhere newsgroups listed below.

When you write to one of these newsgroups, always provide details about your problem, including the build number of your version of SQL Anywhere. You can find this information by running the following command: **dbeng11 -v**.

The newsgroups are located on the *forums.sybase.com* news server.

The newsgroups include the following:

- sybase.public.sqlanywhere.general
- sybase.public.sqlanywhere.linux
- sybase.public.sqlanywhere.mobilink
- sybase.public.sqlanywhere.product_futures_discussion
- sybase.public.sqlanywhere.replication
- sybase.public.sqlanywhere.ultralite
- ianywhere.public.sqlanywhere.qanywhere

For web development issues, see http://groups.google.com/group/sql-anywhere-web-development.

Newsgroup disclaimer

iAnywhere Solutions has no obligation to provide solutions, information, or ideas on its newsgroups, nor is iAnywhere Solutions obliged to provide anything other than a systems operator to monitor the service and ensure its operation and availability.

iAnywhere Technical Advisors, and other staff, assist on the newsgroup service when they have time. They offer their help on a volunteer basis and may not be available regularly to provide solutions and information. Their ability to help is based on their workload.

Using SQL

This section describes the SQL Anywhere SQL language, including data types, functions, and statements.

SQL language elements

Contents

Keywords

Each SQL statement contains one or more keywords. SQL is case insensitive to keywords, but throughout these manuals, keywords are indicated in uppercase.

For example, in the following statement, SELECT and FROM are keywords:

```
SELECT *
   FROM Employees;
```

The following statements are equivalent to the one above:

```
Select *
   From Employees;
select * from Employees;
sELECT * FRoM Employees;
```

Some keywords cannot be used as identifiers without surrounding them in double quotes. These are called reserved words. Other keywords, such as DBA, do not require double quotes, and are not reserved words.

Reserved words

Some keywords in SQL are also **reserved words**. To use a reserved word in a SQL statement as an identifier, you must enclose it in double quotes. Many, but not all, the keywords that appear in SQL statements are reserved words. For example, you must use the following syntax to retrieve the contents of a table named SELECT.

```
SELECT *
   FROM "SELECT"
```

SQL keywords are not case sensitive and the following words may appear in uppercase, lowercase, or any combination of the two. All strings that differ only in capitalization from one of the following words are reserved words.

If you are using embedded SQL, you can use the sql_needs_quotes database library function to determine whether a string requires quotation marks. A string requires quotes if it is a reserved word or if it contains a character not ordinarily allowed in an identifier.

The reserved SQL keywords in SQL Anywhere are as follows:

add	**all**	**alter**	**and**
any	**as**	**asc**	**attach**
backup	**begin**	**between**	**bigint**
binary	**bit**	**bottom**	**break**
by	**call**	**capability**	**cascade**
case	**cast**	**char**	**char_convert**

character	**check**	**checkpoint**	**close**
comment	**commit**	**compressed**	**conflict**
connect	**constraint**	**contains**	**continue**
convert	**create**	**cross**	**cube**
current	**current_timestamp**	**current_user**	**cursor**
date	**dbspace**	**deallocate**	**dec**
decimal	**declare**	**default**	**delete**
deleting	**desc**	**detach**	**distinct**
do	**double**	**drop**	**dynamic**
else	**elseif**	**encrypted**	**end**
endif	**escape**	**except**	**exception**
exec	**execute**	**existing**	**exists**
externlogin	**fetch**	**first**	**float**
for	**force**	**foreign**	**forward**
from	**full**	**goto**	**grant**
group	**having**	**holdlock**	**identified**
if	**in**	**index**	**index_lparen**
inner	**inout**	**insensitive**	**insert**
inserting	**install**	**instead**	**int**
integer	**integrated**	**intersect**	**into**
is	**isolation**	**join**	**kerberos**
key	**lateral**	**left**	**like**
lock	**login**	**long**	**match**
membership	**merge**	**message**	**mode**
modify	**natural**	**nchar**	**new**

no	**noholdlock**	**not**	**notify**
null	**numeric**	**nvarchar**	**of**
off	**on**	**open**	**openstring**
option	**options**	**or**	**order**
others	**out**	**outer**	**over**
passthrough	**precision**	**prepare**	**primary**
print	**privileges**	**proc**	**procedure**
publication	**raiserror**	**readtext**	**real**
reference	**references**	**refresh**	**release**
remote	**remove**	**rename**	**reorganize**
resource	**restore**	**restrict**	**return**
revoke	**right**	**rollback**	**rollup**
save	**savepoint**	**scroll**	**select**
sensitive	**session**	**set**	**setuser**
share	**smallint**	**some**	**sqlcode**
sqlstate	**start**	**stop**	**subtrans**
subtransaction	**synchronize**	**syntax_error**	**table**
temporary	**then**	**time**	**timestamp**
tinyint	**to**	**top**	**tran**
trigger	**truncate**	**tsequal**	**unbounded**
union	**unique**	**uniqueidentifier**	**unknown**
unsigned	**update**	**updating**	**user**
using	**validate**	**values**	**varbinary**
varbit	**varchar**	**variable**	**varying**
view	**wait**	**waitfor**	**when**

where	**while**	**window**	**with**
with_cube	**with_lparen**	**with_rollup**	**within**
work	**writetext**	**xml**	

See also

- "sql_needs_quotes function" [*SQL Anywhere Server - Programming*]

Identifiers

Identifiers are names of objects in the database, such as user IDs, tables, and columns.

Remarks

Identifiers have a maximum length of 128 bytes. They must be enclosed in double quotes or square brackets if any of the following conditions are true:

- The identifier contains spaces.
- The first character of the identifier is not an alphabetic character (as defined below).
- The identifier contains a reserved word.
- The identifier contains characters other than alphabetic characters and digits.

Alphabetic characters includes the alphabet, the underscore character (_), at sign (@), number sign (#), and dollar sign ($). The database collation sequence dictates which characters are considered alphabetic or digit characters.

The following characters are not permitted in identifiers:

- Double quotes
- Control characters (any character less than 0x20)
- Backslashes

If the quoted_identifier database option is set to Off, double quotes are used to delimit SQL strings and cannot be used for identifiers. However, you can always use square brackets to delimit identifiers, regardless of the setting of quoted_identifier. The default setting for the quoted_identifier option is to Off for Open Client and jConnect connections; otherwise the default is On.

See also

- For a complete list of the reserved words, see “Reserved words” on page 4.
- For information about the quoted_identifier option, see “quoted_identifier option [compatibility]” [*SQL Anywhere Server - Database Administration*].

Examples

The following are all valid identifiers.

- Surname
- "Surname"
- [Surname]
- SomeBigName
- "Client Number"

Strings

A string is a sequence of characters up to 2 GB in size. A string can occur in SQL:

- as a **string literal**. A string literal is a sequence of characters enclosed in single quotes (apostrophes). A string literal represents a particular, constant value, and it may contain escape sequences for special characters that cannot be easily typed as characters. See "String literals" on page 11.
- as the value of a column or variable with a CHAR or NCHAR data type.
- as the result of evaluating an expression.

The length of a string can be measured in two ways:

- **Byte length** The byte length is the number of bytes in the string.
- **Character length** The character length is the number of characters in the string, and is based on the character set being used.

For single-byte character sets, such as cp1252, the byte-length and character-length are the same. For multibyte character sets, a string's byte-length is greater than or equal to its character-length.

Constants

This section describes binary literals and string literals.

Binary literals

A binary literal is a sequence of hexadecimal characters consisting of digits 0-9 and uppercase and lowercase letters A-F. When you enter binary data as literals, you must precede the data by 0x (a zero, followed by an x), and there should be an even number of digits to the right of this prefix. For example, the hexadecimal equivalent of 39 is 0027, and is expressed as 0x0027.

Hexadecimal constants in the form of 0x12345678 are treated as binary strings. An unlimited number of digits can be added after the 0x.

A binary literal is sometimes referred to as a binary constant. In SQL Anywhere, the preferred term is binary literal.

Converting to and from hexadecimal values

You can use the CAST, CONVERT, HEXTOINT, and INTTOHEX functions to convert a binary string to an integer. The CAST and CONVERT functions convert hexadecimal constants to TINYINT, signed and unsigned 32-bit integer, signed and unsigned 64-bit integer, NUMERIC, and so on. The HEXTOINT function only converts a hexadecimal constant to a signed 32-bit-integer.

The value returned by the CAST function cannot exceed 8 digits. Values exceeding 8 digits return an error. Zeroes are added to the left of values less than 8 digits. For example, the following argument returns the value -2,147,483,647:

```
SELECT CAST ( 0x0080000001 AS INT );
```

The following argument returns an error because the 10-digit value cannot be represented as a signed 32-bit integer:

```
SELECT CAST ( 0xff80000001 AS INT );
```

The value returned by the HEXTOINT function can exceed 8 digits if the value can be represented as a signed 32-bit integer. The HEXTOINT function accepts string literals or variables consisting only of digits and the uppercase or lowercase letters A-F, with or without a 0x prefix. The hexadecimal value represents a negative integer when the 8th digit from the right is one of the digits 8-9, the uppercase or lowercase letters A-F, or the previous leading digits are all uppercase or lowercase letter F.

The following arguments return the value -2,147,483,647:

```
SELECT HEXTOINT( '0xFF80000001' );
```

```
SELECT HEXTOINT( '0x80000001' );
```

```
SELECT HEXTOINT ( '0xFFFFFFFFFFFFFFFF80000001' );
```

The following argument returns an error because the argument represents a positive integer value that cannot be represented as a signed 32-bit integer:

```
SELECT HEXTOINT( '0x0080000001' );
```

See also

- “CAST function [Data type conversion]” on page 148
- “CONVERT function [Data type conversion]” on page 160
- “HEXTOINT function [Data type conversion]” on page 215
- “INTTOHEX function [Data type conversion]” on page 229

String literals

A string literal is a sequence of characters enclosed in single quotes. For example, `'Hello world'` is a string literal of type CHAR. Its byte length is 11, and its character length is also 11.

A string literal is sometimes referred to as a string constant, literal string, or just as a string. In SQL Anywhere, the preferred term is string literal.

You can specify an NCHAR string literal by prefixing the quoted value with N. For example, `N'Hello world'` is a string literal of type NCHAR. Its byte length is 11, and its character length is 11. The bytes within an NCHAR string literal are interpreted using the database's CHAR character set, and then converted to NCHAR. The syntax `N'`*`string`*`'` is a shortened form for `CAST( '`*`string`*`' AS NCHAR )`.

Escape sequences

Sometimes you need to put characters into string literals that cannot be typed or entered normally. Examples include control characters (such as a new line character), single quotes (which would otherwise mark the end of the string literal), and hexadecimal byte values. For this purpose, you use an escape sequence.

The following examples show how to use escape sequences in string literals.

- A single quote is used to mark the beginning and end of a string literal, so a single quote in a string must be escaped using an additional single quote, as follows: `'John''s database'`
- Hexadecimal escape sequences can be used for any character or binary value. A hexadecimal escape sequence is a backslash followed by an x followed by two hexadecimal digits. The hexadecimal value is interpreted as a character in the CHAR character set for both CHAR and NCHAR string literals. The following example, in code page 1252, represents the digits 1, 2, and 3, followed by the euro currency symbol: `'123\x80'`
- To represent a new line character, use a backslash followed by n (\n), specify: `'First line:\nSecond line:'`
- A backslash is used to mark the beginning of an escape sequence, so a backslash character in a string must be escaped using an additional backslash, as follows: `'c:\\temp'`

You can use the same characters and escape sequences with NCHAR string literals as with CHAR string literals.

If you need to use Unicode characters that cannot be typed directly into the string literal, use the UNISTR function. See “UNISTR function [String]” on page 331.

Operators

This section describes arithmetic, string, and bitwise operators. For information about comparison operators, see “Search conditions” on page 35.

The normal precedence of operations applies. Expressions in parentheses are evaluated first, then multiplication and division before addition and subtraction. String concatenation happens after addition and subtraction.

For more information, see “Operator precedence” on page 15.

Comparison operators

The syntax for comparison is as follows:

expression comparison-operator expression

where *comparison-operator* is one of the following:

Operator	Description
=	Equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
!=	Not equal to
<>	Not equal to
!>	Not greater than
!<	Not less than

Case sensitivity By default, SQL Anywhere databases are created as case insensitive. Comparisons are carried out with the same attention to case as the database they are operating on. You can control the case sensitivity of SQL Anywhere databases with the -c option when you create the database.

For more information about case sensitivity for string comparisons, see “Initialization utility (dbinit)” [*SQL Anywhere Server - Database Administration*].

Note
All string comparisons are *case insensitive* unless the database was created as case sensitive.

Trailing blanks The behavior of SQL Anywhere when comparing strings is controlled by the -b option that is set when creating the database.

For more information about blank padding, see "Initialization utility (dbinit)" [*SQL Anywhere Server - Database Administration*].

Logical operators

Search conditions can be combined using the AND or OR operators. You can also negate them using the NOT operator, or test whether an expression would evaluate to true, false, or unknown, using the IS operator.

- **AND operator** The AND operator is placed between search conditions as follows:

  .. **WHERE** *condition1* **AND** *condition2*

  When using AND, the combined condition is TRUE if both conditions are TRUE, FALSE if either condition is FALSE, and UNKNOWN otherwise.

- **OR operator** The OR operator is placed between search conditions as follows:

  .. **WHERE** *condition1* **OR** *condition2*

  When using OR, the combined condition is TRUE if either condition is TRUE, FALSE if both conditions are FALSE, and UNKNOWN otherwise.

- **NOT operator** The NOT operator is placed before a condition to negate the condition, as follows:

  .. **WHERE NOT** *condition*

  The NOT condition is TRUE if *condition* is FALSE, FALSE if *condition* is TRUE, and UNKNOWN if *condition* is UNKNOWN.

- **IS operator** The IS operator is placed between an expression and the truth value you are testing for. The syntax for the IS operator is as follows:

  expression **IS** [**NOT**] *truth-value*

  The IS condition is TRUE if the *expression* evaluates to the supplied *truth-value*, which must be one of TRUE, FALSE, UNKNOWN, or NULL. Otherwise, the value is FALSE.

  For example, `5*3=15 IS TRUE` tests whether the expression `5*3=15` evaluates to TRUE.

See also: "Three-valued logic" on page 55.

Arithmetic operators

expression + expression Addition. If either expression is the NULL value, the result is NULL.

expression - expression Subtraction. If either expression is the NULL value, the result is NULL.

-expression Negation. If the expression is the NULL value, the result is NULL.

expression * expression Multiplication. If either expression is NULL, the result is NULL.

expression / expression Division. If either expression is NULL or if the second expression is 0, the result is NULL.

expression % expression Modulo finds the integer remainder after a division involving two whole numbers. For example, 21 % 11 = 10 because 21 divided by 11 equals 1 with a remainder of 10.

Standards and compatibility

- **Modulo** The % operator is supported in SQL Anywhere.

String operators

expression || expression String concatenation (two vertical bars). If either string is NULL, it is treated as the empty string for concatenation.

expression + expression Alternative string concatenation. When using the + concatenation operator, you must ensure the operands are explicitly set to character data types rather than relying on implicit data conversion.

For example, the following query returns the integer value 579:

```
SELECT 123 + 456;
```

whereas the following query returns the character string 123456:

```
SELECT '123' + '456';
```

You can use the CAST or CONVERT function to explicitly convert data types.

Standards and compatibility

- **SQL/2003** The || operator is the SQL/2003 string concatenation operator.

Bitwise operators

The following operators can be used on integer data types and bit array data types in SQL Anywhere.

Operator	Description
&	bitwise AND
|	bitwise OR
^	bitwise exclusive OR
~	bitwise NOT

The bitwise operators &, | and ~ are not interchangeable with the logical operators AND, OR, and NOT.

Example

For example, the following statement selects rows in which the correct bits are set.

```
SELECT *
FROM tableA
WHERE ( options & 0x0101 ) <> 0;
```

Join operators

The SQL/2003 join syntax that uses a table expression in the FROM clause is supported. See "FROM clause" on page 613.

Support for Transact-SQL outer join operators *= and =* is deprecated. To use Transact SQL outer joins, the tsql_outer_joins database option must be set to On. See "tsql_outer_joins option [compatibility]" [*SQL Anywhere Server - Database Administration*].

Operator precedence

The precedence of operators in expressions is as follows. The operators at the top of the list are evaluated before those at the bottom of the list.

1. unary operators (operators that require a single operand)
2. &, |, ^, ~
3. *, /, %
4. +, -
5. ||
6. **not**
7. **and**
8. **or**

When you use more than one operator in an expression, it is recommended that you make the order of operation explicit using parentheses.

Expressions

An expression is a statement that can be evaluated to return values.

Syntax

expression:
 case-expression
| *constant*
| [*correlation-name*.]*column-name*
| **-** *expression*
| *expression operator expression*
| **(** *expression* **)**
| *function-name* **(** *expression*, ... **)**
| *if-expression*
| *special value*
| **(** *subquery* **)**
| *variable-name*

Parameters

case-expression:
CASE *expression*
WHEN *expression*
THEN *expression*,...
[**ELSE** *expression*]
END

alternative form of case-expression:
CASE
WHEN *search-condition*
THEN *expression*, ...
[**ELSE** *expression*]
END

constant:
 integer | *number* | *string* | *host-variable*

special-value:
 CURRENT { **DATE** | **TIME** | **TIMESTAMP** }
| **NULL**
| **SQLCODE**
| **SQLSTATE**
| **USER**

if-expression:
IF *condition*
THEN *expression*
[**ELSE** *expression*]
ENDIF

operator:
{ **+** | **-** | ***** | **/** | **||** | **%** }

Remarks

Expressions are used in many different places.

Expressions are formed from several different kinds of elements. These are discussed in the sections on functions and variables. See "SQL functions" on page 123, and "Variables" on page 66.

You must be connected to the database in order evaluate expressions.

Side effects

None.

See also

- "Constants in expressions" on page 17
- "Special values" on page 58
- "Column names in expressions" on page 17
- "SQL functions" on page 123
- "Subqueries in expressions" on page 18
- "Search conditions" on page 35
- "SQL data types" on page 77
- "Variables" on page 66
- "CASE expressions" on page 18

Standards and compatibility

- See the separate descriptions of each class of expression, in the following sections.

Constants in expressions

Constants are numbers or string literals. String constants are enclosed in apostrophes ('single quotes'). An apostrophe is represented inside a string by two apostrophes in a row.

Column names in expressions

A column name is an identifier preceded by an optional correlation name. A correlation name is usually a table name. For more information about correlation names, see "FROM clause" on page 613.

If a column name has characters other than letters, digits and underscore, it must be surrounded by quotation marks (""). For example, the following are valid column names:

- Employees.Name
- address
- "date hired"
- "salary"."date paid"

See also: "Identifiers" on page 8.

Subqueries in expressions

A subquery is a SELECT statement that is nested inside another SELECT, INSERT, UPDATE, or DELETE statement, or another subquery.

If a subquery matches no rows, it evaluates to NULL.

The SELECT statement must be enclosed in parentheses, and must contain one and only one select list item. When used as an expression, a subquery is generally allowed to return only one value.

A subquery can be used anywhere that a column name can be used. For example, a subquery can be used in the select list of another SELECT statement.

For other uses of subqueries, see "Subqueries in search conditions" on page 36.

IF expressions

The syntax of the IF expression is as follows:

IF *condition*
THEN *expression1*
[**ELSE** *expression2*]
{ **ENDIF** | **END IF** }

This expression returns the following:

- If *condition* is TRUE, the IF expression returns *expression1*.
- If *condition* is FALSE, the IF expression returns *expression2*.
- If *condition* is FALSE, and there is no *expression2*, the IF expression returns NULL.
- If *condition* is UNKNOWN, the IF expression returns NULL.

condition is any valid search condition. See "Search conditions" on page 35.

For more information about TRUE, FALSE and UNKNOWN conditions, see "NULL value" on page 73, and "Search conditions" on page 35.

IF statement is different from IF expression
The IF expression is not the same as the IF statement. For information about the IF statement, see "IF statement" on page 641.

CASE expressions

The CASE expression provides conditional SQL expressions. Case expressions can be used anywhere an expression can be used.

The syntax of the CASE expression is as follows:

CASE *expression*
WHEN *expression*
THEN *expression*, ...
[**ELSE** *expression*]
{ **END** | **END CASE** }

If the expression following the CASE statement is equal to the expression following the WHEN statement, then the expression following the THEN statement is returned. Otherwise the expression following the ELSE statement is returned, if it exists.

For example, the following code uses a case expression as the second clause in a SELECT statement.

```
SELECT ID,
   ( CASE Name
      WHEN 'Tee Shirt' then 'Shirt'
      WHEN 'Sweatshirt' then 'Shirt'
      WHEN 'Baseball Cap' then 'Hat'
      ELSE 'Unknown'
   END ) as Type
FROM Products;
```

An alternative syntax is as follows:

CASE
WHEN *search-condition*
THEN *expression*, ...
[**ELSE** *expression*]
END [**CASE**]

If the search-condition following the WHEN statement is satisfied, the expression following the THEN statement is returned. Otherwise the expression following the ELSE statement is returned, if it exists.

For example, the following statement uses a case expression as the third clause of a SELECT statement to associate a string with a search-condition.

```
SELECT ID, Name,
   ( CASE
      WHEN Name='Tee Shirt' then 'Sale'
      WHEN Quantity >= 50  then 'Big Sale'
      ELSE 'Regular price'
   END ) as Type
FROM Products;
```

NULLIF function for abbreviated CASE expressions

The NULLIF function provides a way to write some CASE statements in short form. The syntax for NULLIF is as follows:

NULLIF (*expression-1*, *expression-2*)

NULLIF compares the values of the two expressions. If the first expression equals the second expression, NULLIF returns NULL. If the first expression does not equal the second expression, NULLIF returns the first expression.

CASE statement is different from CASE expression
Do not confuse the syntax of the CASE expression with that of the CASE statement. For information about the CASE statement, see "CASE statement" on page 420.

Regular expressions overview

A **regular expression** is a sequence of characters, wildcards, or operators that defines a pattern to search for within a string. SQL Anywhere supports regular expressions as part of a REGEXP or SIMILAR TO search conditions in the WHERE clause of a SELECT statement, or as an argument to the REGEXP_SUBSTR function. The LIKE search condition does not support regular expressions, although some of the wildcards and operators you can specify with LIKE resemble the regular expression wildcards and operators.

The following SELECT statement uses a regular expression (`(K|C[^h])%`) to search the Contacts table and return contacts whose last name begins with K or C, but not Ch:

```
SELECT Surname, GivenName
   FROM Contacts
   WHERE Surname SIMILAR TO '(K|C[^h])%';
```

A regular expression can include additional syntax to specify grouping, quantification, assertions, and alternation, as described below.

- **Grouping** Grouping allows you to group parts of a regular expression to apply some additional matching criteria. For example, `'(abc){2}'` matches abcabc.

  You can also use grouping to control the order in which the parts of the expression are evaluated. For example, `'ab(cdcd)'` looks first for an incidence of cdcd, and then evaluates whether the instance of cdcd is preceded by ab.

- **Quantification** Quantification allows you to control the number of times the preceding part of the expression can occur. For example, a question mark (?) is a quantifier that matches zero or one instance of the previous character. So, `'honou?r'` matches both honor and honour.

- **Assertions** Normally, searching for a pattern returns that pattern. Assertions allow you to test for the presence of a pattern, without having that pattern become part of what is returned. For example, `'SQL(?= Anywhere)'` matches SQL only if it is followed by a space and then Anywhere.

- **Alternation** Alternation allows you to specify alternative patterns to search for if the preceding pattern cannot be found. Alternate patterns are evaluated from left to right, and searching stops at the first match. For example, `'col(o|ou)r'` looks for an instance of color. If no instance is found, colour is searched for instead.

See also

- "Search conditions" on page 35
- "LIKE, REGEXP, and SIMILAR TO search conditions" on page 39
- "REGEXP search condition" on page 45
- "SIMILAR TO search condition" on page 46
- "REGEXP_SUBSTR function [String]" on page 274

Regular expressions syntax

Regular expressions are supported with the SIMILAR TO, and REGEXP search conditions, and the REGEXP_SUBSTR function. For SIMILAR TO, regular expression syntax is consistent with the ANSI/ ISO SQL standard. For REGEXP and REGEXP_SUBSTR, regular expression syntax and support is consistent with Perl 5.

Regular expressions are used by REGEXP and SIMILAR TO to match a *string*, whereas regular expressions are used by REGEXP_SUBSTR to match a *substring*. To achieve substring matching behavior for REGEXP and SIMILAR TO, you can specify wildcards on either side of the pattern you are trying to match. For example, `REGEXP '.*car.*'` matches car, carwash, and vicar. Or, you can rewrite your query to make use the REGEXP_SUBSTR function. See "REGEXP_SUBSTR function [String]" on page 274.

Regular expression matching with SIMILAR TO is case- and accent-insensitive. REGEXP and REGEXP_SUBSTR is not affected by the database accent and case sensitivity. See "LIKE, REGEXP, and SIMILAR TO: differences in character comparisons" on page 39.

Regular expressions: Metacharacters

Metacharacters are symbols or characters that have a special meaning within a regular expression.

The treatment of metacharacters can vary depending on:

- whether the regular expression is being used with the SIMILAR TO or REGEXP search conditions, or the REGEXP_SUBSTR function
- whether the metacharater is inside of a character class in the regular expression

Before continuing, you should understand the definition of a **character class**. A character class is a set of characters enclosed in square brackets, against which characters in a string are matched. For example, in the syntax `SIMILAR TO 'ab[1-9]'`, [1-9] is a character class and matches one digit in the range of 1 to 9, inclusive. The treatment of metacharacters in a regular expression can vary depending on whether the metacharater is placed inside a character class. Specifically, most metacharacters are handled as regular characters when positioned inside of a character class.

For SIMILAR TO (only), the metacharacters *, ?, +, _, |, (,), { must be escaped within a character class.

To include a literal minus sign (-), caret (^), or right-angle bracket (]) character in a character class, it must be escaped.

The list of supported regular expression metacharacters is provided below. Almost all metacharacters are treated the same when used by SIMILAR TO, REGEXP, and REGEXP_SUBSTR:

Character	Additional information
[and **]**	Left and right square brackets are used to specify a **character class**. A character class is a set of characters to match against. With the exception of the hyphen (-) and the caret (^), metacharacters and quantifiers (such as * and {m}, respectively) specified within a character class have no special meaning and are evaluated as actual characters. SQL Anywhere also supports sub-character classes such as POSIX character classes. See "Regular expressions: Special sub-character classes" on page 24.
*****	The asterisk can be used to match a character 0 or more times. For example, `REGEXP '.*abc'` matches a string that ends with abc, and starts with any prefix. So, aabc, xyzabc, and abc match, but bc and abcc do not.
?	The question mark can be used to match a character 0 or 1 times. For example, `'colou?r'` matches color and colour.
+	The plus sign can be used to match a character 1 or more times. For example, `'bre+'` matches bre and bree, but not br.
-	A hyphen can be used within a character class to denote a range. For example, `REGEXP '[a-e]'` matches a, b, c, d, and e. For details on how ranges are evaluated by REGEXP and SIMILAR TO, see "LIKE, REGEXP, and SIMILAR TO: differences in character comparisons" on page 39.
%	The percent sign can be used with SIMILAR TO to match any number of characters. The percent sign is not considered a metacharacter for REGEXP and REGEXP_SUBSTR. When specified, it matches a percent sign (%).
_ (underscore character)	The underscore can be used with SIMILAR TO to match a single character. The underscore is not considered a metacharacter for REGEXP and REGEXP_SUBSTR. When specified, it matches an underscore (_).
|	The pipe symbol is used to specify alternative patterns to use for matching the string. In a string of patterns separated by a vertical bar, the vertical bar is interpreted as an OR and matching stops at the first match made starting from the leftmost pattern. So, you should list the patterns in descending order of preference. You can specify an unlimited number of alternative patterns.
(and **)**	Left and right parenthesis are metacharacters when used for grouping parts of the regular expression. For example, `(ab)*` matches zero or more repetitions of ab. As with mathematical expressions, you use grouping to control the order in which the parts of a regular expression are evaluated.

Character	Additional information
{ and **}**	Left and right curly braces are metacharacters when used for specifying **quantifiers**. Quantifiers specify the number of times a pattern must repeat to constitute a match. For example: ● ***{m}*** Matches a character exactly *m* times. For example, `'519-[0-9]{3}-[0-9]{4}'` matches a phone number in the 519 area code (providing the data is formatted in the manner defined in the syntax). ● ***{m,}*** Matches a character at least *m* times. For example, `'[0-9]{5,}'` matches any string of five or more digits. ● ***{m,n}*** Matches a character at least *m* times, but not more than *n* times. For example, `SIMILAR TO '_{5,10}'` matches any string with between 5 and 10 (inclusive) characters.
\	The backslash is used as an escape character for metacharacters. It can also be used to escape non-metacharacters.
^	For REGEXP and REGEXP_SUBSTR, when a caret is outside a character class, the caret matches the start of a string. For example, `'^[hc]at'` matches hat and cat, but only at the beginning of the string. When used inside a character class, the following behavior applies: ● **REGEXP and REGEXP_SUBSTR** When the care is the first character in a character class, it matches anything other than the characters in the character set. For example, `REGEXP '[^abc]'` matches any character other than a, b, or c. If the caret is not the first character inside the square brackets, it matches a caret. For example, `REGEXP_SUBSTR '[a-e^c]'` matches a, b, c, d, e, and ^. ● **SIMILAR TO** For SIMILAR TO, the caret is treated as a subtraction operator. For example, `SIMILAR TO '[a-e^c]'` matches a, b, d, and e.
$	When used with REGEXP and REGEXP_SUBSTR, matches the end of a string. For example, `SIMILAR TO 'cat$'` matches cat, but not catfish. When used with SIMILAR TO, it matches a question mark.
.	When used with REGEXP and REGEXP_SUBSTR, matches any single character. For example, `REGEXP 'a.cd'` matches any string of four characters that starts with a and ends with cd. When used with SIMILAR TO, matches a period (.).
:	The colon is used within a character set to specify a subcharacter class. For example, `'[[:alnum:]]'`.

Regular expressions: Special sub-character classes

Sub-character classes are special character classes embedded within a larger character class. In addition to custom character classes where you define the set of characters to match (for example, `[abxq4]` limits the set of matching characters to a, b, x, q, and 4), SQL Anywhere supports sub-character classes such as most of the POSIX character classes. For example, `[[:alpha:]]` represents the set of all upper- and lowercase letters.

The REGEXP search condition and the REGEXP_SUBSTR function support all the syntax conventions in the table below, but the SIMILAR TO search expression does not. Conventions supported by SIMILAR TO have a Y in the SIMILAR TO column.

In REGEXP and when using the REGEXP_SUBSTR function, sub-character classes can be negated using a caret. For example, `[[:^alpha:]]` matches the set of all characters except alpha characters.

Sub-character class	Additional information	SIMILAR TO
[:alpha:]	Matches upper- and lowercase alphabetic characters in the current collation. For example, `'[0-9]{3}[[:alpha:]]{2}'` matches three digits, followed by two letters.	Y
[:alnum:]	Match digits, and upper- and lowercase alphabetic characters in the current collation. For example, `'[[:alnum:]]+'` matches a string of one or more letters and numbers.	Y
[:digit:]	Match digits in the current collation. For example, `'[[:digit:]-]+'` matches a string of one or more digits or dashes. Likewise, `'[^[:digit:]-]+'` matches a string of one or more characters that are not digits or dashes.	Y
[:lower:]	Match lowercase alphabetic characters in the current collation. For example, `'[[:lower:]]'` does not match A because A is uppercase.	Y

[:space:]	Match a single blank (' '). For example, the following statement searches Contacts.City for any city with a two word name:
	
	`SELECT City`
	`FROM Contacts`
	`WHERE City REGEXP '.*[[:space:]].*';`
		Y

[:upper:]	Match uppercase alphabetic characters in the current collation. For example, `'[[:upper:]ab]'` matches one of: any upper case letter, a, or b.	Y
[:whitespace:]	Match a whitespace character such as space, tab, formfeed, and carriage return.	Y
[:ascii:]	Match any seven-bit ASCII character (ordinal value between 0 and 127).	

Sub-character class	Additional information	SIMILAR TO
[:blank:]	Match a blank space, or a horizontal tab. `[[:blank:]]` is equivalent to `[ \t]`.	
[:cntrl:]	Match ASCII characters with an ordinal value of less than 32, or character value 127 (control characters). Control characters include newline, form feed, backspace, and so on.	
[:graph:]	Match printed characters. `[[:graph:]]` is equivalent to `[[:alnum:][:punct:]]`.	
[:print:]	Match printed characters and spaces. `[[:print:]]` is equivalent to `[[:graph:][:whitespace:]]`.	
[:punct:]	Match one of: !"#$%&'()*+,-./:;<=>?@[\]^_`{|}~. The `[:punct:]` sub-character class may not include non-ASCII punctuation characters available in the current collation.	
[:word:]	Match alphabetic, digit, or underscore characters in the current collation. `[[:word:]]` is equivalent to `[[:alnum:]_]`.	
[:xdigit:]	Match a character that is in the character class [0-9A-Fa-f].	

Regular expressions: Other supported syntax conventions

The following syntax conventions are supported by the REGEXP search condition and the REGEXP_SUBSTR function, and they assume that the backslash is the escape character. *These conventions are not supported by the SIMILAR TO search expression.*

Regular expression syntax	Name and meaning
\0*xxx*	Matches the character whose value is \0*xxx*, where *xxx* is any sequence of octal digits, and 0 is a zero. For example, `\0134` matches a backslash.
\a	Matches the bell character.
\A	Used outside a character set to match the start of a string. Equivalent to `^` used outside a character set.
\b	Matches a backspace character.

Regular expression syntax	Name and meaning
\B	Matches the backslash character (\\).
\c*X*	Matches a named control character. For example, \cZ for ctrl-Z.

\d	Matches a digit in the current collation. For example, the following statement searches Contacts.Phone for all phone numbers that end with 00:
	
	`SELECT Surname, Surname, City, Phone`
	`   FROM Contacts`
	`   WHERE Phone REGEXP '\\d{8}00';`
	
	\d can be used both inside and outside character classes, and is equivalent to `[[:digit:]]`.

\D	Matches anything that is not a digit. This is the opposite of \d. \D can be used both inside and outside character classes, and is equivalent to `[^[:digit:]]`. Be careful when using the negated shorthands inside square brackets. `[\D\S]` is not the same as `[^\d\s]`. The latter matches any character that is not a digit or whitespace. So it matches x, but not 8. The former, however, matches any character that is either not a digit, or is not whitespace. Because a digit is not whitespace, and whitespace is not a digit, `[\D\S]` matches any character, digit, whitespace or otherwise.
\e	Matches the escape character.
\E	Ends the treatment of metacharacters as non-metacharacters, initiated by a \Q. For a list of regular expression metacharacters, see "Regular expressions: Metacharacters" on page 21.
\f	Matches a form feed.
\n	Matches a new line.

Regular expression syntax	Name and meaning
\Q	Treat all metacharacters as non-metacharacters, until \E is encountered. For example, `\Q[$\E` is equivalent to `\[\$`. For a list of regular expression metacharacters, see "Regular expressions: Metacharacters" on page 21.
\r	Matches a carriage return.

\s	Matches a space or a character treated as whitespace. For example, the following statement returns all product names from Products.ProductName that have at least one space in the name:
	
	`SELECT Name`
	`FROM Products`
	`WHERE Name REGEXP '.*\\s.*'`
	
	\s can be used both inside and outside character classes, and is equivalent to `[[:whitespace:]]`. See "Regular expressions: Special sub-character classes" on page 24.

\S	Matches a non-whitespace character. This is the opposite of \s, and is equivalent to `[^[:whitespace:]]`. \S can be used both inside and outside character classes. See "Regular expressions: Special sub-character classes" on page 24. Be careful when using the negated shorthands inside square brackets. `[\D\S]` is not the same as `[^\d\s]`. The latter matches any character that is not a digit or whitespace. So it matches x, but not 8. The former, however, matches any character that is either not a digit, or is not whitespace. Because a digit is not whitespace, and whitespace is not a digit, `[\D\S]` matches any character, digit, whitespace or otherwise.
\t	Matches a horizontal tab.
\v	Matches a vertical tab.

Regular expression syntax	Name and meaning

\w	Matches a alphabetic character, digit, or underscore in the current collation. For example, the following statement returns all surnames from Contacts.Surname that are exactly seven alpha-numeric characters in length:
	`SELECT Surname`
	`FROM Contacts`
	`WHERE Surname REGEXP '\\w{7}';`
	\w can be used both inside and outside character classes. See "Regular expressions: Special sub-character classes" on page 24.
	Equivalent to `[[:alnum:]_]`..

\W	Matches anything that is not an alphabetic character, digit, or underscore in the current collation. This is the opposite of \w, and is equivalent to `[^[:alnum:]_]`. This regular expression can be used both inside and outside character classes. See "Regular expressions: Special sub-character classes" on page 24.
\x*hh*	Matches the character whose value is 0x*hh*, where *hh* is, at most, two hex digits. For example, `\x2D` is equivalent to a hyphen. Equivalent to \x{*hh*}.
\x{*hhh***}**	Matches the character whose value is 0x*hhh*, where *hhh* is, at most, eight hex digits.
\z and **\Z**	Matches the position (not character) at the end of the string. Equivalent to $.

Regular expressions: Assertions

Assertions test whether a condition is true, and affect the position in the string where matching begins. Assertions do not return characters; the assertion pattern is not included in the final match. These assertions are supported by the REGEXP search condition and the REGEXP_SUBSTR function. These conventions are not supported by the SIMILAR TO search expression.

Lookahead and lookbehind assertions can be useful with REGEXP_SUBSTR when trying to split a string. For example, you can return the list of street names (without the street numbers) in the Address column of the Customers table by executing the following statement:

```
SELECT REGEXP_SUBSTR( Street, '(?<=^\\S+\\s+).*$' )
FROM Customers;
```

Another example is if you want to use a regular expression to verify that a password conforms to certain rules. You could use a zero width assertion similar to the following:

```
IF password REGEXP '(?=.*[[:digit:]])(?=.*[[:alpha:]].*[[:alpha:]])[[:word:]]
{4,12}'
   MESSAGE 'Password conforms' TO CLIENT;
ELSE
   MESSAGE 'Password does not conform' TO CLIENT;
END IF
```

The password is valid when the following are true:

- *password* has at least one digit (zero width positive assertion with [[:digit:]])
- *password* has at least two alphabetic characters (zero width positive assertion with [[:alpha:]].*[[:alpha:]])
- *password* contains only alpha-numeric or underscore characters ([[:word:]])
- *password* is at least 4 characters, and at most 12 characters ({4,12})

The following table contains the assertions supported by SQL Anywhere:

Syntax	Meaning
(?=*pattern***)**	**Positive lookahead zero-width assertion** Looks to see if the current position in the string is immediately followed by an occurrence of *pattern*, without *pattern* becoming part of the match string. `'A(?=B)'` matches an A that is followed by a B, without making the B part of the match. For example, `SELECT REGEXP_SUBSTR( 'in new york city', 'new(?=\\syork)');` returns the substring new since it is immediately followed by ' york' (note the space before york).
(?!*pattern***)**	**Negative lookahead zero-width assertions** Looks to see if the current position in the string is *not* immediately followed by an occurrence of *pattern*, without *pattern* becoming part of the match string. So, `'A(?!B)'` matches an A that is not followed by a B. For example,`SELECT REGEXP_SUBSTR('new jersey', 'new(?!\\syork)');` returns the substring new.
(?<=*pattern***)**	**Positive lookbehind zero-width assertions** Looks to see if the current position in the string is immediately preceded by an occurrence of *pattern*, without *pattern* becoming part of the match string. So, `'(?<=A)B'` matches a B that is immediately preceded by an A, without making A part of the match. For example, `SELECT REGEXP_SUBSTR('new york', '(?<=new\\s)york');` returns the substring york.

Syntax	Meaning
(?<!*pattern***)**	**Negative lookbehind zero-width assertions** Looks to see if the current position in the string is *not* immediately preceded by an occurrence of *pattern*, without *pattern* becoming part of the match string. For example, `SELECT REGEXP_SUBSTR('about york', '(?<!new\\s)york');` returns the substring york.
(?>*pattern***)**	**Possessive local subexpression** Matches only the largest prefix of the remaining string that matches *pattern*. For example, in `'aa' REGEXP '(?>a*)a'`, `(?>a*)` matches (and consumes) the aa, and never just the leading a. As a result, `'aa' REGEXP '(?>a*)a'` evaluates to false.
(?:*pattern***)**	**Non-capturing block** This is functionally equivalent to just *pattern*, and is provided for compatibility. For example, in `'bb' REGEXP '(?:b*)b'`, `(?:b*)` matches (and consumes) the bb. However, unlike possessive local subexpression, the last b in bb is given up to allow the whole match to succeed (that is, to allow the matching to the b found outside the non-capturing block). Likewise, `'a(?:bc|b)c'` matches abcc, and abc. In matching abc, backtracking on the final c in bc takes place so that the c outside the group can be used to make the match successful.
(?#*text***)**	Used for comments. The content of *text* is ignored.

See also

- "Regular expression examples" on page 30

Regular expression examples

The following table shows example uses of regular expressions. All examples work for REGEXP and some also work for SIMILAR TO, as noted in the Example column. Results vary depending on the search condition you use for searching. For those that work with SIMILAR TO, results can vary further depending on case and accent sensitivity.

For a comparison of how REGEXP and SIMILAR TO handle matches and evaluate ranges, see "LIKE, REGEXP, and SIMILAR TO search conditions" on page 39.

Note that backslashes should be doubled if the examples are used in literal strings (for example, `'.+@.+\\..+'`)

Example	Sample matches

Credit Card Numbers (REGEXP only):
Visa:
4[0-9]{3}\s[0-9]{4}\s[0-9]{4}\s[0-9]{4}
MasterCard:
5[0-9]{3}\s[0-9]{4}\s[0-9]{4}\s[0-9]{4}
American Express:
37[0-9]{2}\s[0-9]{4}\s[0-9]{4}\s[0-9]{4}
Discover:
6011\s[0-9]{4}\s[0-9]{4}\s[0-9]{4}
	Matches (Visa): 4123 6453 2222 1746
	Non-Matches (Visa):
	3124 5675 4400 4567, 4123-6453-2222-1746
	Similarly, MasterCard matches a set of 16 numbers, starting with 5, with a space between each subset of four numbers. American Express and Discover are the same, but must start with 37 and 6011 respectively.

Dates (REGEXP and SIMILAR TO):
([0-2][0-9]|30|31)/(0[1-9]|1[0-2])/[0-9]{4}
	Matches: 31/04/1999, 15/12/4567
	Non-Matches: 31/4/1999, 31/4/99, 1999/04/19, 42/67/25456

Windows absolute paths (REGEXP only): ([A-Za-z]:|\\\\)\\\\[[:alnum:][:whitespace:]!"#$%&'()+,-.\\\\;=@\\[\\]^_`{}~.]*	Matches: \\\\server\\share\\file Non-Matches: \\directory\\directory2, /directory2

Email Addresses (REGEXP only):
[[:word:]\\-.]+@[[:word:]\\-.]+\\.[[:alpha:]]{2,3}
	Matches: abc.123@def456.com, _123@abc.ca
	Non-Matches: abc@dummy, ab*cd@efg.hijkl

Email Addresses (REGEXP only): .+@.+\\..+	Matches: *@qrstuv@wxyz.12345.com, __1234^%@@abc.def.ghijkl Non-Matches: abc.123.*&ca, ^%abcdefg123

HTML Hexadecimal Color Codes (REGEXP and SIMILAR TO):
[A-F0-9]{6}
	Matches: AB1234, CCCCCC, 12AF3B
	Non-Matches: 123G45, 12-44-CC

HTML Hexadecimal Color Codes (REGEXP only):
[A-F0-9]{2}\s[A-F0-9]{2}\s[A-F0-9]{2}
	Matches: AB 11 00, CC 12 D3
	Non-Matches: SS AB CD, AA BB CC DD, 1223AB

Example	Sample matches

IP Addresses (REGEXP only):
`((2(5[0-5]|[0-4][0-9])|1([0-9][0-9])|([1-9][0-9])|[0-9])\.){3}(2(5[0-5]|[0-4][0-9])|1([0-9][0-9])|([1-9][0-9])|[0-9])`
	Matches: 10.25.101.216
	Non-Matches: 0.0.0, 256.89.457.02

Java Comments (REGEXP only):
`/\*.*\*/|//[^\n]*`
	Matches Java comments that are between /* and */, or one line comments prefaced by //.
	Non-Matches: a=1

Money (REGEXP only):
`(\+|-)?\$[0-9]*\.[0-9]{2}`
	Matches: $1.00, -$97.65
	Non-Matches: $1, 1.00$, $-75.17

Positive, negative numbers, and decimal values (REGEXP only): `(\+|-)?[0-9]+(\.[0-9]+)?`	Matches: +41, -412, 2, 7968412, 41, +41.1, -3.141592653 Non-Matches: ++41, 41.1.19, -+97.14

Passwords (REGEXP and SIMILAR TO):
`[[:alnum:]]{4,10}`
	Matches: abcd, 1234, A1b2C3d4, 1a2B3
	Non-Matches: abc, *ab12, abcdefghijkl

Passwords (REGEXP only):
`[a-zA-Z]\w{3,7}`
	Matches: AB_cd, A1_b2c3, a123_
	Non-Matches: *&^g, abc, 1bcd

Phone Numbers (REGEXP and SIMILAR TO):
`([2-9][0-9]{2}-[2-9][0-9]{2}-[0-9]{4})|([2-9][0-9]{2}\s[2-9][0-9]{2}\s[0-9]{4})`
	Matches: 519-883-6898, 519 888 6898
	Non-Matches: 888 6898, 5198886898, 519 883-6898

Sentences (REGEXP only):
`[A-Z0-9].*(\.|\?|!)`
	Matches: Hello, how are you?
	Non-Matches: i am fine

Sentences (REGEXP only):
`[[:upper:]0-9].*[.?!]`
	Matches: Hello, how are you?
	Non-Matches: i am fine

Social Security Numbers (REGEXP and SIMILAR TO):
`[0-9]{3}-[0-9]{2}-[0-9]{4}`
	Matches: 123-45-6789
	Non-Matches:123 45 6789, 123456789, 1234-56-7891

Example	Sample matches
URLs (REGEXP only): `(http://)?www\.[a-zA-Z0-9]+\.[a-zA-Z]{2,3}`	Matches: http://www.sample.com, www.sample.com Non-Matches: http://sample.com, http://www.sample.comm

See also

- "Regular expressions syntax" on page 21

Compatibility of expressions

Default interpretation of delimited strings

SQL Anywhere employs the SQL/2003 convention, that strings enclosed in apostrophes are constant expressions, and strings enclosed in quotation marks (double quotes) are delimited identifiers (names for database objects).

The quoted_identifier option

SQL Anywhere provides a quoted_identifier option that allows the interpretation of delimited strings to be changed. By default, the quoted_identifier option is set to On in SQL Anywhere. See "quoted_identifier option [compatibility]" [*SQL Anywhere Server - Database Administration*].

You cannot use SQL reserved words as identifiers if the quoted_identifier option is Off.

For a complete list of reserved words, see "Reserved words" on page 4.

Setting the option

The following statement changes the setting of the quoted_identifier option to On:

```
SET quoted_identifier On;
```

The following statement changes the setting of the quoted_identifier option to Off:

```
SET quoted_identifier Off;
```

Compatible interpretation of delimited strings

You can choose to use either the SQL/2003 or the default Transact-SQL convention in SQL Anywhere as long as the quoted_identifier option is set to the same value in each DBMS.

Examples

If you choose to operate with the quoted_identifier option On (the default setting), then the following statements involving the SQL keyword **user** are valid for both DBMSs.

```
CREATE TABLE "user" ( col1 char(5) ) ;
INSERT "user" ( col1 )
   VALUES ( 'abcde' );
```

If you choose to operate with the quoted_identifier option off then the following statements are valid for both DBMSs.

```
SELECT *
FROM Employees
WHERE Surname = "Chin":
```

Search conditions

A search condition is the criteria specified for a WHERE clause, a HAVING clause, a CHECK clause, an ON phrase in a join, or an IF expression. A search condition is also known as a **predicate**.

Syntax

```
search-condition :
 expression comparison-operator expression
| expression comparison-operator { [ ANY | SOME ] | ALL } ( subquery )
| expression IS [ NOT ] NULL
| expression [ NOT ] BETWEEN expression AND expression
| expression [ NOT ] LIKE pattern [ ESCAPE expression ]
| expression [ NOT ] SIMILAR TO pattern [ ESCAPE escape-expression ]
| expression [ NOT ] REGEXP pattern [ ESCAPE escape-expression ]
| expression [ NOT ] IN ( { expression
     | subquery
     | value-expression1 , ... } )
| CONTAINS (column-name [,... ] , query-string )
| EXISTS ( subquery )
| NOT condition
| search-condition [ { AND | OR } search-condition ] [ ... ]
| ( search-condition )
| ( search-condition , estimate )
|  search-condition IS [ NOT ] { TRUE | FALSE | UNKNOWN }
| trigger-operation

comparison-operator :
 =
| >
| <
| >=
| <=
| <>
| !=
| !<
| !>

trigger-operation :
INSERTING
| DELETING
| UPDATING [ ( column-name-string ) ]
| UPDATE( column-name )
```

Parameters

- **ALL search condition** See “ALL search condition” on page 37.
- **ANY and SOME search conditions** See “ANY and SOME search conditions” on page 37.
- **BETWEEN search condition** See “BETWEEN search condition” on page 38.
- **CONTAINS search condition** See “CONTAINS search condition” on page 48.
- **EXISTS search condition** See “EXISTS search condition” on page 54.
- **LIKE search condition** See “LIKE search condition” on page 41.

- **SIMILAR TO search condition** See "SIMILAR TO search condition" on page 46.
- **REGEXP search condition** See "REGEXP search condition" on page 45.

Remarks

Search conditions are used to choose a subset of the rows from a table, or in a control statement such as an IF statement to determine control of flow.

In SQL, every condition evaluates as one of TRUE, FALSE, or UNKNOWN. This is called three-valued logic. The result of a comparison is UNKNOWN if either value being compared is the NULL value. For tables displaying how logical operators combine in three-valued logic, see "Three-valued logic" on page 55.

Rows satisfy a search condition if and only if the result of the condition is TRUE. Rows for which the condition is UNKNOWN or FALSE do not satisfy the search condition. For more information about NULL, see "NULL value" on page 73.

Subqueries form an important class of expression that is used in many search conditions. For information about using subqueries in search conditions, see "Subqueries in search conditions" on page 36.

The different types of search condition are discussed in the following sections.

The LIKE, SIMILAR TO, and REGEXP search conditions are very similar. To understand similarities and differences between them, see "LIKE, REGEXP, and SIMILAR TO search conditions" on page 39.

Permissions

Must be connected to the database.

Side effects

None.

See also

- "Expressions" on page 16

Subqueries in search conditions

Subqueries that return exactly one column and either zero or one row can be used in any SQL statement wherever a column name could be used, including in the middle of an expression.

For example, expressions can be compared to subqueries in comparison conditions as long as the subquery does not return more than one row. If the subquery (which must have exactly one column) returns one row, then the value of that row is compared to the expression. If a subquery returns no rows, the value of the subquery is NULL.

Subqueries that return exactly one column and any number of rows can be used in IN, ANY, ALL, and SOME search conditions. Subqueries that return any number of columns and rows can be used in EXISTS search conditions. These search conditions are discussed in the following sections.

See also

- "Comparison operators" on page 12

ALL search condition

Syntax

expression comparison-operator **ALL (** *subquery* **)**

comparison-operator:
 =
| **>**
| **<**
| **>=**
| **<=**
| **<>**
| **!=**
| **!<**
| **!>**

Remarks

With the ALL search condition, if the value of subquery result set is the empty set, the search condition evaluates to TRUE. Otherwise, the search condition evaluates to TRUE, FALSE, or UNKNOWN, depending on the value of *expression*, and the result set returned by the subquery, as follows:

If the expression value is..	and the result set returned by the subquery contains at least one NULL, then..	or the result set returned by the subquery contains no NULLs, then..
NULL	UNKNOWN	UNKNOWN
not NULL	If there exists at least one value in the subquery result set for which the comparison with the expression value is FALSE, then the search condition evaluates to FALSE. Otherwise, the search condition evaluates to UNKNOWN.	If there exists at least one value in the subquery result set for which the comparison with the expression value is FALSE, then the search condition evaluates to FALSE. Otherwise, the search condition evaluates to TRUE.

ANY and SOME search conditions

Syntax

expression comparison-operator { **ANY** | **SOME** }**(** *subquery* **)**

comparison-operator:
 =

| >
| <
| >=
| <=
| <>
| !=
| !<
| !>

Remarks

The keywords ANY and SOME are synonymous.

With the ANY and SOME search conditions, if the subquery result set is the empty set, the search condition evaluates to FALSE. Otherwise, the search condition evaluates to TRUE, FALSE, or UNKNOWN, depending on the value of *expression*, and the result set returned by the subquery, as follows:

If the expression value is..	and the result set returned by the subquery contains at least one NULL, then..	or the result set returned by the subquery contains no NULLs, then..
NULL	UNKNOWN	UNKNOWN
not NULL	If there exists at least one value in the subquery result set for which the comparison with the expression value is TRUE, then the search condition evaluates to TRUE. Otherwise, the search condition evaluates to UNKNOWN.	If there exists at least one value in the subquery result set for which the comparison with the expression value is TRUE, then the search condition evaluates to TRUE. Otherwise, the search condition evaluates to FALSE.

An ANY or SOME search condition with an equality operator, evaluates to TRUE if *expression* is equal to any of the values in the result of the subquery, and FALSE if the value of the expression is not NULL, does not equal any of the values in the result of the subquery, and the result set doesn't contain NULLs.

Note
The usage of **= ANY** or **= SOME** is equivalent to using the IN keyword.

BETWEEN search condition

Syntax

expression [**NOT**] **BETWEEN** *start-expression* **AND** *end-expression*

Remarks

The BETWEEN search condition can evaluate as TRUE, FALSE, or UNKNOWN. Without the NOT keyword, the search condition evaluates as TRUE if *expression* is between *start-expression* and *end-*

expression. The NOT keyword reverses the meaning of the search condition but leaves UNKNOWN unchanged.

The BETWEEN search condition is equivalent to a combination of two inequalities:

[**NOT**] (*expression* >= *start-expression* **AND** *expression* <= *end-expression*)

LIKE, REGEXP, and SIMILAR TO search conditions

The REGEXP, LIKE, and SIMILAR TO search conditions are similar in that they all attempt to match a pattern to a string. Also, all three attempt to match an entire string, not a substring within the string.

The basic syntax for all three search conditions is similar:

expression search-condition pattern

LIKE, REGEXP, and SIMILAR TO: differences in pattern definition

REGEXP, LIKE, and SIMILAR TO search conditions differ in how you define *pattern*:

- REGEXP supports a superset of regular expression syntax supported by SIMILAR TO. In addition, for compatibility with other products, the REGEXP search condition supports several syntax extensions. Also, REGEXP and SIMILAR TO have a different default escape character. REGEXP behavior matches closely with Perl 5 (except where Perl syntax and operators are not supported).
- LIKE syntax for *pattern* is simple and supports a small set of wildcards, but does not support the full regular expression syntax.
- SIMILAR TO syntax for *pattern* allows a robust pattern matching using the regular expression syntax defined in the ANSI/ISO SQL standard.

LIKE, REGEXP, and SIMILAR TO: differences in character comparisons

When performing comparisons, REGEXP behavior is different from LIKE and SIMILAR TO. For REGEXP comparisons, the database server uses code point values in the **database character set** for comparisons. This is consistent with other regular expression implementations such as Perl.

For LIKE and SIMILAR TO, the database server uses the equivalence and sort order in the **database collation** for comparisons. This is consistent with how the database evaluates comparison operators such as > and =.

The difference in character comparison methods means that results for matching and range evaluation for REGEXP and LIKE/SIMILAR differ as well.

- **Differences in matching** Since REGEXP uses code point values, it only matches a literal in a pattern if it is the exact same character. REGEXP matching is therefore not impacted by such things as datebase collation case-sensitivity or accent sensitivity. For example, 'A' could never be returned as a match for 'a'.

  Since LIKE and SIMILAR TO use the database collation, results are impacted by case- and accent-sensitivity when determining character equivalence. For example, if the database collation is case- and accent-insensitive, matches are case- and accent-insensitive. So, an 'A' could be returned as a match for 'a'.

- **Differences in range evaluation** Since REGEXP uses code points for range evaluation, a character is considered to be in the range if its code point value is equal to, or between, the code point values for the start and end of the range. For example, the comparison `x REGEXP '[A-C]'`, for the single character *x*, is equivalent to `CAST(x AS BINARY) >= CAST(A AS BINARY) AND CAST(x AS BINARY) <= CAST(C AS BINARY)` .

  Since LIKE and SIMILAR TO use the collation sort order for range evaluation, a character is considered to be in the range if its position in the collation is the same as, or between, the position of the start and end characters for the range. For example, the comparison `x SIMILAR TO '[A-C]'` (where *x* is a single character) is equivalent to `x >= A AND x <= C`, and the comparison operators are evaluated using the collation sort ordering.

  The following table shows the set of characters included in the range `'[A-C]'` as evaluated by LIKE, SIMILAR TO, and REGEXP. Both databases use the 1252LATIN1 collation, but the first database is case-insensitive, while the second one is case sensitive.

	LIKE/SIMILAR TO '[A-C]'	**REGEXP '[A-C]'**
demo.db (case-insensitive)	A,B,C,a,b,c,ª,À,Á,Â,Ã,Ä,Å,Æ,Ç,à,á,â,ã,ä,å,æ,ç	A, B, C
charsensitive.db (case-sensitive)	A,B,C,b,c,À,Á,Â,Ã,Ä,Å,Æ,Ç,ç	A, B, C

  The following can be observed in the results:

  - LIKE and SIMILAR TO include accented characters in the range.
  - LIKE and SIMILAR TO include different characters depending on database case-sensitivity. Specifically, they include any lower case letters found within the range, which you may not have anticipated when searching on a case-sensitive database.

    Similarly, on a case-insensitive database, some characters you assume would be included in the range are not. For example, `SIMILAR TO '[a-c]'` on a case-insensitve database includes a, A, b, B, c, but not C because C occurs after lower case c in the sort order.
  - REGEXP returns only A, B, C regardless of database case sensitivity. If you want the range to include lower case letters, you must add them to the range definition. For example, `REGEXP '[a-cA-C]'`.
  - the REGEXP set of characters does not change, regardless of database case-sensitivity.

  Even though your database uses a different collation, or has different case- or accent-sensitivity settings than the examples above, you can perform a similar test to see what is returned by LIKE, SIMILAR TO, or REGEXP by connecting to the database and executing any of these statements:

  ```
  SELECT CHAR( row_num ) FROM RowGenerator WHERE CHAR( row_num ) LIKE '[A-
  C]';
  SELECT CHAR( row_num ) FROM RowGenerator WHERE CHAR( row_num ) REGEXP '[A-
  C]';
  SELECT CHAR( row_num ) FROM RowGenerator WHERE CHAR( row_num ) SIMILAR TO
  '[A-C]';
  ```

See also

- "Regular expressions overview" on page 20
- "Regular expressions syntax" on page 21
- "Regular expression examples" on page 30

LIKE search condition

Syntax

The syntax for the LIKE search condition is as follows:

expression [**NOT**] **LIKE** *pattern* [**ESCAPE** *escape-character*]

Parameters

- **expression** The string to be searched.
- **pattern** The pattern to search for within *expression*.
- **escape-character** The character to use to escape special characters such as underscores and percent signs. The default escape character is the null character, which can be specified in a string literal as '\x00'.

Remarks

The LIKE search condition attempts to match *expression* with *pattern* and evaluates to TRUE, FALSE, or UNKNOWN.

The search condition evaluates to TRUE if *expression* matches *pattern* (assuming NOT was not specified). If either *expression* or *pattern* is the NULL value, the search condition evaluates to UNKNOWN. The NOT keyword reverses the meaning of the search condition, but leaves UNKNOWN unchanged.

expression is interpreted as a CHAR or NCHAR string. The entire contents of *expression* is used for matching. Similarly, *pattern* is interpreted as a CHAR or NCHAR string and can contain any number of the supported wildcards from the following table:

Wildcard	Matches
_ (underscore)	Any one character. For example, `a_` matches ab and ac, but not a.
% (percent)	Any string of zero or more characters. For example, `bl%` matches bl and bla.
[]	Any single character in the specified range or set. For example, `T[oi]m` matches Tom or Tim.
[^]	Any single character *not* in the specified range or set. For example, `M[^c]` matches Mb and Md, but not Mc.

All other characters must match exactly.

For example, the following search condition returns TRUE for any row where name starts with the letter a and has the letter b as its second last character.

```
... name LIKE 'a%b_'
```

If *escape-character* is specified, it must evaluate to a single-byte CHAR or NCHAR character. The escape character can precede a percent, an underscore, a left square bracket, or another escape character in the *pattern* to prevent the special character from having its special meaning. When escaped in this manner, a percent matches a percent, and an underscore matches an underscore.

All patterns of 126 bytes or less are supported. Patterns of greater than 126 bytes that do not contains wildcards are not supported. Patterns containing wildcard characters that are longer than 126 bytes are supported, depending on the contents of the pattern. The number of bytes used to represent the pattern depends on whether the pattern is CHAR or NCHAR.

Different ways to use the LIKE search condition

To search for	Example	Additional information
One of a set of characters	`LIKE 'sm[iy]th'`	A set of characters to look for is specified by listing the characters inside square brackets. In this example, the search condition matches *smith* and *smyth*.
One of a range of characters	`LIKE '[a-r]ough'`	A range of characters to look for is specified by giving the ends of the range inside square brackets, separated by a hyphen. In this example, the search condition matches bough and rough, but not tough. The range of characters [a-z] is interpreted as "greater than or equal to a, and less than or equal to z", where the greater than and less than operations are carried out within the collation of the database. For information about matching ranges, see "LIKE, REGEXP, and SIMILAR TO: differences in character comparisons" on page 39. The lower end of the range must precede the higher end of the range. For example, `[z-a]` does not match anything because no character matches the [z-a] range.
Ranges and sets combined	`... LIKE '[a-rt]ough'`	You can combine ranges and sets within square brackets. In this example, `... LIKE '[a-rt]ough'` matches bough, rough, and tough. The pattern [a-rt] is interpreted as exactly one character that is either in the range a to r inclusive, or is t.

To search for	Example	Additional information
One character not in a range	`... LIKE '[^a-r]ough'`	The caret character (^) is used to specify a range of characters that is excluded from a search. In this example, `LIKE '[^a-r]ough'` matches the string tough, but not the strings rough or bough. The caret negates the rest of the contents of the brackets. For example, the bracket [^a-rt] is interpreted as exactly one character that is not in the range a to r inclusive, and is not t.
Search patterns with trailing blanks	`'90 '`, `'90[ ]'` and `'90_'`	When your search pattern includes trailing blanks, the database server matches the pattern only to values that contain blanks—it does not blank pad strings. For example, the patterns '90 ', '90[ ]', and '90_' match the expression '90 ', but do not match the expression '90', even if the value being tested is in a CHAR or VARCHAR column that is three or more characters in width.

Special cases of ranges and sets

Any single character in square brackets means that character. For example, `[a]` matches just the character a. `[^]` matches just the caret character, `[%]` matches just the percent character (the percent character does not act as a wildcard in this context), and `[_]` matches just the underscore character. Also, `[[]` matches just the character [.

Other special cases are as follows:

- The pattern `[a-]` matches either of the characters a or -.
- The pattern `[]` is never matched and always returns no rows.
- The patterns `[` or `[abp-q` return syntax errors because they are missing the closing bracket.
- You cannot use wildcards inside square brackets. The pattern `[a%b]` finds one of a, %, or b.
- You cannot use the caret character to negate ranges except as the first character in the bracket. The pattern `[a^b]` finds one of a, ^, or b.

Case sensitivity and how comparisons are performed

If the database collation is case sensitive, the search condition is also case sensitive. To perform a case insensitive search with a case sensitive collation, you must include upper and lower characters. For example, the following search condition evaluates to true for the strings Bough, rough, and TOUGH:

```
LIKE '[a-zA-Z][oO][uU][gG][hH]'
```

Comparisons are performed character-by-character, unlike the equivalence (=) operator and other operators where the comparison is done string-by-string. For example, when a comparison is done in a UCA collation (CHAR or NCHAR with the collation set to UCA), `'Æ'='AE'` is true, but `'Æ' LIKE 'AE'` is false.

For a character-by-character comparison to match, each single character in the expression being searched must match a single character (using the collation's character equivalence), or a wildcard in the LIKE expression.

For a comparison of how matching and range evaluations are handled for LIKE, SIMILAR TO, and REGEXP, see "LIKE, REGEXP, and SIMILAR TO search conditions" on page 39.

National character (NCHAR) support

LIKE search conditions can be used to compare CHAR and NCHAR strings. In this case, character set conversion is performed so that the comparison is done using a common data type. Then, a character-by-character comparison is performed. See "Comparisons between CHAR and NCHAR" on page 112.

You can specify *expression* or *pattern* as an NCHAR string literal by prefixing the quoted value with N (for example, `*expression* LIKE N'*pattern*'`). You can also use the CAST function to cast the pattern to CHAR or NCHAR (for example, `*expression* LIKE CAST(*pattern* AS *datatype*)`.

See "String literals" on page 11, and "CAST function [Data type conversion]" on page 148.

Blank padded databases

The semantics of a LIKE pattern does not change if the database is blank-padded since matching *expression* to *pattern* involves a character-by-character comparison in a left-to-right fashion. No additional blank padding is performed on the value of either *expression* or *pattern* during the evaluation. Therefore, the expression `a1` matches the pattern a1, but not the patterns 'a1 ' (a1, with a space after it) or a1_.

See also

- "LIKE, REGEXP, and SIMILAR TO search conditions" on page 39
- "The WHERE clause: Specifying rows" [*SQL Anywhere Server - SQL Usage*]
- "Optimization of LIKE predicates" [*SQL Anywhere Server - SQL Usage*]
- "REGEXP search condition" on page 45
- "SIMILAR TO search condition" on page 46

Standards and compatibility

- The LIKE search condition is a core feature of the ANSI SQL/2003 standard.
- SQL Anywhere supports ANSI SQL/2003 feature F281, which permits the pattern and escape-expressions to be arbitrary expressions evaluated at execution time. Feature F281 also permits *expression* to be an expression more complex than a simple column reference.
- The use of character ranges and sets contained in square brackets [] is a vendor extension.
- SQL Anywhere supports ANSI SQL/2003 feature T042, which permits LIKE search conditions to reference string-expressions that are LONG VARCHAR values.
- LIKE search conditions that specify NCHAR string expressions or patterns is feature F421 of the ANSI SQL/2003 standard.

REGEXP search condition

Match a pattern against a string.

Syntax

expression [**NOT**] **REGEXP** *pattern* [**ESCAPE** *escape-expression*]

Parameters

expression The string to be searched.

pattern The regular expression to search for within *expression*.

For more information about the syntax for regular expressions, see "Regular expressions overview" on page 20.

escape-expression The escape character to be used in the match. The default is the backslash character (\).

Remarks

The REGEXP search condition matches a whole string, not a substring. To match on a substring with the string, enclose the string in wildcards that match the rest of the string (`.*`*`pattern`*`.*`). For example, `SELECT ... WHERE Description REGEXP 'car'` matches only car, not sportscar. However, `SELECT ... WHERE Description REGEXP '.*car'` matches car, sportscar, and any string that ends with car. Alternatively, you can rewrite your query to make use the REGEXP_SUBSTR function, which is designed to search for substrings within a string.

When matching against only a sub-character class, you must include the outer square brackets and the square brackets for the sub-character class. For example, *`expression`* `REGEXP '[[:digit:]]'`). For more on sub-character class matching, see "Regular expressions: Special sub-character classes" on page 24.

Database collation and matching

REGEXP only matches a literal in a pattern if it is the exact same character (that is, they have the same code point value). Ranges in character classes (for example, `'[A-F]'`) only match characters that code point values greater than or equal to the code point value of the first character in the range (A) and less than or equal to the code point value of the second character in the range (F).

For a comparison of how matching and range evaluations are handled for LIKE, SIMILAR TO, and REGEXP, see "LIKE, REGEXP, and SIMILAR TO search conditions" on page 39.

Comparisons are performed character-by-character, unlike the equivalence (=) operator and other operators where the comparison is done string-by-string. For example, when a comparison is done in a UCA collation (CHAR or NCHAR with the collation set to UCA), `'Æ'='AE'` is true, but `'Æ' REGEXP 'AE'` is false.

National character (NCHAR) support

REGEXP search conditions can be used to compare CHAR and NCHAR strings. In this case, character set conversion is performed so that the comparison is done using a common data type. Then, a code point by code point comparison is performed. See "Comparisons between CHAR and NCHAR" on page 112.

You can specify *expression* or *pattern* as an NCHAR string literal by prefixing the quoted value with N (for example, `expression REGEXP N'pattern'`). You can also use the CAST function to cast the pattern to CHAR or NCHAR (for example, `expression REGEXP CAST(pattern AS datatype)`.

See "String literals" on page 11, and "CAST function [Data type conversion]" on page 148.

See also

- "Regular expressions overview" on page 20
- "SIMILAR TO search condition" on page 46
- "LIKE search condition" on page 41
- "REGEXP_SUBSTR function [String]" on page 274
- "LIKE, REGEXP, and SIMILAR TO search conditions" on page 39

SIMILAR TO search condition

Match a pattern against a string.

Syntax

expression [**NOT**] **SIMILAR TO** *pattern* [**ESCAPE** *escape-expression*]

Parameters

expression The expression to be searched.

pattern The regular expression to search for within *expression*.

For more information about the supported syntax for regular expressions, see "Regular expressions overview" on page 20.

escape-expression The escape character to use in the match. The default escape character is the null character, which can be specified in a string literal as '\x00'.

Regular expression syntax	Meaning
\x	Match anything that compares equal to *x*, where the escape character is assumed to be the backslash character (\\). For example, `\[` matches '['.
x	Any character (other than a meta-character) matches itself. For example, A matches 'A'.

Remarks

To match a substring with the string, use the percentage sign wildcard (%*expression*). For example, `SELECT ... WHERE Description SIMILAR TO 'car'` matches only car, not sportscar. However, `SELECT ... WHERE Description SIMILAR TO '%car'` matches car, sportscar, and any string that ends with car.

When matching against only a sub-character class, you must include the outer square brackets, and the square brackets for the sub-character class. For example, *expression* SIMILAR TO '[[:digit:]]'). For more on sub-character class matching, see "Regular expressions: Special sub-character classes" on page 24.

Comparisons are performed character-by-character, unlike the equivalence (=) operator and other operators where the comparison is done string-by-string. For example, when a comparison is done in a UCA collation (CHAR or NCHAR with the collation set to UCA), `'Æ'='AE'` is true, but `'Æ' SIMILAR TO 'AE'` is false.

For a character-by-character comparison to match, each single character in the expression being searched must match a single character or a wildcard in the SIMILAR TO pattern.

Database collation and matching

SIMILAR TO use the collation to determine character equivalence and evaluate character class ranges. For example, if the database is case- and accent-insensitive, matches are case- and accent-insensitive. Ranges are also evaluated using the collation sort order.

For a comparison of how matching and range evaluations are handled for LIKE, SIMILAR TO, and REGEXP, see "LIKE, REGEXP, and SIMILAR TO search conditions" on page 39.

National character (NCHAR) support

SIMILAR TO search conditions can be used to compare CHAR and NCHAR strings. In this case, character set conversion is performed so that the comparison is done using a common data type. Then, a character-by-character comparison is performed. See "Comparisons between CHAR and NCHAR" on page 112.

You can specify *expression* or *pattern* as an NCHAR string literal by prefixing the quoted value with N (for example, `expression SIMILAR TO N'pattern'`). You can also use the CAST function to cast the pattern to CHAR or NCHAR (for example, `expression SIMILAR TO CAST(pattern AS datatype)`.

See "String literals" on page 11, and "CAST function [Data type conversion]" on page 148.

See also

- "Regular expressions overview" on page 20
- "REGEXP search condition" on page 45
- "LIKE search condition" on page 41
- "REGEXP_SUBSTR function [String]" on page 274
- "LIKE, REGEXP, and SIMILAR TO search conditions" on page 39

IN search condition

Syntax

expression [**NOT**] **IN** { **(** *subquery* **)** | **(** *expression2* **)** | **(** *value-expression1*, ... **)** }

Remarks

An IN search condition, without the NOT keyword, evaluates according to the following rules:

- TRUE if *expression* is not NULL and equals at least one of the values.
- UNKNOWN if *expression* is NULL and the values list is not empty, or if at least one of the values is NULL and *expression* does not equal any of the other values.
- FALSE if *expression* is NULL and *subquery* returns no values; or if *expression* is not NULL, none of the values are NULL, and *expression* does not equal any of the values.

The NOT keyword interchanges TRUE and FALSE.

The search condition *expression* **IN (** *values* **)** is equivalent to *expression* = **ANY (** *values* **)**.

The search condition *expression* **NOT IN (** *values* **)** is equivalent to *expression* <> **ALL (** *values* **)**.

The *value-expression* arguments are expressions that take on a single value, which may be a string, a number, a date, or any other SQL data type.

CONTAINS search condition

Syntax

CONTAINS (*column-name* [,...]**,** *contains-query-string* **)**

contains-query-string :
simple-expression
| *or-expression*

simple-expression :
primary-expression
| *and-expression*

or-expression :
simple-expression { **OR** | **|** } *contains-query-string*

primary-expression :
basic-expression
| **FUZZY "** *fuzzy-expression* **"**
| *and-not-expression*

and-expression :
primary-expression [**AND** | **&**] *simple-expression*

and-not-expression :
primary-expression [**AND** | **&**] { **NOT** | **-** } *basic-expression*

basic-expression :
term
| *phrase*
| **(** *contains-query-string* **)**
| *near-expression*

fuzzy-expression :
term
| *fuzzy-expression term*

term :
simple-term
| *prefix-term*

prefix-term :
*simple-term******

phrase :
" *phrase-string* **"**

near-expression :
term **NEAR[***distance***]** *term*
| *term* { **NEAR** | **~** } *term*

phrase-string :
term
| *phrase-string term*

simple-term : A string separated by whitespace and special characters that represents a single indexed term (word) to search for.

distance : a positive integer

Parameters

- **and-expression** Use *and-expression* to specify that both *primary-expression* and *simple-expression* must be found in the text index.

  By default, if no operator is specified between terms or expressions, an *and-expression* is assumed. For example, `'a b'` is interpreted as `'a AND b'`.

  An ampersand (&) can be used instead of AND, and can abut the expressions or terms on either side (for example, `'a &b'`).

  See "Allowed syntax for special characters" on page 52.

- **and-not-expression** Use *and-not-expression* to specify that *primary-expression* must be present in the text index, but that *basic-expression must not* be found in the text index. This is also known as a **negation**.

  If you use a hyphen for negation, the hyphen must have a space to the left of it, and must abut the term to the right; otherwise, the hyphen is not interpreted as a negation. For example, `'a -b'` is equivalent to `'a AND NOT b'`; whereas for `'a - b'`, the hyphen is ignored and the string is equivalent to `'a AND b'`. `'a-b'` is equivalent to the phrase `'"a b"'`. See "Allowed syntax for hyphen (-)" on page 52.

- **or-expression** Use *or-expression* to specify that at least one of *simple-expression* or *contains-query-string* must be present in the text index. For example, `'a|b'` is interpreted as `'a OR b'`. See "Allowed syntax for special characters" on page 52.

- **fuzzy-expression** Use *fuzzy-expression* to find terms that are similar to what you specify. Fuzzy matching is only supported on NGRAM text indexes. See "Fuzzy searches" [*SQL Anywhere Server - SQL Usage*].

- **near-expression** Use *near-expression* to search for terms that are near each other. This is also known as a **proximity search**. For example, `'b NEAR[5] c'` searches for instances of b and c that are five or less terms away from each other. The order of terms is not significant; `'b NEAR c'` is equivalent to `'c NEAR b'`.

  If NEAR is specified without *distance*, a default of 10 terms is applied.

  You can specify a tilde (~) instead of NEAR. This is equivalent to specifying NEAR without a distance so a default of 10 terms is applied.

  NEAR expressions cannot be chained together (for example, `'a NEAR[1] b NEAR[1] c'`).

  See "Allowed syntax for special characters" on page 52, and "Proximity searching" [*SQL Anywhere Server - SQL Usage*].

- **prefix-term** Use *prefix-term* to search for terms that start with the specified prefix. For example, `'datab*'` searches for any term beginning with datab. This is also known as a **prefix search**. In a prefix search, matching is performed for the portion of the term to the left of the asterisk. See "Allowed syntax for asterisk (*)" on page 51, and "Prefix searching" [*SQL Anywhere Server - SQL Usage*].

Remarks

The CONTAINS search condition takes a column list and *contains-query-string* as arguments. It can be used anywhere a search condition (also referred to as predicate) can be specified, and returns TRUE or FALSE. *contains-query-string* must be a constant string, or a variable, with a value that is known at query time.

If multiple columns are specified, then they must all refer to a single base table; a text index cannot span multiple base tables. The base table can be referenced directly in the FROM clause, or it can used in a view or derived table provided that the view or derived table does not use DISTINCT, GROUP BY, ORDER BY, UNION, INTERSECT, EXCEPT, or a row limitation.

The following warnings apply to the use of non-alphanumeric characters in query strings:

- An asterisk in the middle of a term returns an error.
- You should not use non-alphanumerics (including special characters) in *fuzzy-expression* because they are treated as whitespace and serve as term breakers.
- If possible, do not include non-alphanumeric characters that are not special characters in your query string. Any non-alphanumeric character that is not a special character causes the term containing it to be treated as a phrase, breaking the term at the location of the character. For example, `'things we've done'` is interpreted as `'things "we ve" done'`.

Within phrases, the asterisk is the only special character that continues to be interpreted as a special character. All other special characters within phrases are treated as whitespace and serve as term breakers.

Interpretation of *contains-query-string* takes place in two main steps:

- **Step 1: Interpreting operators and precedence** During this step, keywords are interpreted as operators, and rules of precedence are applied. See "Operator precedence in a CONTAINS search condition" on page 51.

- **Step 2: Applying text configuration object settings** During this step, the text configuration object settings are applied to terms. For example, on an NGRAM text index, terms are broken down into their n-gram representation. During this step, the query terms that exceed the term length settings, or that are in the stoplist, are dropped. For more information about how a query string is interpreted when terms are dropped, see "Example text configuration objects" [*SQL Anywhere Server - SQL Usage*].

Operator precedence in a CONTAINS search condition

During query evaluation, expressions are evaluated using the following order of precedence:

1. FUZZY, NEAR
2. AND NOT
3. AND
4. OR

Allowed syntax for asterisk (*)

The asterisk is used for **prefix searching**. An asterisk can occur at the end of the query string, or be followed by a space, ampersand, vertical bar, closing bracket, or closing quotation mark. Any other usage of asterisk returns an error.

The following table shows allowable asterisk usage:

Query string	Equivalent to:	Interpreted as:
`'th*'`		Find any term beginning with th.
`'th*&best'`	`'th* AND best'` and `'th* best'`	Find any term beginning with th, and the term best.
`'th*|best'`	`'th* OR best'`	Find either any term beginning with th, or the term best
`'very&(best|th*)'`	`'very AND (best OR th*)'`	Find the term very, and the term best or any term beginning with th.
`'"fast auto*"'`		Find the term fast, immediately followed by a term beginning with auto.
`'"auto* price"'`		Find a term beginning with auto, immediately followed by the term price.

Note
Interpretation of query strings containing asterisks can vary depending on the text configuration object settings. See "Prefix searching" [*SQL Anywhere Server - SQL Usage*].

Allowed syntax for hyphen (-)

The hyphen can be used for term or expression **negation**, and is equivalent to NOT. Whether a hyphen is interpreted as a negation depends on its location in the query string. For example, when a hyphen immediately precedes a term or expression it is interpreted as a negation. If the hyphen is embedded within a term, it is interpreted as a hyphen.

A hyphen used for negation must be preceded by a whitespace, and followed immediately by an expression.

When used in a phrase of a fuzzy expression, the hyphen is treated as whitespace and used as a term breaker.

The following table shows the allowed syntax for hyphen:

Query string	**Equivalent to:**	**Interpreted as:**
`'the -best'`	`'the AND NOT best'`, `'the AND -best'`, `'the & -best'`, `'the NOT best'`	Find the term the, and not the term best.
`'the -(very best)'`	`'the AND NOT (very AND best)'`	Find the term the, and not the terms very and best.
`'the -"very best"'`	`'the AND NOT "very best"'`	Find the term the, and not the phrase very best.
`'alpha-numerics'`	`'"alpha numerics"'`	Find the term alpha, immediately followed by the term numerics.
`'wild - west'`	`'wild west'`, and `'wild AND west'`	Find the term wild, and the term west.

Allowed syntax for special characters

The following table shows the allowed syntax for all special characters except asterisk and hyphen, which are covered in the previous sections: “Allowed syntax for asterisk (*)” on page 51, and “Allowed syntax for hyphen (-)” on page 52.

These characters are not considered special characters if they are found in a phrase, and are dropped.

Note
The same restrictions with regards to specifying string literals also apply to the query string. For example, apostrophes must be escaped, and so on. For more information on formatting string literals, see “String literals” on page 11.

Character or syntax	Usage Examples and remarks
ampersand (&)	The ampersand is equivalent to AND, and can be specified as follows: ● `'a & b'` ● `'a &b'` ● `'a&b'` ● `'a& b'`
vertical bar (|)	The vertical bar is equivalent to OR, and can be specified as follows: ● `'a|b'` ● `'a |b'` ● `'a | b'` ● `'a| b'`
double-quotes (")	Double-quotes are used to contain a sequence of terms where order and relative distance are important. For example, in the query string `'learn "full text search"'`, "full text search" is a phrase. In this example, learn can come before or after the phrase, or exist in another column (if the text index is built on more than one column), but the exact phrase must be found in a single column.
parentheses ()	Parentheses are used to specify the order of evaluation of expressions if different from the default order. For example `'a AND (b|c)'` is interpreted as a, and b or c. For more information about the default order of evaluation, see "Operator precedence in a CONTAINS search condition" on page 51.
tilde (~)	The tilde is equivalent to NEAR, and has no special syntax rules. The query string `'full~text'` is equivalent to `'full NEAR text'`, and is interpreted as: the term full within ten terms of the term text.
square brackets [ ]	Square brackets are used in conjunction with the keyword NEAR to contain *distance*. Other uses of square brackets returns an error.

See also

- "Full text searching" [*SQL Anywhere Server - SQL Usage*]
- "Text configuration object settings" [*SQL Anywhere Server - SQL Usage*]
- "Example text configuration objects" [*SQL Anywhere Server - SQL Usage*]
- "FROM clause" on page 613
- "sa_char_terms system procedure" on page 843
- "sa_nchar_terms system procedure" on page 914

EXISTS search condition

Syntax

EXISTS (*subquery* **)**

Remarks

The EXISTS search condition is TRUE if the subquery result contains at least one row, and FALSE if the subquery result does not contain any rows. The EXISTS search condition cannot be UNKNOWN.

IS NULL and IS NOT NULL search conditions

Syntax

expression **IS** [**NOT**] **NULL**

Remarks

Without the NOT keyword, the IS NULL search condition is TRUE if the expression is the NULL value, and FALSE otherwise. The NOT keyword reverses the meaning of the search condition.

Truth value search conditions

Syntax

IS [**NOT**] *truth-value*

Remarks

Without the NOT keyword, the search condition is TRUE if the *condition* evaluates to the supplied *truth-value*, which must be one of TRUE, FALSE, or UNKNOWN. Otherwise, the value is FALSE. The NOT keyword reverses the meaning of the search condition, but leaves UNKNOWN unchanged.

Standards and compatibility

- Vendor extension.

Trigger operation conditions

Syntax

trigger-operation:
INSERTING
| **DELETING**
| **UPDATING** [**(** *column-name-string* **)**]
| **UPDATE (** *column-name* **)**

Remarks

Trigger-operation conditions can be used only in triggers, to carry out actions depending on the kind of action that caused the trigger to fire.

The argument for UPDATING is a quoted string (for example, `UPDATING( 'mycolumn' )`). The argument for UPDATE is an identifier (for example, `UPDATE( mycolumn )`). The two versions are interoperable, and are included for compatibility with SQL dialects of other vendors' DBMS.

If you supply an UPDATING or UPDATE function, you must also supply a REFERENCING clause in the CREATE TRIGGER statement to avoid syntax errors.

Example

The following trigger displays a message showing which action caused the trigger to fire.

```
CREATE TRIGGER tr BEFORE INSERT, UPDATE, DELETE
ON sample_table
REFERENCING OLD AS t1old
FOR EACH ROW
BEGIN
    DECLARE msg varchar(255);
    SET msg = 'This trigger was fired by an ';
    IF INSERTING THEN
        SET msg = msg || 'insert'
    ELSEIF DELETING THEN
        set msg = msg || 'delete'
    ELSEIF UPDATING THEN
        set msg = msg || 'update'
    END IF;
    MESSAGE msg TO CLIENT
END;
```

See also

- "BEGIN statement" on page 413
- "Using procedures, triggers, and batches" [*SQL Anywhere Server - SQL Usage*]

Three-valued logic

The following tables display how the AND, OR, NOT, and IS logical operators of SQL work in three-valued logic.

AND operator

AND	TRUE	FALSE	UNKNOWN
TRUE	TRUE	FALSE	UNKNOWN
FALSE	FALSE	FALSE	FALSE
UNKNOWN	UNKNOWN	FALSE	UNKNOWN

OR operator

OR	TRUE	FALSE	UNKNOWN
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	UNKNOWN
UNKNOWN	TRUE	UNKNOWN	UNKNOWN

NOT operator

TRUE	FALSE	UNKNOWN
FALSE	TRUE	UNKNOWN

IS operator

IS	TRUE	FALSE	UNKNOWN
TRUE	TRUE	FALSE	FALSE
FALSE	FALSE	TRUE	FALSE
UNKNOWN	FALSE	FALSE	TRUE

Explicit selectivity estimates

SQL Anywhere uses statistical information to determine the most efficient strategy for executing each statement. SQL Anywhere automatically gathers and updates these statistics. These statistics are stored permanently in the database in the system table ISYSCOLSTAT. Statistics gathered while processing one statement are available when searching for efficient ways to execute subsequent statements.

Occasionally, the statistics may become inaccurate or relevant statistics may be unavailable. This condition is most likely to arise when few queries have been executed since a large amount of data was added, updated, or deleted. In this situation, you may want to execute a CREATE STATISTICS statement. See "CREATE STATISTICS statement" on page 516.

If there are problems with a particular execution plan, you can use optimizer hints to require that a particular index be used. For more information, see "FROM clause" on page 613.

In unusual circumstances, however, these measures may prove ineffective. In such cases, you can sometimes improve performance by supplying explicit selectivity estimates.

For each table in a potential execution plan, the optimizer must estimate the number of rows that will be part of the result set. If you know that a condition has a success rate that differs from the optimizer's estimate, you can explicitly supply a user estimate in the search condition.

The estimate is a percentage. It can be a positive integer or can contain fractional values.

Caution
Whenever possible, avoid supplying explicit estimates in statements that are to be used on an ongoing basis. Should the data change, the explicit estimate may become inaccurate and may force the optimizer to select poor plans. If you do use explicit selectivity estimates, ensure that the number is accurate. Do not, for example, supply values of 0% or 100% to force the use of an index.

You can disable user estimates by setting the database option user_estimates to Off. The default value for user_estimates is Override-Magic, which means that user-supplied selectivity estimates are used only when the optimizer would use a MAGIC (default) selectivity value for the condition. The optimizer uses MAGIC values as a last resort when it is unable to accurately predict the selectivity of a predicate.

For more information about disabling user-defined selectivity estimates, see "user_estimates option [database]" [*SQL Anywhere Server - Database Administration*].

For more information about statistics, see "Optimizer estimates and column statistics" [*SQL Anywhere Server - SQL Usage*].

Examples

The following query provides an estimate that one percent of the ShipDate values are later than 2001/06/30:

```
SELECT  ShipDate
   FROM  SalesOrderItems
WHERE ( ShipDate > '2001/06/30', 1 )
ORDER BY ShipDate DESC;
```

The following query estimates that half a percent of the rows satisfy the condition:

```
SELECT *
   FROM Customers c, SalesOrders o
WHERE (c.ID = o.CustomerID, 0.5);
```

Fractional values enable more accurate user estimates for joins and large tables.

Special values

Special values can be used in expressions, and as column defaults when creating tables.

While some special values can be queried, some can only be used as default values for columns. For example, **USER**, **LAST USER**, **TIMESTAMP** and **UTC TIMESTAMP** can only be used as default values.

CURRENT DATABASE special value

CURRENT DATABASE returns the name of the current database.

Data type

STRING

See also

- “Expressions” on page 16

CURRENT DATE special value

CURRENT DATE returns the current year, month, and day.

Data type

DATE

See also

- “Expressions” on page 16
- “TIME data type” on page 103

CURRENT PUBLISHER special value

CURRENT PUBLISHER returns a string that contains the publisher user ID of the database for SQL Remote replications.

Data type

STRING

Remarks

CURRENT PUBLISHER can be used as a default value in columns with character data types.

See also

- “Expressions” on page 16
- “SQL Remote replication design and set up” [*SQL Remote*]

CURRENT TIME special value

The current hour, minute, second and fraction of a second.

Data type

TIME

Remarks

The fraction of a second is stored to 6 decimal places. The accuracy of the current time is limited by the accuracy of the system clock.

See also

- "Expressions" on page 16
- "TIME data type" on page 103

CURRENT TIMESTAMP special value

CURRENT TIMESTAMP combines CURRENT DATE and CURRENT TIME to form a TIMESTAMP value containing the year, month, day, hour, minute, second and fraction of a second. The fraction of a second is stored to 3 decimal places. The accuracy is limited by the accuracy of the system clock.

Unlike DEFAULT TIMESTAMP, columns declared with DEFAULT CURRENT TIMESTAMP do not necessarily contain unique values. If uniqueness is required, consider using DEFAULT TIMESTAMP instead.

The information CURRENT TIMESTAMP returns is equivalent to the information returned by the GETDATE and NOW functions.

CURRENT_TIMESTAMP is equivalent to CURRENT TIMESTAMP.

Note
The main difference between DEFAULT CURRENT TIMESTAMP and DEFAULT TIMESTAMP is that DEFAULT CURRENT TIMESTAMP is set only at INSERT, while DEFAULT TIMESTAMP is set at both INSERT and UPDATE.

Data type

TIMESTAMP

See also

- "CURRENT TIME special value" on page 59
- "TIMESTAMP special value" on page 63
- "Expressions" on page 16
- "TIMESTAMP data type" on page 104
- "GETDATE function [Date and time]" on page 210
- "NOW function [Date and time]" on page 258

CURRENT USER special value

CURRENT USER returns a string that contains the user ID of the current connection.

Data type

STRING

Remarks

CURRENT USER can be used as a default value in columns with character data types.

On UPDATE, columns with a default value of CURRENT USER are not changed. CURRENT_USER is equivalent to CURRENT USER.

See also

- "Expressions" on page 16

CURRENT UTC TIMESTAMP special value

CURRENT UTC TIMESTAMP combines CURRENT DATE and CURRENT TIME, adjusted by the server's time zone adjustment value, to form a Coordinated Universal Time (UTC) TIMESTAMP value containing the year, month, day, hour, minute, second and fraction of a second. This feature allows data to be entered with a consistent time reference, regardless of the time zone in which the data was entered.

Data type

TIMESTAMP

See also

- "TIMESTAMP data type" on page 104
- "UTC TIMESTAMP special value" on page 65
- "CURRENT TIMESTAMP special value" on page 59
- "truncate_timestamp_values option [database] [MobiLink client]" [*SQL Anywhere Server - Database Administration*]

LAST USER special value

LAST USER is the name of the user who last modified the row.

Data type

String

Remarks

LAST USER can be used as a default value in columns with character data types.

On INSERT, this constant has the same effect as CURRENT USER. On UPDATE, if a column with a default value of LAST USER is not explicitly modified, it is changed to the name of the current user.

When combined with the DEFAULT TIMESTAMP, a default value of LAST USER can be used to record (in separate columns) both the user and the date and time a row was last changed.

See also

- "CURRENT USER special value" on page 60
- "CURRENT TIMESTAMP special value" on page 59
- "CREATE TABLE statement" on page 523

SQLCODE special value

SQLCODE indicates the disposition of the most recently executed SQL statement.

Data type

Signed INTEGER

Remarks

The database server sets a SQLSTATE and SQLCODE for each SQL statement it executes. SQLCODEs are product-specific (for example, MobiLink has its own SQLCODEs), and can be used to learn additional information about the SQLSTATE. For example, positive values other than 100 indicate product-specific *warning* conditions. Negative values indicate product-specific *exception* conditions. The value 100 indicates "no data" (for example, at the end of a result set fetched via a cursor).

SQLSTATE and SQLCODE are related in that each SQLCODE corresponds to a SQLSTATE, and each SQLSTATE can correspond to one or more SQLCODEs.

To return the error condition associated with a SQLCODE, you can use the ERRORMSG function. See "ERRORMSG function [Miscellaneous]" on page 194.

Note
SQLSTATE is the preferred status indicator for the outcome of a SQL statement. See "SQLSTATE special value" on page 62.

See also

- "SQLSTATE special value" on page 62
- "SQL Anywhere error messages sorted by SQLCODE" [*Error Messages*]
- "Expressions" on page 16

Standards and compatibility

SQLCODE was deprecated in the ANSI SQL/1992 standard, and was eliminated entirely from SQL/1999. SQLCODE values continue to be maintained in SQL Anywhere for backward compatibility for applications. SQLSTATE is the preferred status indicator.

SQLSTATE special value

SQLSTATE indicates whether the most recently executed SQL statement resulted in a success, error, or warning condition.

Data type

String

Remarks

The database server sets a SQLSTATE and SQLCODE for each SQL statement it executes. A SQLSTATE is a string that indicates the whether the most recently executed SQL statement resulted in a success, warning, or error condition.

Each SQLSTATE represents errors that are common to all platforms, and usually contain non-product-specific wording. The format of a SQLSTATE value is a two-character class value, followed by a three-character subclass value. Guidelines for SQLSTATE conformance with regard to class and subclass values are outlined in the ISO/ANSI SQL standard.

SQL Anywhere conforms to the ISO/ANSI SQLSTATE conventions with the following additions and exceptions:

Class and subclass	**Condition**
01WC*x*	Warnings related to character set conversion
38*xxx*	External function exception
42X*xx*	Syntax error: expressions
42R*xx*	Syntax error: referential integrity (for example, attempt to create 2nd primary key)
42W*xx*	Syntax error: generic
42U*xx*	Syntax error: duplicate, undefined, or ambiguous object reference
42Z*xx*	Access violation
54W*xx*	Product limit exceeded
55W*xx*	Object not in required state for operation to succeed
57*xxx*	Resource not available or operator intervention
5R*xxx*	SQL Remote errors
WB*xxx*	Online backup errors
WI*xxx*	Internal database errors

Class and subclass	Condition
WP*xxx*	Errors in procedures, variables, and so on
WL*xxx*	Errors loading and/or unloading
WW*xxx*	Miscellaneous SQL Anywhere-specific errors/warnings (including system failures)
WO*xxx*	Remote data access feature-related errors
WJ*xxx*	JCS and JDBC related errors
WC*xxx*	Character translation errors
WX*xxx*	XML-related errors
WT*xxx*	Text-related errors

The successful completion class is '00*xxx*' (for example, '00000').

SQLSTATE and SQLCODE are related in that each SQLCODE corresponds to a SQLSTATE, and each SQLSTATE can correspond to one or more SQLCODEs.

To return the error condition associated with a SQLSTATE, you can use the ERRORMSG function. See "ERRORMSG function [Miscellaneous]" on page 194.

To see the SQLSTATE values used by SQL Anywhere, see "SQL Anywhere error messages sorted by SQLSTATE" [*Error Messages*].

See also

- "SQLCODE special value" on page 61
- "Expressions" on page 16

Standards and compatibility

- **SQL/2003: core feature** SQLSTATE classes (the first two characters) beginning with the values '0'-'4', and 'A'-'H' are defined by the ANSI standard. Other classes are implementation-defined. Similarly, subclass values that begin with values '0'-'4', and 'A'-'H' are defined by the ANSI standard. Subclass values outside these ranges are implementation-defined.

TIMESTAMP special value

TIMESTAMP indicates when each row in the table was last modified. When a column is declared with DEFAULT TIMESTAMP, a default value is provided for inserts, and the value is updated with the current date and time whenever the row is updated.

Data type

TIMESTAMP

Remarks

Columns declared with DEFAULT TIMESTAMP contain unique values so that applications can detect near-simultaneous updates to the same row. If the current timestamp value is the same as the last value, it is incremented by the value of the default_timestamp_increment option.

You can automatically truncate timestamp values in SQL Anywhere based on the default_timestamp_increment option. This is useful for maintaining compatibility with other database software that records less precise timestamp values.

The global variable @@dbts returns a TIMESTAMP value representing the last value generated for a column using DEFAULT TIMESTAMP.

> **Note**
> The main difference between DEFAULT TIMESTAMP and DEFAULT CURRENT TIMESTAMP is that DEFAULT CURRENT TIMESTAMP is set only at INSERT, while DEFAULT TIMESTAMP is set at both INSERT and UPDATE.

See also

- "TIMESTAMP data type" on page 104
- "CURRENT TIMESTAMP special value" on page 59
- "CURRENT UTC TIMESTAMP special value" on page 60
- "default_timestamp_increment option [database] [MobiLink client]" [*SQL Anywhere Server - Database Administration*]
- "truncate_timestamp_values option [database] [MobiLink client]" [*SQL Anywhere Server - Database Administration*]

USER special value

USER returns a string that contains the user ID of the current connection.

Data type

STRING

Remarks

USER can be used as a default value in columns with character data types.

On UPDATE, columns with a default value of USER are not changed.

See also

- "Expressions" on page 16
- "CURRENT USER special value" on page 60
- "LAST USER special value" on page 60

UTC TIMESTAMP special value

UTC TIMESTAMP indicates the Coordinated Universal (UTC) time when each row in the table was last modified.

When a column is declared with DEFAULT UTC TIMESTAMP, a default value is provided for inserts, and the value is updated with the current UTC date and time whenever the row is updated.

Data type

TIMESTAMP

Remarks

Columns declared with DEFAULT UTC TIMESTAMP contain unique values so that applications can detect near-simultaneous updates to the same row. If the current UTC timestamp value is the same as the last value, it is incremented by the value of the default_timestamp_increment option.

You can automatically truncate UTC timestamp values in SQL Anywhere with the default_timestamp_increment option. This is useful for maintaining compatibility with other database software that records less precise timestamp values.

Note
The main difference between DEFAULT UTC TIMESTAMP and DEFAULT CURRENT UTC TIMESTAMP is that DEFAULT CURRENT UTC TIMESTAMP is set only at INSERT, while DEFAULT UTC TIMESTAMP is set at both INSERT and UPDATE.

See also

- “TIMESTAMP data type” on page 104
- “CURRENT UTC TIMESTAMP special value” on page 60
- “TIMESTAMP special value” on page 63
- “default_timestamp_increment option [database] [MobiLink client]” [*SQL Anywhere Server - Database Administration*]
- “truncate_timestamp_values option [database] [MobiLink client]” [*SQL Anywhere Server - Database Administration*]

Variables

SQL Anywhere supports three levels of variables:

- **Local variables** These are defined inside a compound statement in a procedure or batch using the DECLARE statement. They exist only inside the compound statement.
- **Connection-level variables** These are defined with a CREATE VARIABLE statement. They belong to the current connection, and disappear when you disconnect from the database or when you use the DROP VARIABLE statement.
- **Global variables** These are system-supplied variables that have system-supplied values. All global variables have names beginning with two @ signs. For example, the global variable **@@version** has a value that is the current version number of the database server. Users cannot define global variables.

Local and connection-level variables are declared by the user, and can be used in procedures or in batches of SQL statements to hold information. Global variables are system-supplied variables that provide system-supplied values.

See also

- “TIMESTAMP data type” on page 104
- “CREATE VARIABLE statement” on page 545

Local variables

SQL Anywhere supports local variables. Local variables are declared using the DECLARE statement, which can be used only within a compound statement (that is, bracketed by the BEGIN and END keywords). Only one variable can be declared for each DECLARE statement in SQL Anywhere.

If the DECLARE is executed within a compound statement, the scope is limited to the compound statement.

The variable is initially set as NULL. The value of the variable can be set using the SET statement, or can be assigned using a SELECT statement with an INTO clause.

The syntax of the DECLARE statement is as follows:

```
DECLARE variable-name data-type
```

Local variables can be passed as arguments to procedures, as long as the procedure is called from within the compound statement.

Examples

The following batch illustrates the use of local variables.

```
BEGIN
   DECLARE local_var INT;
   SET local_var = 10;
   MESSAGE 'local_var = ', local_var TO CLIENT;
END
```

Running this batch from Interactive SQL gives the message `local_var = 10` in the Interactive SQL Messages tab.

The variable local_var does not exist outside the compound statement in which it is declared. The following batch is invalid, and gives a `column not found` error.

```
-- This batch is invalid.
BEGIN
   DECLARE local_var INT;
   SET local_var = 10;
END;
MESSAGE 'local_var = ', local_var TO CLIENT;
```

The following example illustrates the use of SELECT with an INTO clause to set the value of a local variable:

```
BEGIN
   DECLARE local_var INT;
   SELECT 10 INTO local_var;
   MESSAGE 'local_var = ', local_var TO CLIENT;
END
```

Running this batch from Interactive SQL gives the message `local_var = 10` on the database server messages window.

For more information about batches and local variable scope, see "Variables in Transact-SQL procedures" [*SQL Anywhere Server - SQL Usage*].

Connection-level variables

Connection-level variables are declared with the CREATE VARIABLE statement. Connection-level variables can be passed as parameters to procedures.

The syntax for the CREATE VARIABLE statement is as follows:

CREATE VARIABLE *variable-name data-type*

When a variable is created, it is initially set to NULL. The value of connection-level variables can be set in the same way as local variables, using the SET statement or using a SELECT statement with an INTO clause.

Connection-level variables exist until the connection is terminated, or until the variable is explicitly dropped using the DROP VARIABLE statement. The following statement drops the variable con_var:

```
DROP VARIABLE con_var;
```

Example

The following batch of SQL statements illustrates the use of connection-level variables.

```
CREATE VARIABLE con_var INT;
SET con_var = 10;
MESSAGE 'con_var = ', con_var TO CLIENT;
```

Running this batch from Interactive SQL gives the message `con_var = 10` on the database server messages window.

Global variables

Global variables have values set by the database server. For example, the global variable @@version has a value that is the current version number of the database server.

Global variables are distinguished from local and connection-level variables by having two @ signs preceding their names. For example, @@error and @@rowcount are global variables. Users cannot create global variables, and cannot update the values of global variables directly.

Some global variables, such as @@identity, hold connection-specific information, and so have connection-specific values. Other variables, such as @@connections, have values that are common to all connections.

Global variable and special constants

The special constants (for example, CURRENT DATE, CURRENT TIME, USER, and SQLSTATE) are similar to global variables.

The following statement retrieves a value of the version global variable.

```
SELECT @@version;
```

In procedures and triggers, global variables can be selected into a variable list. The following procedure returns the server version number in the *ver* parameter.

```
CREATE PROCEDURE VersionProc ( OUT ver VARCHAR(100) )
 BEGIN
   SELECT @@version
   INTO ver;
 END;
```

In Embedded SQL, global variables can be selected into a host variable list.

List of global variables

The following table lists the global variables available in SQL Anywhere. Some global variables are supplied for compatibility with Transact-SQL, and return a fixed value of either 0, 1, or NULL, as noted.

Variable name	**Meaning**
@@char_convert	0 (Provided for compatibility with Transact-SQL.)
@@client_csid	-1 (Provided for compatibility with Transact-SQL.)
@@client_csname	NULL (Provided for compatibility with Transact-SQL.)
@@connections	The number of logins since the server was last started.
@@cpu_busy	0 (Provided for compatibility with Transact-SQL.)
@@dbts	A value of type TIMESTAMP representing the last generated value used for all columns defined with DEFAULT TIMESTAMP.

Variable name	Meaning
@@error	A Transact-SQL error code that checks the success or failure of the most recently executed statement. If the previous transaction succeeded, 0 is returned. If the previous transaction was unsuccessful, the last error number generated by the system is returned. To view descriptions of the values returned by @@error, see "Error handling in Transact-SQL procedures" [*SQL Anywhere Server - SQL Usage*]. A statement such as `if @@error != 0 return` causes an exit if an error occurs. Every statement resets @@error, including PRINT statements or IF tests, so the status check must immediately follow the statement whose success you want verified.
@@fetch_status	Contains status information resulting from the last fetch statement. This feature is the same as @@sqlstatus, except that it returns different values. It is for Microsoft SQL Server compatibility. @@fetch_status may contain the following values: ● 0 The fetch statement completed successfully. ● -1 The fetch statement resulted in an error. ● -2 There is no more data in the result set.
@@identity	Last value inserted into any IDENTITY or DEFAULT AUTOINCREMENT column by an INSERT or SELECT INTO statement. See "@@identity global variable" on page 71.
@@idle	0 (Provided for compatibility with Transact-SQL.)
@@io_busy	0 (Provided for compatibility with Transact-SQL.)
@@isolation	Current isolation level of the connection. @@isolation takes the value of the active level.
@@langid	Unique language ID for the language in use by the current connection.
@@language	Name of the language in use by the connection.
@@max_connections	For the personal server, the maximum number of simultaneous connections that can be made to the server, which is 10. For the network server, the maximum number of active clients (not database connections, as each client can support multiple connections).
@@maxcharlen	Maximum length, in bytes, of a character in the CHAR character set.
@@ncharsize	Maximum length, in bytes, of a character in the NCHAR character set.
@@nestlevel	-1 (Provided for compatibility with Transact-SQL.)

Variable name	Meaning
@@pack_received	0 (Provided for compatibility with Transact-SQL.)
@@pack_sent	0 (Provided for compatibility with Transact-SQL.)
@@packet_errors	0 (Provided for compatibility with Transact-SQL.)
@@procid	Stored procedure ID of the currently executing procedure.
@@rowcount	Number of rows affected by the last statement. The value of @@rowcount should be checked immediately after the statement. Inserts, updates, and deletes set @@rowcount to the number of rows affected. With cursors, @@rowcount represents the cumulative number of rows returned from the cursor result set to the client, up to the last fetch request. The @@rowcount is not reset to zero by any statement which does not affect rows, such as an IF statement.
@@servername	Name of the current database server.
@@spid	The connection handle for the current connection. This is the same value as that displayed by the sa_conn_info procedure.
@@sqlstatus	Contains status information resulting from the last fetch statement. @@sqlstatus may contain the following values: ● 0 The fetch statement completed successfully. ● 1 The fetch statement resulted in an error. ● 2 There is no more data in the result set.
@@textsize	Current value of the SET TEXTSIZE option, which specifies the maximum length, in bytes, of text or image data to be returned with a select statement. The default setting is 32765, which is the largest bytestring that can be returned using READTEXT. The value can be set using the SET statement.
@@thresh_hysteresis	0 (Provided for compatibility with Transact-SQL.)
@@timeticks	0 (Provided for compatibility with Transact-SQL.)
@@total_errors	0 (Provided for compatibility with Transact-SQL.)
@@total_read	0 (Provided for compatibility with Transact-SQL.)
@@total_write	0 (Provided for compatibility with Transact-SQL.)

Variable name	Meaning
@@tranchained	Current transaction mode; 0 for unchained or 1 for chained.
@@trancount	Nesting level of transactions. Each BEGIN TRANSACTION in a batch increments the transaction count.
@@transtate	-1 (Provided for compatibility with Transact-SQL.)
@@version	Version number of the current version of SQL Anywhere.

@@identity global variable

The @@identity variable holds the most recent value inserted into an IDENTITY column or a DEFAULT AUTOINCREMENT column, or zero if the most recent insert was into a table that had no such column.

The value of @@identity is connection specific and is incremented each time a row is inserted into a table. If a statement inserts multiple rows, @@identity reflects the IDENTITY value for the last row inserted. If the affected table does not contain an IDENTITY column, @@ identity is set to 0.

The value of @@identity is not affected by the failure of an INSERT or SELECT INTO statement, or the rollback of the transaction that contained it. @@identity retains the last value inserted into an IDENTITY column, even if the statement that inserted it fails to commit.

@@identity and triggers

When an insert causes referential integrity actions or fires a trigger, @@identity behaves like a stack. For example, if an insert into a table T1 (with an identity or autoincrement column) fires a trigger that inserts a row into table T2 (also with an identity or autoincrement column), then the value returned to the application or procedure which carried out the insert is the value inserted into T1. Within the trigger, @@identity has the T1 value before the insert into T2 and the T2 value after. The trigger can copy the values to local variables if it needs to access both.

Comments

Comments are used to attach explanatory text to SQL statements or statement blocks. The database server does not execute comments.

The following comment indicators are supported in SQL Anywhere:

- **-- (Double hyphen)** The database server ignores any remaining characters on the line. This is the SQL/2003 comment indicator. You can add and remove this comment indicator by pressing Ctrl+minus sign in Interactive SQL and in the Stored Procedure window of Sybase Central. See "Interactive SQL keyboard shortcuts" [*SQL Anywhere Server - Database Administration*].
- **// (Double slash)** The double slash has the same meaning as the double hyphen. You can add and remove this comment indicator by pressing Ctrl+forward slash in Interactive SQL and in the Stored Procedure window of Sybase Central. See "Interactive SQL keyboard shortcuts" [*SQL Anywhere Server - Database Administration*].
- **/* ... */ (Slash-asterisk)** Any characters between the two comment markers are ignored. The two comment markers can be on the same or different lines. Comments indicated in this style can be nested. This style of commenting is also called **C-style comments**.

Examples

The following example illustrates the use of double-hyphen comments:

```
CREATE FUNCTION fullname ( firstname CHAR(30),
         lastname CHAR(30))
RETURNS CHAR(61)
-- fullname concatenates the firstname and lastname
-- arguments with a single space between.
BEGIN
   DECLARE name CHAR(61);
   SET name = firstname || ' ' || lastname;
   RETURN ( name );
END;
```

The following example illustrates the use of C-style comments:

```
/* Lists the names and employee IDs of employees
   who work in the sales department. */
CREATE VIEW SalesEmployees AS
   SELECT EmployeeID, Surname, GivenName
   FROM Employees
   WHERE DepartmentID = 200;
```

NULL value

The NULL value specifies a value that is unknown or not applicable.

Syntax

NULL

Remarks

NULL is a special value that is different from any valid value for any data type. However, the NULL value is a legal value in any data type. NULL is used to represent missing or inapplicable information. There are two separate and distinct cases where NULL is used:

Situation	Description
missing	The field does have a value, but that value is unknown.
inapplicable	The field does not apply for this particular row.

SQL allows columns to be created with the NOT NULL restriction. This means that those particular columns cannot contain NULL.

The NULL value introduces the concept of three valued logic to SQL. The NULL value compared using any comparison operator with any value (including the NULL value) is "UNKNOWN." The only search condition that returns TRUE is the IS NULL predicate. In SQL, rows are selected only if the search condition in the WHERE clause evaluates to TRUE; rows that evaluate to UNKNOWN or FALSE are not selected.

Column space utilization for NULL values is 1 bit per column and space is allocated in multiples of 8 bits. The NULL bit usage is fixed based on the number of columns in the table that allow NULL values.

The IS [NOT] *truth-value* clause, where *truth-value* is one of TRUE, FALSE or UNKNOWN can be used to select rows where the NULL value is involved. For a description of this clause, see "Search conditions" on page 35.

In the following examples, the column Salary contains NULL.

Condition	Truth value	Selected?
Salary = NULL	UNKNOWN	NO
Salary <> NULL	UNKNOWN	NO
NOT (Salary = NULL)	UNKNOWN	NO
NOT (Salary <> NULL)	UNKNOWN	NO
Salary = 1000	UNKNOWN	NO
Salary IS NULL	TRUE	YES

Condition	Truth value	Selected?
Salary IS NOT NULL	FALSE	NO
Salary = *expression* IS UNKNOWN	TRUE	YES

The same rules apply when comparing columns from two different tables. Therefore, joining two tables together will not select rows where any of the columns compared contain the NULL value.

NULL also has an interesting property when used in numeric expressions. The result of any numeric expression involving the NULL value is NULL. This means that if NULL is added to a number, the result is NULL—not a number. If you want NULL to be treated as 0, you must use the **ISNULL(** *expression*, **0)** function.

Many common errors in formulating SQL queries are caused by the behavior of NULL. You will have to be careful to avoid these problem areas. For a description of the effect of three-valued logic when combining search conditions, see "Search conditions" on page 35.

Set operators and DISTINCT clause

In set operations (UNION, INTERSECT, EXCEPT), and in the DISTINCT operation, NULL is treated differently from in search conditions. Rows that contain NULL and are otherwise identical are treated as identical for the purposes of these operations.

For example, if a column called redundant contained NULL for every row in a table T1, then the following statement would return a single row:

```
SELECT DISTINCT redundant FROM T1;
```

Permissions

Must be connected to the database.

Side effects

None.

Standards and compatibility

- **SQL/2003** Core feature.
- **Sybase** In some contexts, Adaptive Server Enterprise treats NULL as a value, whereas SQL Anywhere does not. For example, rows of a column c1 that are NULL are not included in the results of a query with the following WHERE clause in SQL Anywhere, as the condition has a value of UNKNOWN:

  ```
  WHERE NOT( C1 = NULL )
  ```

  In Adaptive Server Enterprise, the condition is evaluated as TRUE, and these rows are returned. You should use IS NULL rather than a comparison operator for compatibility.

  Unique indexes in SQL Anywhere can hold rows that hold NULL and are otherwise identical. Adaptive Server Enterprise does not permit such entries in unique indexes.

If you use jConnect, the tds_empty_string_is_null option controls whether empty strings are returned as NULL strings or as a string containing one blank character.

For more information, see "tds_empty_string_is_null option [database]" [*SQL Anywhere Server - Database Administration*].

See also

- "Expressions" on page 16
- "Search conditions" on page 35

Example

The following INSERT statement inserts a NULL into the date_returned column of the Borrowed_book table.

```
INSERT INTO Borrowed_book ( date_borrowed, date_returned, book )
VALUES ( CURRENT DATE, NULL, '1234' );
```

SQL data types

Contents

Character data types

Character data types are used to store strings of letters, numbers, and other symbols.

SQL Anywhere provides two classes of character data types and some domains defined using those types.

- **CHAR, VARCHAR, LONG VARCHAR** Character data stored in a single- or multibyte character set, often chosen to correspond most closely to the primary language or languages stored in the database.
- **NCHAR, NVARCHAR, LONG NVARCHAR** Character data stored in Unicode's UTF-8 encoding. All Unicode code points can be stored using these types, regardless of the primary language or languages stored in the database.
- **TEXT, UNIQUEIDENTIFIERSTR, XML** Domains based on other character data types.

Storage

All character data values are stored in the same manner. By default, values up to 128 bytes are stored in a single piece. Values longer than 128 bytes are stored with a 4-byte prefix kept locally on the database page and the full value stored in one or more other database pages. These default sizes are controlled by the INLINE and PREFIX clauses of the CREATE TABLE statement.

See also

- "CREATE TABLE statement" on page 523
- "string_rtruncation option [compatibility]" [*SQL Anywhere Server - Database Administration*]

CHAR data type

The CHAR data type stores character data, up to 32767 bytes.

Syntax

CHAR [**(** *max-length* [**CHAR** | **CHARACTER**] **)**]

Parameters

- **max-length** The maximum length of the string. If byte-length semantics are used (CHAR or CHARACTER is not specified as part of the length), then the length is in bytes, and the length must be in the range 1 to 32767. If the length is not specified, then it is 1.

  If character-length semantics are used (CHAR or CHARACTER is specified as part of the length), then the length is in characters, and you must specify *max-length*. When using character-length semantics, the length multiplied by the maximum length of a character in the database encoding must not exceed 32767 bytes. The following table shows the maximum lengths for the supported types of character sets:

Character set	Maximum length of CHAR
Single-byte character set	32767 characters

Character set	Maximum length of CHAR
Double-byte character set	16383 characters
UTF-8	8191 characters

Remarks

Multibyte characters can be stored as CHAR, but the declared length refers to bytes, not characters, unless character-length semantics are used.

CHAR can also be specified as CHARACTER. Regardless of which syntax is used, the data type is described as CHAR.

CHAR is semantically equivalent to VARCHAR, although they are different types. In SQL Anywhere, CHAR is a variable-length type. In other relational database management systems, CHAR is a fixed-length type, and data is padded with blanks to *max-length* bytes of storage. SQL Anywhere does not blank-pad stored character data.

Using character-length semantics may impact what is returned when a client application performs a DESCRIBE on a column, depending on the interface used. For example, when an embedded SQL client performs a DESCRIBE on a column that was declared using byte-length semantics, the length returned is the byte length specified. So, a CHAR(10) column is described as type DT_FIXCHAR with a length of 10 bytes. However, when an embedded SQL client performs a DESCRIBE on a column that was declared using character-length semantics, the length returned is the maximum byte length in the client's CHAR character set. For example, for an embedded SQL client using UTF-8 as the CHAR character set, a CHAR(10 CHAR) column is described as type DT_FIXCHAR with a length of 40 bytes (10 characters multiplied by the maximum of four bytes per character).

See also

- "VARCHAR data type" on page 84
- "LONG VARCHAR data type" on page 80
- "NCHAR data type" on page 80

Standards and compatibility

- **SQL/2003** Compatible with SQL/2003. Character-length semantics is a vendor extension.

LONG NVARCHAR data type

The LONG NVARCHAR data type stores Unicode character data of arbitrary length.

Syntax

LONG NVARCHAR

Remarks

The maximum size is 2 GB.

Characters are stored in UTF-8. Each character requires from one to four bytes. The maximum number of characters that can be stored in a LONG NVARCHAR is over 500 million and possibly over 2 billion, depending on the lengths of the characters stored.

When an embedded SQL client performs a DESCRIBE on a LONG NVARCHAR column, the data type returned is either DT_LONGVARCHAR or DT_LONGNVARCHAR, depending on whether the db_change_nchar_charset function has been called. See "db_change_nchar_charset function" [*SQL Anywhere Server - Programming*].

For ODBC, a LONG NVARCHAR expression is described as SQL_WLONGVARCHAR.

See also

- "NCHAR data type" on page 80
- "NVARCHAR data type" on page 82
- "LONG VARCHAR data type" on page 80

Standards and compatibility

- **SQL/2003** Vendor extension.

LONG VARCHAR data type

The LONG VARCHAR data type stores character data of arbitrary length.

Syntax

LONG VARCHAR

Remarks

The maximum size is 2 GB.

Multibyte characters can be stored as LONG VARCHAR, but the length is in bytes, not characters.

See also

- "CHAR data type" on page 78
- "VARCHAR data type" on page 84
- "LONG NVARCHAR data type" on page 79

Standards and compatibility

- **SQL/2003** Vendor extension.

NCHAR data type

The NCHAR data type stores Unicode character data, up to 8191 characters.

Syntax

NCHAR [**(** *max-length* **)**]

Parameters

- **max-length** The maximum length of the string, in characters. The length must be in the range 1 to 8191. If the length is not specified, then it is 1.

Remarks

Characters are stored using UTF-8 encoding. The maximum number of bytes of storage required is four multiplied by *max-length*, although the actual storage required is usually much less.

NCHAR can also be specified as NATIONAL CHAR or NATIONAL CHARACTER. Regardless of which syntax is used, the data type is described as NCHAR.

When an embedded SQL client performs a DESCRIBE on an NCHAR column, the data type returned is either DT_FIXCHAR or DT_NFIXCHAR, depending on whether the db_change_nchar_charset function has been called. See "db_change_nchar_charset function" [*SQL Anywhere Server - Programming*].

Also, when an embedded SQL client performs a DESCRIBE on an NCHAR column, the length returned is the maximum byte length in the client's NCHAR character set. For example, for an embedded SQL client using the Western European character set cp1252 as the NCHAR character set, an NCHAR(10) column is described as type DT_NFIXCHAR of length 10 (10 characters multiplied by a maximum one byte per character). For an embedded SQL client using the Japanese character set cp932, the same column is described as type DT_NFIXCHAR of length 20 (10 characters multiplied by a maximum two bytes per character).

NCHAR is semantically equivalent to NVARCHAR, although they are different types. In SQL Anywhere, NCHAR is a variable-length type. In other relational database management systems, NCHAR is a fixed-length type, and data is padded with blanks to *max-length* characters of storage. SQL Anywhere does not blank-pad stored character data.

For ODBC, NCHAR is described as either SQL_WCHAR or SQL_WVARCHAR depending on the odbc_distingish_char_and_varchar option. See "odbc_distinguish_char_and_varchar option [database]" [*SQL Anywhere Server - Database Administration*].

See also

- "CHAR data type" on page 78
- "NVARCHAR data type" on page 82
- "LONG NVARCHAR data type" on page 79

Standards and compatibility

- **SQL/2003** Vendor extension.

NTEXT data type

The NTEXT data type stores Unicode character data of arbitrary length.

Syntax

NTEXT

Remarks

NTEXT is a domain, implemented as a LONG NVARCHAR.

See also

- "LONG NVARCHAR data type" on page 79
- "TEXT data type" on page 83

Standards and compatibility

- **SQL/2003** Vendor extension.

NVARCHAR data type

The NVARCHAR data type stores Unicode character data, up to 8191 characters.

Syntax

NVARCHAR [**(** *max-length* **)**]

Parameters

- **max-length** The maximum length of the string, in characters. The length must be in the range 1 to 8191. If the length is not specified, then it is 1.

Remarks

Characters are stored in UTF-8 encoding. The maximum storage number of bytes required is four multiplied by *max-length*, although the actual storage required is usually much less.

NVARCHAR can also be specified as NCHAR VARYING, NATIONAL CHAR VARYING, or NATIONAL CHARACTER VARYING. Regardless of which syntax is used, the data type is described as NVARCHAR.

When an embedded SQL client performs a DESCRIBE on a NVARCHAR column, the data type returned is either DT_VARCHAR or DT_NVARCHAR, depending on whether the db_change_nchar_charset function has been called. See "db_change_nchar_charset function" [*SQL Anywhere Server - Programming*].

Also, when an embedded SQL client performs a DESCRIBE on an NVARCHAR column, the length returned is the maximum byte length in the client's NCHAR character set. For example, for an embedded SQL client using the Western European character set cp1252 as the NCHAR character set, an NVARCHAR(10) column is described as type DT_NVARCHAR of length 10 (10 characters multiplied by a maximum of one byte per character). For an embedded SQL client using the Japanese character set cp932, the same column is described as type DT_NVARCHAR of length 20 (10 characters multiplied by a maximum two bytes per character).

For ODBC, NVARCHAR is described as SQL_WCHAR or SQL_WVARCHAR, depending on the odbc_distingish_char_and_varchar option. See "odbc_distinguish_char_and_varchar option [database]" [*SQL Anywhere Server - Database Administration*].

See also

- “NCHAR data type” on page 80
- “LONG NVARCHAR data type” on page 79
- “VARCHAR data type” on page 84

Standards and compatibility

- **SQL/2003** Vendor extension.

TEXT data type

The TEXT data type stores character data of arbitrary length.

Syntax

TEXT

Remarks

TEXT is a domain, implemented as a LONG VARCHAR.

See also

- “LONG VARCHAR data type” on page 80
- “NTEXT data type” on page 81

Standards and compatibility

- **SQL/2003** Vendor extension.

UNIQUEIDENTIFIERSTR data type

The UNIQUEIDENTIFIERSTR data type is a domain implemented as CHAR(36).

Syntax

UNIQUEIDENTIFIERSTR

Remarks

Used for remote data access, when mapping Microsoft SQL Server uniqueidentifier columns.

See also

- “Data type conversions: Microsoft SQL Server” [*SQL Anywhere Server - SQL Usage*]
- “STRTOUUID function [String]” on page 316

Standards and compatibility

- **SQL/2003** Vendor extension.

VARCHAR data type

The VARCHAR data type stores character data, up to 32767 bytes.

Syntax

VARCHAR [**(** *max-length* [**CHAR** | **CHARACTER**] **)**]

Parameters

- **max-length** The maximum length of the string. If byte-length semantics are used (CHAR or CHARACTER is *not* specified as part of the length), then the length is in bytes, and the length must be in the range of 1 to 32767. If the length is not specified, then it is 1.

  If character-length semantics are used (CHAR or CHARACTER is specified as part of the length), then the length is in characters, and you must specify *max-length*. When using character-length semantics, the length multiplied by the maximum length of a character in the database encoding must not exceed 32767 bytes. The following table shows the maximum lengths for the supported types of character sets:

Character set	Maximum length of VARCHAR
Single-byte character set	32767 characters
Double-byte character set	16383 characters
UTF-8	8191 characters

Remarks

Multibyte characters can be stored as VARCHAR, but the declared length refers to bytes, not characters.

VARCHAR can also be specified as CHAR VARYING or CHARACTER VARYING. Regardless of which syntax is used, the data type is described as VARCHAR.

Using character-length semantics may impact what is returned when a client application performs a DESCRIBE on a column, depending on the interface used. For example, when an embedded SQL client application performs a DESCRIBE on a column that was declared using byte-length semantics, the length returned is the byte length specified. So, a VARCHAR(10) column is described as type DT_VARCHAR with a length of 10 bytes. However, when an embedded SQL client application performs a DESCRIBE on a column that was declared using character-length semantics, the length returned is the maximum byte length in the client's CHAR character set. For example, for a client that is using UTF-8 as the CHAR character set, a VARCHAR(10 CHAR) column is described as type DT_VARCHAR with a length of 40 bytes (10 characters multiplied by a maximum of four bytes per character).

See also

- "CHAR data type" on page 78
- "LONG VARCHAR data type" on page 80
- "NVARCHAR data type" on page 82

Standards and compatibility

- **SQL/2003** Compatible with SQL/2003. Character-length semantics is a vendor extension.

XML data type

The XML data type stores character data of arbitrary length, and is used to store XML documents.

Syntax

XML

Remarks

The maximum size is 2 GB.

Data of type XML is not quoted when generating element content from relational data.

You can cast between the XML data type and any other data type that can be cast to or from a string. Note that there is no checking that the string is well-formed when it is cast to XML.

For information about using the XML data type when generating XML elements, see "Storing XML documents in relational databases" [*SQL Anywhere Server - SQL Usage*].

When an embedded SQL client application performs a DESCRIBE on an XML column, it is described as LONG VARCHAR.

See also

- "Using XML in the database" [*SQL Anywhere Server - SQL Usage*]

Standards and compatibility

- **SQL/2003** Compatible with SQL/2003.

Numeric data types

The numeric data types are used for storing numerical data.

The NUMERIC and DECIMAL data types, and the various kinds of INTEGER data types, are sometimes called **exact** numeric data types, in contrast to the **approximate** numeric data types FLOAT, DOUBLE, and REAL.

The exact numeric data types are those for which precision and scale values can be specified, while approximate numeric data types are stored in a predefined manner. *Only exact numeric data is guaranteed accurate to the least significant digit specified after an arithmetic operation*.

Data type lengths and precision of less than one are not allowed.

Compatibility

Only the NUMERIC data type with scale = 0 can be used for the Transact-SQL identity column.

Be careful using default precision and scale settings for NUMERIC and DECIMAL data types, because these settings could be different in other database solutions. In SQL Anywhere, the default precision is 30 and the default scale is 6.

You should avoid default precision and scale settings for NUMERIC and DECIMAL data types, because these are different between SQL Anywhere and Adaptive Server Enterprise. In SQL Anywhere, the default precision is 30 and the default scale is 6. In Adaptive Server Enterprise, the default precision is 18 and the default scale is 0.

The FLOAT (*p*) data type is a synonym for REAL or DOUBLE, depending on the value of *p*. For SQL Anywhere, the cutoff is platform-dependent, but on all platforms the cutoff value is greater than 15.

For information about changing the defaults by setting database options, see "precision option [database]" [*SQL Anywhere Server - Database Administration*] and "scale option [database]" [*SQL Anywhere Server - Database Administration*].

BIGINT data type

The BIGINT data type is used to store BIGINTs, which are integers requiring 8 bytes of storage.

Syntax

[**UNSIGNED**] **BIGINT**

Remarks

The BIGINT data type is an exact numeric data type: its accuracy is preserved after arithmetic operations.

A BIGINT value requires 8 bytes of storage.

The range for signed BIGINT values is -2^{63} to $2^{63} - 1$, or -9223372036854775808 to 9223372036854775807.

The range for unsigned BIGINT values is 0 to $2^{64} - 1$, or 0 to 18446744073709551615.

By default, the data type is signed.

When converting a string to a BIGINT, leading and trailing spaces are removed. If the leading character is '+' it is ignored. If the leading character is '-' the remaining digits are interpreted as a negative number. Leading '0' characters are skipped, and the remaining characters are converted to an integer value. An error is returned if the value is out of the valid range for the destination data type, if the string contains illegal characters, or if the string cannot be decoded as an integer value.

See also

- "BIT data type" on page 87
- "INTEGER data type" on page 90
- "SMALLINT data type" on page 93
- "TINYINT data type" on page 93
- "Numeric functions" on page 129
- "Aggregate functions" on page 124

Standards and compatibility

- **SQL/2003** Vendor extension.

BIT data type

The BIT data type is used to store a bit (0 or 1).

Syntax

BIT

Remarks

BIT is an integer type that can store the values 0 or 1.

By default, the BIT data type does not allow NULL.

When converting a string to a BIT, leading and trailing spaces are removed. If the leading character is '+' it is ignored. If the leading character is '-' the remaining digits are interpreted as a negative number. Leading '0' characters are skipped, and the remaining characters are converted to an integer value. An error is returned if the value is out of the valid range for the destination data type, if the string contains illegal characters, or if the string cannot be decoded as an integer value.

See also

- "BIGINT data type" on page 86
- "INTEGER data type" on page 90
- "SMALLINT data type" on page 93
- "TINYINT data type" on page 93
- "Numeric functions" on page 129
- "Aggregate functions" on page 124

Standards and compatibility

- **SQL/2003** Vendor extension.

DECIMAL data type

The DECIMAL data type is a decimal number with *precision* total digits and with *scale* digits after the decimal point.

Syntax

DECIMAL [**(** *precision* [, *scale*] **)**]

Parameters

- **precision** An integer expression between 1 and 127, inclusive, that specifies the number of digits in the expression. The default setting is 30.
- **scale** An integer expression between 0 and 127, inclusive, that specifies the number of digits after the decimal point. The scale value should always be less than, or equal to, the precision value. The default setting is 6.

  The defaults can be changed by setting database options. For information, see "precision option [database]" [*SQL Anywhere Server - Database Administration*] and "scale option [database]" [*SQL Anywhere Server - Database Administration*].

Remarks

The DECIMAL data type is an exact numeric data type; its accuracy is preserved to the least significant digit after arithmetic operations.

The storage required for a decimal number can be estimated as

```
2 + int( (before + 1)/2 ) + int( (after + 1)/2 )
```

The function int takes the integer portion of its argument, and before and after are the number of significant digits before and after the decimal point. The storage is based on the value being stored, not on the maximum precision and scale allowed in the column.

DECIMAL can also be specified as DEC. Regardless of which syntax is used, the data type is described as DECIMAL.

DECIMAL is semantically equivalent to NUMERIC.

See also

- "FLOAT data type" on page 89
- "REAL data type" on page 92
- "DOUBLE data type" on page 89
- "NUMERIC data type" on page 91
- "Numeric functions" on page 129
- "Aggregate functions" on page 124

Standards and compatibility

- **SQL/2003** Compatible with SQL/2003.

DOUBLE data type

The DOUBLE data type is used to store double-precision floating-point numbers.

Syntax

DOUBLE [**PRECISION**]

Remarks

The DOUBLE data type holds a double-precision floating point number. An approximate numeric data type, it is subject to rounding errors after arithmetic operations. The approximate nature of DOUBLE values means that queries using equalities should generally be avoided when comparing DOUBLE values.

DOUBLE values require 8 bytes of storage.

The range of values is -1.79769313486231e+308 to 1.79769313486231e+308, with numbers close to zero as small as 2.22507385850721e-308. Values held as DOUBLE are accurate to 15 significant digits, but may be subject to round-off error beyond the fifteenth digit.

See also

- "FLOAT data type" on page 89
- "REAL data type" on page 92
- "DECIMAL data type" on page 88
- "NUMERIC data type" on page 91
- "Numeric functions" on page 129
- "Aggregate functions" on page 124
- "Converting between numeric sets" on page 117

Standards and compatibility

- **SQL/2003** Compatible with SQL/2003.

FLOAT data type

The FLOAT data type is used to store a floating point number, which can be single or double precision.

Syntax

FLOAT [**(** *precision* **)**]

Parameters

- **precision** An integer expression that specifies the number of bits in the mantissa. A mantissa is the decimal part of a logarithm. For example, in the logarithm 5.63428, the mantissa is 0.63428. The IEEE standard 754 floating-point precision is as follows:

Supplied precision value	Decimal precision	Equivalent SQL data type	Storage size
1-24	7 decimal digits	REAL	4 bytes

Supplied precision value	Decimal precision	Equivalent SQL data type	Storage size
25-53	15 decimal digits	DOUBLE	8 bytes

Remarks

When a column is created using the FLOAT (*precision*) data type, columns on all platforms are guaranteed to hold the values to at least the specified minimum precision. In contrast, REAL and DOUBLE do not guarantee a platform-independent minimum precision.

If *precision* is not supplied, the FLOAT data type is a single precision floating point number, equivalent to the REAL data type, and requires 4 bytes of storage.

If *precision* is supplied, the FLOAT data type is either single or double precision, depending on the value of precision specified. The cutoff between REAL and DOUBLE is platform-dependent. Single precision FLOAT values require 4 bytes of storage, and double precision FLOAT values require 8 bytes.

The FLOAT data type is an approximate numeric data type. It is subject to round-off errors after arithmetic operations. The approximate nature of FLOAT values means that queries using equalities should generally be avoided when comparing FLOAT values.

See also

- "DOUBLE data type" on page 89
- "REAL data type" on page 92
- "DECIMAL data type" on page 88
- "NUMERIC data type" on page 91
- "Numeric functions" on page 129
- "Aggregate functions" on page 124

Standards and compatibility

- **SQL/2003** Compatible with SQL/2003.

INTEGER data type

The INTEGER data type is used to store integers that require 4 bytes of storage.

Syntax

[**UNSIGNED**] **INTEGER**

Remarks

The INTEGER data type is an exact numeric data type; its accuracy is preserved after arithmetic operations.

If you specify UNSIGNED, the integer can never be assigned a negative number. By default, the data type is signed.

The range for signed integers is -2^{31} to $2^{31} - 1$, or -2147483648 to 2147483647.

The range for unsigned integers is 0 to $2^{32} - 1$, or 0 to 4294967295.

INTEGER can also be specified as INT. Regardless of which syntax is used, the data type is described as INTEGER.

When converting a string to a INTEGER, leading and trailing spaces are removed. If the leading character is '+' it is ignored. If the leading character is '-' the remaining digits are interpreted as a negative number. Leading '0' characters are skipped, and the remaining characters are converted to an integer value. An error is returned if the value is out of the valid range for the destination data type, if the string contains illegal characters, or if the string cannot be decoded as an integer value.

See also

- "BIGINT data type" on page 86
- "BIT data type" on page 87
- "SMALLINT data type" on page 93
- "TINYINT data type" on page 93
- "Numeric functions" on page 129
- "Aggregate functions" on page 124

Standards and compatibility

- **SQL/2003** Compatible with SQL/2003. The UNSIGNED keyword is a vendor extension.

NUMERIC data type

The NUMERIC data types is used to store decimal numbers with *precision* total digits and with *scale* digits after the decimal point.

Syntax

NUMERIC [**(** *precision* [, *scale*] **)**]

Parameters

- **precision** An integer expression between 1 and 127, inclusive, that specifies the number of digits in the expression. The default setting is 30.
- **scale** An integer expression between 0 and 127, inclusive, that specifies the number of digits after the decimal point. The scale value should always be less than or equal to the precision value. The default setting is 6.

  The defaults can be changed by setting database options. For information, see "precision option [database]" [*SQL Anywhere Server - Database Administration*] and "scale option [database]" [*SQL Anywhere Server - Database Administration*].

Remarks

The NUMERIC data type is an exact numeric data type; its accuracy is preserved to the least significant digit after arithmetic operations.

The number of bytes required to store a decimal number can be estimated as

```
2 + INT( (BEFORE+1)/2 ) + INT( (AFTER+1)/2 )
```

The INT function takes the integer portion of its argument, and BEFORE and AFTER are the number of significant digits before and after the decimal point. The storage is based on the value being stored, not on the maximum precision and scale allowed in the column.

NUMERIC is semantically equivalent to DECIMAL.

See also

- "FLOAT data type" on page 89
- "REAL data type" on page 92
- "DOUBLE data type" on page 89
- "DECIMAL data type" on page 88
- "Numeric functions" on page 129
- "Aggregate functions" on page 124
- "Converting between numeric sets" on page 117

Standards and compatibility

- **SQL/2003** Compatible with SQL/2003 if the scale option is set to zero.

REAL data type

The REAL data type is used to store single-precision floating-point numbers stored in 4 bytes.

Syntax

REAL

Remarks

The REAL data type is an approximate numeric data type; it is subject to roundoff errors after arithmetic operations.

The range of values is -3.402823e+38 to 3.402823e+38, with numbers close to zero as small as 1.175494e-38. Values held as REAL are accurate to 7 significant digits, but may be subject to round-off error beyond the sixth digit.

The approximate nature of REAL values means that queries using equalities should generally be avoided when comparing REAL values

See also

- "DOUBLE data type" on page 89
- "FLOAT data type" on page 89
- "DECIMAL data type" on page 88
- "NUMERIC data type" on page 91
- "Numeric functions" on page 129
- "Aggregate functions" on page 124

Standards and compatibility

- **SQL/2003** Compatible with SQL/2003.

SMALLINT data type

The SMALLINT data type is used to store integers that require 2 bytes of storage.

Syntax

[**UNSIGNED**] **SMALLINT**

Remarks

The SMALLINT data type is an exact numeric data type; its accuracy is preserved after arithmetic operations. It requires 2 bytes of storage.

The range for signed SMALLINT values is -2^{15} to 2^{15} - 1, or -32768 to 32767.

The range for unsigned SMALLINT values is 0 to 2^{16} - 1, or 0 to 65535.

When converting a string to a SMALLINT, leading and trailing spaces are removed. If the leading character is '+' it is ignored. If the leading character is '-' the remaining digits are interpreted as a negative number. Leading '0' characters are skipped, and the remaining characters are converted to an integer value. An error is returned if the value is out of the valid range for the destination data type, if the string contains illegal characters, or if the string cannot be decoded as an integer value.

See also

- "BIGINT data type" on page 86
- "BIT data type" on page 87
- "INTEGER data type" on page 90
- "TINYINT data type" on page 93
- "Numeric functions" on page 129
- "Aggregate functions" on page 124

Standards and compatibility

- **SQL/2003** Compatible with SQL/2003. The UNSIGNED keyword is a vendor extension.

TINYINT data type

The TINYINT data type is used to store unsigned integers requiring 1 byte of storage.

Syntax

[**UNSIGNED**] **TINYINT**

Remarks

The TINYINT data type is an exact numeric data type; its accuracy is preserved after arithmetic operations.

You can explicitly specify TINYINT as UNSIGNED, but the UNSIGNED modifier has no effect as the type is always unsigned.

The range for TINYINT values is 0 to 2^{8} - 1, or 0 to 255.

In embedded SQL, TINYINT columns should not be fetched into variables defined as char or unsigned char, since the result is an attempt to convert the value of the column to a string and then assign the first byte to the variable in the program. Instead, TINYINT columns should be fetched into 2-byte or 4-byte integer columns. Also, to send a TINYINT value to a database from an application written in C, the type of the C variable should be integer.

When converting a string to a TINYINT, leading and trailing spaces are removed. If the leading character is '+' it is ignored. If the leading character is '-' the remaining digits are interpreted as a negative number. Leading '0' characters are skipped, and the remaining characters are converted to an integer value. An error is returned if the value is out of the valid range for the destination data type, if the string contains illegal characters, or if the string cannot be decoded as an integer value.

See also

- "BIGINT data type" on page 86
- "BIT data type" on page 87
- "INTEGER data type" on page 90
- "SMALLINT data type" on page 93
- "Numeric functions" on page 129
- "Aggregate functions" on page 124

Standards and compatibility

- **SQL/2003** Vendor extension.

Money data types

Money data types are used for storing monetary data.

MONEY data type

The MONEY data type stores monetary data.

Syntax

MONEY

Remarks

MONEY is implemented as a domain, as NUMERIC(19,4).

See also

- "SMALLMONEY data type" on page 95
- "Numeric functions" on page 129
- "Aggregate functions" on page 124

Standards and compatibility

- **SQL/2003** Vendor extension.

SMALLMONEY data type

The SMALLMONEY data type is used to store monetary data that is less than one million currency units.

Syntax

SMALLMONEY

Remarks

SMALLMONEY is implemented as a domain, as NUMERIC(10,4).

See also

- "MONEY data type" on page 95
- "Numeric functions" on page 129
- "Aggregate functions" on page 124

Standards and compatibility

- **SQL/2003** Vendor extension.

Bit array data types

Bit arrays are used for storing bit data (0s and 1s).

A **bit array** data type is used to store an array of bits. The bit array data types supported by SQL Anywhere include VARBIT and LONG VARBIT.

LONG VARBIT data type

The LONG VARBIT data type is used to store arbitrary length bit arrays.

Syntax

LONG VARBIT

Remarks

Used to store arbitrary length array of bits (1s and 0s), or bit arrays longer than 32767 bits.

LONG VARBIT can also be specified as LONG BIT VARYING. Regardless of which syntax is used, the data type is described as LONG VARBIT.

See also

- "BIT data type" on page 87
- "VARBIT data type" on page 96
- "Converting bit arrays" on page 116
- "Bit array functions" on page 125
- "Aggregate functions" on page 124

Standards and compatibility

- **SQL/2003** Vendor extension.

VARBIT data type

The VARBIT data type is used for storing bit arrays that are under 32767 bits in length.

Syntax

VARBIT [**(***max-length* **)**]

Parameters

- **max-length** The maximum length of the bit array, in bits. The length must be in the range 1 to 32767. If the length is not specified, then it is 1.

Remarks

VARBIT can also be specified as BIT VARYING. Regardless of which syntax is used, the data type is described as VARBIT.

See also

- "BIT data type" on page 87
- "LONG VARBIT data type" on page 96
- "Converting bit arrays" on page 116
- "Bit array functions" on page 125
- "Aggregate functions" on page 124

Standards and compatibility

- **SQL/2003** Vendor extension.

Date and time data types

The following list provides a quick overview of how dates are handled:

- SQL Anywhere always returns correct values for any legal arithmetic and logical operations on dates, regardless of whether the calculated values span different centuries.
- The internal storage of dates by SQL Anywhere always explicitly includes the century portion of a year value.
- The operation of SQL Anywhere is unaffected by any return value, including the current date.
- Date values can always be output in full century format.

How dates are stored

Dates containing year values are used internally and stored in SQL Anywhere databases using either of the following data types:

Data type	Contains	Stored in	Range of possible values
DATE	Calendar date (year, month, day)	4-bytes	0001-01-01 to 9999-12-31
TIMESTAMP	Time stamp (year, month, day, hour minute, second, and fraction of second accurate to 6 decimal places)	8-bytes	0001-01-01 to 9999-12-31 (precision of the time portion of TIMESTAMP is dropped prior to 1600-02-28 23:59:59 and after 7911-01-01 00:00:00)

For more information about SQL Anywhere date and time data types see "Date and time data types" on page 98.

Sending dates and times to the database

Date and times may be sent to the database in one of the following ways:

- Using any interface, as a string
- Using ODBC, as a TIMESTAMP structure
- Using embedded SQL, as a SQLDATETIME structure

When a time is sent to the database as a string (for the TIME data type) or as part of a string (for TIMESTAMP or DATE data types), the hours, minutes, and seconds must be separated by colons in the format *hh*:*mm*:*ss.sss*, but can appear anywhere in the string. The following are valid and unambiguous strings for specifying times:

```
21:35 -- 24 hour clock if no am or pm specified
10:00pm -- pm specified, so interpreted as 12 hour clock
```

```
10:00 -- 10:00am in the absence of pm
10:23:32.234 -- seconds and fractions of a second included
```

When a date is sent to the database as a string conversion to a DATE or TIMESTAMP data type is automatic. The string can be supplied in one of two ways:

- As a string of format *yyyy/mm/dd* or *yyyy-mm-dd*, which is interpreted unambiguously by the database.
- As a string interpreted according to the date_order database option. See "date_order option [database]" [*SQL Anywhere Server - Database Administration*].

Transact-SQL string-to-date/time conversions

Converting strings to date and time data types.

If a string containing only a time value (no date) is converted to a date/time data type, SQL Anywhere uses the current date.

If the fraction portion of a time is less than 3 digits, SQL Anywhere interprets the value the same way regardless of the whether it is preceded by a period or a colon: one digit means tenths, two digits mean hundredths, and three digits mean thousandths.

Examples

SQL Anywhere converts the milliseconds value in the same manner regardless of the separator.

```
12:34:56.7 to 12:34:56.700
12:34:56:7 to 12:34:56.700
12.34.56.78 to 12:34:56.780
12.34.56:78 to 12:34:56.780
12:34:56.789 to 12:34:56.789
12:34:56:789 to 12:34:56.789
```

Retrieving dates and times from the database

Dates and times may be retrieved from the database in one of the following ways:

- Using any interface, as a string
- Using ODBC, as a TIMESTAMP structure
- Using embedded SQL, as a SQLDATETIME structure

When a date or time is retrieved as a string, it is retrieved in the format specified by the database options date_format, time_format and timestamp_format. For descriptions of these options, see "SET OPTION statement" on page 742.

For information about functions that deal with dates and times, see "Date and time functions" on page 126. The following arithmetic operators are allowed on dates:

- **timestamp + integer** Add the specified number of days to a date or timestamp.

- **timestamp - integer** Subtract the specified number of days from a date or timestamp.
- **date - date** Compute the number of days between two dates or timestamps.
- **date + time** Create a timestamp combining the given date and time.

Leap Years

SQL Anywhere uses a globally accepted algorithm for determining which years are leap years. Using this algorithm, a year is considered a leap year if it is divisible by four, unless the year is a century date (such as the year 1900), in which case it is a leap year only if it is divisible by 400.

SQL Anywhere handles all leap years correctly. For example, the following SQL statement results in a return value of "Tuesday":

```
SELECT DAYNAME('2000-02-29');
```

SQL Anywhere accepts February 29, 2000—a leap year—as a date, and using this date determines the day of the week.

However, the following statement is rejected by SQL Anywhere:

```
SELECT DAYNAME('2001-02-29');
```

This statement results in an error (cannot convert '2001-02-29' to a date) because February 29th does not exist in the year 2001.

Comparing dates and times

By default, values stored as DATE do not have any hour or minute values, and so comparison of dates is straightforward.

The DATE data type can also contain a time, which introduces complications when comparing dates. If the time is not specified when a date is entered into the database, the time defaults to 0:00 or 12:00am (midnight). Any date comparisons with this option setting compare the times and the date. A database date value of 1999-05-23 10:00 is not equal to the constant 1999-05-23. The DATEFORMAT function or one of the other date functions can be used to compare parts of a date and time field. For example,

```
DATEFORMAT(invoice_date,'yyyy/mm/dd') = '1999/05/23';
```

If a database column requires only a date, client applications should ensure that times are not specified when data is entered into the database. This way, comparisons with date-only strings will work as expected.

If you want to compare a date to a string *as a string*, you must use the DATEFORMAT function or CAST function to convert the date to a string before comparing.

Using unambiguous dates and times

Dates in the format *yyyy/mm/dd* or *yyyy-mm-dd* are always recognized unambiguously as dates, regardless of the date_order setting. Other characters can be used as separators instead of '/' or '-'; for example, '?', a space character, or ','. You should use this format in any context where different users may be employing different date_order settings. For example, in stored procedures, use of the unambiguous date format prevents misinterpretation of dates according to the user's date_order setting.

Also, a string of the form *hh:mm:ss:ssss* is interpreted unambiguously as a time.

For combinations of dates and times, any unambiguous date and any unambiguous time yield an unambiguous date-time value. Also, the form *yyyy-mm-dd hh.mm.ss.sss* is an unambiguous date-time value. Periods can be used in the time only in combination with a date.

In other contexts, a more flexible date format can be used. SQL Anywhere can interpret a wide range of strings as dates. The interpretation depends on the setting of the database option date_order. The date_order database option can have the value *MDY*, *YMD*, or *DMY* (see "SET OPTION statement" on page 742). For example, the following statement sets the date_order option to *DMY*:

```
SET OPTION date_order = 'DMY' ;
```

The default date_order setting is YMD. The ODBC driver sets the date_order option to YMD whenever a connection is made. The value can still be changed using the SET TEMPORARY OPTION statement.

The database option date_order determines whether the string 10/11/12 is interpreted by the database as November 12, 2010; October 11, 2012; or November 10, 2012. The year, month, and day of a date string should be separated by some character (/, -, or space) and appear in the order specified by the date_order option.

The year can be supplied as either 2 or 4 digits. The value of the nearest_century option affects the interpretation of 2-digit years: 2000 is added to values less than nearest_century and 1900 is added to all other values. The default value of this option is 50. So, by default, 50 is interpreted as 1950 and 49 is interpreted 2049.

The month can be the name or number of the month. The hours and minutes are separated by a colon, but can appear anywhere in the string.

Notes

- It is recommended that you always specify the year using the four-digit format.
- With an appropriate setting of date_order, the following strings are all valid dates:

  ```
  99-05-23 21:35
  99/5/23
  1999/05/23
  May 23 1999
  23-May-1999
  Tuesday May 23, 1999 10:00pm
  ```

- If a string contains only a partial date specification, default values are used to fill out the date. The following defaults are used:
  - **year** This year
  - **month** No default
  - **day** 1 (useful for month fields; for example, May 1999 will be the date 1999-05-01 00:00)
  - **hour, minute, second, fraction** 0

DATE data type

The DATE data type is used to store calendar dates, such as a year, month and day.

Syntax

DATE

Remarks

The year can be from the year 0001 to 9999. The minimum date in SQL Anywhere is 0001-01-01 00:00:00.

For historical reasons, a DATE column can also contain an hour and minute. The TIMESTAMP data type is recommended for anything with hours and minutes.

The format in which DATE values are retrieved by applications is controlled by the date_format setting. For example, a date value representing the 19th of July, 2010 may be returned to an application as `2010/07/19`, or as `Jul 19, 2010`.

The way in which a string is interpreted by the database server as a date is controlled by the date_order option. For example, depending on the date_order setting, a value of `02/05/2002` supplied by an application for a DATE value may be interpreted in the database as the 2nd of May or the 5th of February.

A DATE value requires 4 bytes of storage.

See also

- "date_format option [database]" [*SQL Anywhere Server - Database Administration*]
- "date_order option [database]" [*SQL Anywhere Server - Database Administration*]
- "DATETIME data type" on page 102
- "SMALLDATETIME data type" on page 103
- "TIMESTAMP data type" on page 104
- "Date and time functions" on page 126

Standards and compatibility

- **SQL/2003** Vendor extension.

DATETIME data type

The DATETIME data type is a domain, implemented as TIMESTAMP, used to store date and time information.

Syntax

DATETIME

See also

- "DATE data type" on page 102
- "SMALLDATETIME data type" on page 103
- "TIMESTAMP data type" on page 104
- "Date and time functions" on page 126

Standards and compatibility

- **SQL/2003** Vendor extension.

SMALLDATETIME data type

The SMALLDATETIME data type is a domain, implemented as TIMESTAMP, used to store date and time information.

Syntax

SMALLDATETIME

See also

- "DATE data type" on page 102
- "DATETIME data type" on page 102
- "TIMESTAMP data type" on page 104
- "Date and time functions" on page 126

Standards and compatibility

- **SQL/2003** Vendor extension.

TIME data type

The TIME data type is used to store the time of day, containing hour, minute, second and fraction of a second.

Syntax

TIME

Remarks

The fraction is stored to 6 decimal places. A TIME value requires 8 bytes of storage. (ODBC standards restrict TIME data type to an accuracy of seconds. For this reason you should not use TIME data types in WHERE clause comparisons that rely on a higher accuracy than seconds.)

See also

- "TIMESTAMP data type" on page 104
- "Date and time functions" on page 126

Standards and compatibility

- **SQL/2003** Vendor extension.

TIMESTAMP data type

The TIMESTAMP data type is used to store a point in time containing year, month, day, hour, minute, second and fraction of a second.

Syntax

TIMESTAMP

Remarks

The fraction is stored to 6 decimal places. A TIMESTAMP value requires 8 bytes of storage.

Although the range of possible dates for the TIMESTAMP data type is the same as the DATE type (covering years 0001 to 9999), the useful range of TIMESTAMP date types is from 1600-02-28 23:59:59 to 7911-01-01 00:00:00. Prior to, and after this range the time portion of the TIMESTAMP may be incomplete.

See also

- "TIME data type" on page 103
- "Date and time functions" on page 126

Standards and compatibility

- **SQL/2003** Vendor extension.

Binary data types

Binary data types are used for storing binary data, including images and other types of information that are not interpreted by the database.

BINARY data type

The BINARY data type is used to store binary data of a specified maximum length (in bytes).

Syntax

BINARY [**(** *max-length* **)**]

Parameters

- **max-length** The maximum length of the value, in bytes. The length must be in the range 1 to 32767. If the length is not specified, then it is 1.

Remarks

During comparisons, BINARY values are compared exactly byte for byte. This differs from the CHAR data type, where values are compared using the collation sequence of the database. If one binary string is a prefix of the other, the shorter string is considered to be less than the longer string.

Unlike CHAR values, BINARY values are not transformed during character set conversion.

BINARY is semantically equivalent to VARBINARY. It is a variable-length type. In other database management systems, BINARY is a fixed-length type.

See also

- “VARBINARY data type” on page 107
- “LONG BINARY data type” on page 106
- “String functions” on page 131

Standards and compatibility

- **SQL/2003** Vendor extension.

IMAGE data type

The IMAGE data type is used to store binary data of arbitrary length.

Syntax

IMAGE

Remarks

IMAGE is implemented as a domain, as LONG BINARY.

See also

- “LONG BINARY data type” on page 106
- “String functions” on page 131

Standards and compatibility

- **SQL/2003** Vendor extension.

LONG BINARY data type

The LONG BINARY data type is used to store binary data of arbitrary length.

Syntax

LONG BINARY

Remarks

The maximum size is 2 GB.

See also

- “BINARY data type” on page 105
- “VARBINARY data type” on page 107
- “String functions” on page 131

Standards and compatibility

- **SQL/2003** Vendor extension.

UNIQUEIDENTIFIER data type

The UNIQUEIDENTIFIER data type is used to store UUID (also known as GUID) values.

Syntax

UNIQUEIDENTIFIER

Remarks

The UNIQUEIDENTIFIER data type is typically used for a primary key or other unique column to hold UUID (Universally Unique Identifier) values that uniquely identify rows. The NEWID function generates UUID values in such a way that a value produced on one computer will not match a UUID produced on another computer. UNIQUEIDENTIFIER values generated using NEWID can therefore be used as keys in a synchronization environment.

For example:

```
CREATE TABLE T1 (
    pk UNIQUEIDENTIFIER PRIMARY KEY DEFAULT NEWID(),
    c1 INT );
```

UUID values are also referred to as GUIDs (Globally Unique Identifier). UUID values contain hyphens so they are compatible with other RDBMSs.

SQL Anywhere automatically converts UNIQUEIDENTIFIER values between string and binary values as needed.

UNIQUEIDENTIFIER values are stored as BINARY(16), but are described to client applications as BINARY(36). This description ensures that if the client fetches the value as a string, it has allocated enough space for the result. For ODBC client applications, uniqueidentifier values appear as a SQL_GUID type.

See also

- "The NEWID default" [*SQL Anywhere Server - SQL Usage*]
- "NEWID function [Miscellaneous]" on page 252
- "UUIDTOSTR function [String]" on page 335
- "STRTOUUID function [String]" on page 316
- "String functions" on page 131

Standards and compatibility

- **SQL/2003** Vendor extension.

VARBINARY data type

The VARBINARY data type is used to store binary data of a specified maximum length (in bytes).

Syntax

VARBINARY [**(** *max-length* **)**]

Parameters

- **max-length** The maximum length of the value, in bytes. The length must be in the range 1 to 32767. If the length is not specified, then it is 1.

Remarks

During comparisons, VARBINARY values are compared exactly byte for byte. This differs from the CHAR data type, where values are compared using the collation sequence of the database. If one binary string is a prefix of the other, the shorter string is considered to be less than the longer string.

Unlike CHAR values, VARBINARY values are not transformed during character set conversion.

VARBINARY can also be specified as BINARY VARYING. Regardless of which syntax is used, the data type is described as VARBINARY.

See also

- "BINARY data type" on page 105
- "LONG BINARY data type" on page 106
- "String functions" on page 131

Standards and compatibility

- **SQL/2003** Vendor extension.

Domains

Domains are aliases for built-in data types, including precision and scale values where applicable, and optionally including DEFAULT values and CHECK conditions. Some domains, such as the monetary data types, are pre-defined in SQL Anywhere, but you can add more of your own.

Domains, also called **user-defined data types**, allow columns throughout a database to be automatically defined on the same data type, with the same NULL or NOT NULL condition, with the same DEFAULT setting, and with the same CHECK condition. Domains encourage consistency throughout the database and can eliminate some types of errors.

Simple domains

Domains are created using the CREATE DOMAIN statement. For a full description of the syntax, see "CREATE DOMAIN statement" on page 443.

The following statement creates a data type named street_address, which is a 35-character string.

```
CREATE DOMAIN street_address CHAR( 35 );
```

CREATE DATATYPE can be used as an alternative to CREATE DOMAIN, but is not recommended.

Resource authority is required to create data types. Once a data type is created, the user ID that executed the CREATE DOMAIN statement is the owner of that data type. Any user can use the data type. Unlike with other database objects, the owner name is never used to prefix the data type name.

The street_address data type may be used in exactly the same way as any other data type when defining columns. For example, the following table with two columns has the second column as a street_address column:

```
CREATE TABLE twocol (
    id INT,
    street street_address
);
```

Domains can be dropped by their owner or by a user with DBA authority, using the DROP DOMAIN statement:

```
DROP DOMAIN street_address;
```

This statement can be carried out only if the data type is not used in any table in the database. If you attempt to drop a domain that is in use, the message "Primary key for row in table 'SYSUSERTYPE' is referenced in another table" appears.

Constraints and defaults with domains

Many of the attributes associated with columns, such as allowing NULL values, having a DEFAULT value, and so on, can be built into a domain. Any column that is defined on the data type automatically inherits the NULL setting, CHECK condition, and DEFAULT values. This allows uniformity to be built into columns with a similar meaning throughout a database.

For example, many primary key columns in the SQL Anywhere sample database are integer columns holding ID numbers. The following statement creates a data type that may be useful for such columns:

```
CREATE DOMAIN id INT
NOT NULL
DEFAULT AUTOINCREMENT
CHECK( @col > 0 );
```

By default, a column created using the id data type does not allow NULLs, defaults to an auto-incremented value, and must hold a positive number. Any identifier could be used instead of *col* in the *@col* variable.

The attributes of a data type can be overridden by explicitly providing attributes for the column. A column created using the id data type with NULL values explicitly allowed does allow NULLs, regardless of the setting in the id data type.

Compatibility

- **Named constraints and defaults** In SQL Anywhere, domains are created with a base data type, and optionally a NULL or NOT NULL condition, a default value, and a CHECK condition. Named constraints and named defaults are not supported.
- **Creating data types** In SQL Anywhere, you can use the sp_addtype system procedure to add a domain, or you can use the CREATE DOMAIN statement.

Data type conversions

Type conversions can happen automatically, or they can be explicitly requested using the CAST or CONVERT function. The following functions can also be used to force type conversions :

- **DATE function** Converts the expression into a date, and removes any hours, minutes or seconds. Conversion errors may be reported.
- **STRING function** This function is equivalent to `CAST( value AS LONG VARCHAR)`.
- **VALUE+0.0** Equivalent to CAST(value AS DECIMAL).

The following list is a high-level view of automatic data type conversions:

- If a string is used in a numeric expression or as an argument to a function that expects a numeric argument, the string is converted to a number.
- If a number is used in a string expression or as a string function argument, it is converted to a string before being used.
- All date constants are specified as strings. The string is automatically converted to a date before use.

There are certain cases where the automatic database conversions are not appropriate. For example, the automatic data type conversion fails in the example below.

```
'12/31/90' + 5
'a' > 0
```

See also

- "Data type conversion functions" on page 125
- "DATE function [Date and time]" on page 172
- "STRING function [String]" on page 315
- "CAST function [Data type conversion]" on page 148

Comparisons between data types

When a comparison (such as =) is performed between arguments with different data types, one or more arguments must be converted so that the comparison operation is done using one data type.

Some rules may lead to conversions that fail, or lead to unexpected results from the comparison. In these cases, you should explicitly convert one of the arguments using CAST or CONVERT.

You can override these conversion rules by explicitly casting arguments to another type. For example, if you want to compare a DATE and a CHAR as a CHAR, then you need to explicitly cast the DATE to a CHAR. See "CAST function [Data type conversion]" on page 148.

Substitution characters

When a character cannot be represented in the character set into which it is being converted, a substitution character is used instead. Conversions of this type are considered **lossy**; the original character is lost if it cannot be represented in the destination character set.

Also, not only may different character sets have a different substitution character, but the substitution character for one character set may be a non-substitution character in another character set. This is important to understand when multiple conversions are performed on a character because the final character may not appear as the expected substitution character of the destination character set.

For example, suppose that the client character set is Windows-1252, and the database character set is ISO_8859-1:1987, the U.S. default for some versions of Unix. Then, suppose a non-Unicode client application (for example, embedded SQL) attempts to insert the euro symbol into a CHAR, VARCHAR, or LONG VARCHAR column. Since the character does not exist in the CHAR character set, the substitution character for ISO_8859-1:1987, 0x1A, is inserted.

Now, if this same ISO_8859-1:1987 substitution character is then fetched as Unicode (for example, by doing a `SELECT * FROM t` into a SQL_C_WCHAR bound column in ODBC), this character becomes the Unicode code point U+001A. (In Unicode the code point U+001A is the record separator control character.) However, the substitution character for Unicode is the code point U+FFFD. This example illustrates that even if your data contains substitution characters, those characters, due to multiple conversions, may not be converted to the substitution character of the destination character set.

Therefore, it is important to understand and test how substitution characters are used when converting between multiple character sets.

The on_charset_conversion_failure option can help determine the behavior during conversion when a character cannot be represented in the destination character set. See "on_charset_conversion_failure option [database]" [*SQL Anywhere Server - Database Administration*].

See also

- "Data type conversions" on page 111
- "Comparisons between CHAR and NCHAR" on page 112
- "on_charset_conversion_failure option [database]" [*SQL Anywhere Server - Database Administration*]

Comparisons between CHAR and NCHAR

When a comparison is performed between a value of CHAR type (CHAR, VARCHAR, LONG VARCHAR) and a value of NCHAR type (NCHAR, NVARCHAR, LONG NVARCHAR), SQL Anywhere uses inference rules to determine the type in which the comparison should be performed. Generally, if one value is based on a column reference and the other is not, the comparison is performed in the type of the value containing the column reference.

The inference rules revolve around whether a value is based on a column reference. In the case where one value is a variable, a host variable, a literal constant, or a complex expression not based on a column reference and the other value is based on a column reference, then the constant-based value is implicitly cast to the type of the column-based value.

Following are the inference rules, in the order in which they are applied:

- If the NCHAR value is based on a column reference, the CHAR value is implicitly cast to NCHAR, and the comparison is done as NCHAR. This includes the case where both the NCHAR and CHAR value are based on column references.
- Else if the NCHAR value is not based on a column reference, and the CHAR value is based on a column reference, the NCHAR value is implicitly cast to CHAR, and the comparison is done as CHAR.

  It is important to consider the setting for the on_charset_conversion_failure option if you anticipate NCHAR to CHAR conversions since this option controls behavior if an NCHAR character cannot be represented in the CHAR character set. For further explanation, see "Converting NCHAR to CHAR" on page 115.
- Else if neither value is based on a column reference, then the CHAR value is implicitly cast to NCHAR and the comparison is done as NCHAR.

Examples

The condition `Employees.GivenName = N'Susan'` compares a CHAR column (Employees.GivenName) to the literal N'Susan'. The value N'Susan' is cast to CHAR, and the comparison is performed as if it had been written as:

```
Employees.GivenName  = CAST( N'Susan' AS CHAR )
```

Alternatively, the condition `Employees.GivenName = T.nchar_column` would find that the value T.nchar_column can not be cast to CHAR. The comparison would be performed as if it were written as follows, and an index on Employees.GivenName can not be used:

```
CAST( Employees.GivenName AS NCHAR ) = T.nchar_column;
```

See also

- "Converting NCHAR to CHAR" on page 115
- "Substitution characters" on page 112
- "CAST function [Data type conversion]" on page 148
- "CONVERT function [Data type conversion]" on page 160
- "CAST function [Data type conversion]" on page 148
- "on_charset_conversion_failure option [database]" [*SQL Anywhere Server - Database Administration*]

Comparisons between numeric data types

SQL Anywhere uses the following rules when comparing numeric data types. The rules are examined in the order listed, and the first rule that applies is used:

1. If one argument is TINYINT and the other is INTEGER, convert both to INTEGER and compare.
2. If one argument is TINYINT and the other is SMALLINT, convert both to SMALLINT and compare.
3. If one argument is UNSIGNED SMALLINT and the other is INTEGER, convert both to INTEGER and compare.

4. If the data types of the arguments have a common super type, convert to the common super type and compare. The super types are the final data type in each of the following lists:
   - BIT » TINYINT » UNSIGNED SMALLINT » UNSIGNED INTEGER » UNSIGNED BIGINT » NUMERIC
   - SMALLINT » INTEGER » BIGINT » NUMERIC
   - REAL » DOUBLE
   - CHAR » LONG VARCHAR
   - BINARY » LONG BINARY

   For example, if the two arguments are of types BIT and TINYINT, they are converted to NUMERIC.

Comparisons between time and date data types

SQL Anywhere uses the following rules when comparing time and date data types. The rules are examined in the order listed, and the first rule that applies is used:

1. If the data type of either argument is TIME, convert both to TIME and compare.
2. If either data type has the type DATE or TIMESTAMP, convert to both to TIMESTAMP and compare.

   For example, if the two arguments are of type REAL and DATE, they are both converted to TIMESTAMP.
3. If one argument has NUMERIC data type and the other has FLOAT, convert both to DOUBLE and compare.

Other comparisons

1. If the data types are a mixture of CHAR (such as CHAR, VARCHAR, LONG VARCHAR, and so on, but not NCHAR types), convert to LONG VARCHAR and compare.
2. If the data type of any argument is UNIQUEIDENTIFIER, convert to UNIQUEIDENTIFIER and compare.
3. If the data type of any argument is a bit array (VARBIT or LONG VARBIT), convert to LONG VARBIT and compare.
4. If one argument has CHARACTER data type and the other has BINARY data type, convert to BINARY and compare.
5. If one argument is a CHAR type, and the other argument is an NCHAR type, use predefined inference rules. See “Comparisons between CHAR and NCHAR” on page 112.
6. If no rule exists, convert to NUMERIC and compare.

   For example, if the two arguments have REAL and CHAR data types, they are both converted to NUMERIC.

Converting NCHAR to CHAR

NCHAR to CHAR conversions can occur as part of a comparison of CHAR and NCHAR data, or when specifically requested. This type of conversion is lossy because depending on the CHAR character set, there may be some NCHAR characters that can not be represented in the CHAR type. When an NCHAR character cannot be converted to CHAR, a substitution character from the CHAR character set is used instead. For single-byte character sets, this is usually hex 1A.

Depending on the setting of the on_charset_conversion_failure option, when a character cannot be converted, one of the following can happen:

- a substitute character is used, and no warning is issued
- a substitute character is used, and a warning is issued
- an error is returned

Therefore, it is important to consider this option when converting from NCHAR to CHAR. See "on_charset_conversion_failure option [database]" [*SQL Anywhere Server - Database Administration*].

See also

- "Comparisons between CHAR and NCHAR" on page 112
- "on_charset_conversion_failure option [database]" [*SQL Anywhere Server - Database Administration*]

Converting NULL constants to NUMERIC and string types

When converting a NULL constant to a NUMERIC, or to a string type such as CHAR, VARCHAR, LONG VARCHAR, BINARY, VARBINARY, and LONG BINARY the size is set to 0. For example:

`SELECT CAST( NULL AS CHAR )` returns CHAR(0)

`SELECT CAST( NULL AS NUMERIC )` returns NUMERIC(1,0)

Converting dates to strings

SQL Anywhere provides several functions for converting SQL Anywhere date and time values into a wide variety of strings and other expressions. It is possible in converting a date value into a string to reduce the year portion into a two-digit number representing the year, thereby losing the century portion of the date.

Wrong century values

Consider the following statement, which incorrectly converts a string representing the date January 1, 2000 into a string representing the date January 1, 1900 even though no database error occurs.

```
SELECT DATEFORMAT (
            DATEFORMAT('2000-01-01', 'Mmm dd/yy' ),
            'yyyy-Mmm-dd' )
    AS Wrong_year;
```

SQL Anywhere automatically and correctly converts the unambiguous date string 2000-01-01 into a date value. However, the 'Mmm dd/yy' formatting of the inner, or nested, DATEFORMAT function drops the century portion of the date when it is converted back to a string and passed to the outer DATEFORMAT function.

Because the database option nearest_century in this case is set to 0, the outer DATEFORMAT function converts the string representing a date with a two-digit year value into a year between 1900 and 1999.

For more information about date and time functions, see “Date and time functions” on page 126.

Converting bit arrays

Converting integers to bit arrays

When converting an integer to a bit array, the length of the bit array is the number of bits in the integer type, and the bit array's value is the integer's binary representation. The most significant bit of the integer becomes the first bit of the array.

Examples

`SELECT CAST( CAST( 1 AS BIT ) AS VARBIT )` returns a VARBIT(1) containing 1.

`SELECT CAST( CAST( 8 AS TINYINT ) AS VARBIT )` returns a VARBIT(8) containing 00001000.

`SELECT CAST( CAST( 194 AS INTEGER ) AS VARBIT )` returns a VARBIT(32) containing 00000000000000000000000011000010.

Converting binary to bit arrays

When converting a binary type of length *n* to a bit array, the length of the array is *n* * 8 bits. The first 8 bits of the bit array become the first byte of the binary value. The most significant bit of the binary value becomes the first bit in the array. The next 8 bits of the bit array become the second byte of the binary value, and so on.

Examples

`SELECT CAST( 0x8181 AS VARBIT )` returns a VARBIT(16) containing 1000000110000001.

Converting characters to bit arrays

When converting a character data type of length *n* to a bit array, the length of the array is *n* bits. Each character must be either '0' or '1' and the corresponding bit of the array is assigned the value 0 or 1.

Example

`SELECT CAST( '001100' AS VARBIT )` returns a VARBIT(6) containing 001100.

Converting bit arrays to integers

When converting a bit array to an integer data type, the bit array's binary value is interpreted according to the storage format of the integer type, using the most significant bit first.

Example

`SELECT CAST( CAST( '11000010' AS VARBIT ) AS INTEGER )` returns 194 (11000010_2 = 0xC2 = 194).

Converting bit arrays to binary

When converting a bit array to a binary, the first 8 bits of the array become the first byte of the binary value. The first bit of the array becomes the most significant bit of the binary value. The next 8 bits are used as the second byte, and so on. If the length of the bit array is not a multiple of 8, then extra zeroes are used to fill the least significant bits of the last byte of the binary value.

Examples

`SELECT CAST( CAST( '1111' AS VARBIT ) AS BINARY )` returns 0xF0 (1111_2 becomes 11110000_2 = 0xF0).

`SELECT CAST( CAST( '0011000000110001' AS VARBIT ) AS BINARY )` returns 0x3031 (0011000000110001_2 = 0x3031).

Converting bit arrays to characters

When converting a bit array of length *n* bits to a character data type, the length of the result is *n* characters. Each character in the result is either '0' or '1', corresponding to the bit in the array.

Example

`SELECT CAST( CAST( '01110' AS VARBIT ) AS VARCHAR )` returns the character string '01110'.

Converting between numeric sets

When converting a DOUBLE type to a NUMERIC type, precision is maintained for the first 15 significant digits.

See also

- "CAST function [Data type conversion]" on page 148
- "CONVERT function [Data type conversion]" on page 160
- "CAST function [Data type conversion]" on page 148

Ambiguous string to date conversions

SQL Anywhere automatically converts a string into a date when a date value is expected, even if the year is represented in the string by only two digits.

If the century portion of a year value is omitted, the method of conversion is determined by the nearest_century database option.

The nearest_century database option is a numeric value that acts as a break point between 19YY date values and 20YY date values.

Two-digit years less than the nearest_century value are converted to 20yy, while years greater than or equal to the value are converted to 19yy.

If this option is not set, the default setting of 50 is assumed. So, two-digit year strings are understood to refer to years between 1950 and 2049.

This nearest_century option was introduced in SQL Anywhere Version 5.5. In version 5.5, the default setting was 0.

Ambiguous date conversion example

The following statement creates a table that can be used to illustrate the conversion of ambiguous date information in SQL Anywhere.

```
CREATE TABLE T1 (C1 DATE);
```

The table T1 contains one column, C1, of the type DATE.

The following statement inserts a date value into the column C1. SQL Anywhere automatically converts a string that contains an ambiguous year value, one with two digits representing the year but nothing to indicate the century.

```
INSERT INTO T1 VALUES('00-01-01');
```

By default, the nearest_century option is set to 50, so SQL Anywhere converts the above string into the date 2000-01-01. The following statement verifies the result of this insert.

```
SELECT * FROM T1;
```

Changing the nearest_century option using the following statement alters the conversion process.

```
SET OPTION nearest_century = 0;
```

When nearest_century option is set to 0, executing the previous insert using the same statement will create a different date value:

```
INSERT INTO T1 VALUES('00-01-01');
```

The above statement now results in the insertion of the date 1900-01-01. Use the following statement to verify the results.

```
SELECT * FROM T1;
```

Java and SQL data type conversion

Data type conversion between Java types and SQL types is required for both Java stored procedures and JDBC applications. Java to SQL and SQL to Java data type conversions are carried out according to the JDBC standard. The conversions are described in the following tables.

Java to SQL data type conversion

Java type	**SQL type**
String	CHAR
String	VARCHAR
String	TEXT
java.math.BigDecimal	NUMERIC
java.math.BigDecimal	MONEY
java.math.BigDecimal	SMALLMONEY
boolean	BIT
byte	TINYINT
Short	SMALLINT
Int	INTEGER
long	INTEGER
float	REAL
double	DOUBLE
byte[ ]	VARBINARY
byte[ ]	IMAGE
java.sql.Date	DATE
java.sql.Time	TIME
java.sql.Timestamp	TIMESTAMP
java.lang.Double	DOUBLE

Java type	SQL type
java.lang.Float	REAL
java.lang.Integer	INTEGER
java.lang.Long	INTEGER

SQL to Java data type conversion

SQL type	Java type
CHAR	String
VARCHAR	String
TEXT	String
NUMERIC	java.math.BigDecimal
DECIMAL	java.math.BigDecimal
MONEY	java.math.BigDecimal
SMALLMONEY	java.math.BigDecimal
BIT	boolean
TINYINT	byte
SMALLINT	short
INTEGER	int
BIGINT	long
REAL	float
FLOAT	double
DOUBLE	double
BINARY	byte[ ]
VARBINARY	byte[ ]
LONG VARBINARY	byte[ ]

SQL type	Java type
IMAGE	byte[ ]
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp

SQL functions

Contents

Functions are used to return information from the database. They are allowed anywhere an expression is allowed.

Unless otherwise specified in the documentation, NULL is returned for a function if any argument is NULL.

Functions use the same syntax conventions used by SQL statements. For a complete list of syntax conventions, see "Syntax conventions" on page 355.

Function types

This section groups the available function by type.

Aggregate functions

Aggregate functions summarize data over a group of rows from the database. The groups are formed using the GROUP BY clause of the SELECT statement. Aggregate functions are allowed only in the select list and in the HAVING and ORDER BY clauses of a SELECT statement.

List of functions

The following aggregate functions are available:

- "AVG function [Aggregate]" on page 140
- "BIT_AND function [Aggregate]" on page 142
- "BIT_OR function [Aggregate]" on page 143
- "BIT_XOR function [Aggregate]" on page 145
- "COVAR_POP function [Aggregate]" on page 166
- "COVAR_SAMP function [Aggregate]" on page 167
- "COUNT function [Aggregate]" on page 164
- "CORR function [Aggregate]" on page 162
- "FIRST_VALUE function [Aggregate]" on page 206
- "GROUPING function [Aggregate]" on page 213
- "LAST_VALUE function [Aggregate]" on page 233
- "LIST function [Aggregate]" on page 238
- "MAX function [Aggregate]" on page 245
- "MIN function [Aggregate]" on page 246
- "REGR_AVGX function [Aggregate]" on page 276
- "REGR_AVGY function [Aggregate]" on page 277
- "REGR_COUNT function [Aggregate]" on page 279
- "REGR_INTERCEPT function [Aggregate]" on page 280
- "REGR_R2 function [Aggregate]" on page 281
- "REGR_SLOPE function [Aggregate]" on page 282
- "REGR_SXX function [Aggregate]" on page 284
- "REGR_SXY function [Aggregate]" on page 285
- "REGR_SYY function [Aggregate]" on page 286
- "SET_BITS function [Aggregate]" on page 301
- "STDDEV function [Aggregate]" on page 312
- "STDDEV_POP function [Aggregate]" on page 312
- "STDDEV_SAMP function [Aggregate]" on page 313
- "SUM function [Aggregate]" on page 320
- "VAR_POP function [Aggregate]" on page 336
- "VAR_SAMP function [Aggregate]" on page 337
- "VARIANCE function [Aggregate]" on page 338
- "XMLAGG function [Aggregate]" on page 342

Bit array functions

Bit array functions allow you to perform tasks on bit arrays. The following bit array functions are available:

- "BIT_AND function [Aggregate]" on page 142
- "BIT_OR function [Aggregate]" on page 143
- "BIT_XOR function [Aggregate]" on page 145
- "BIT_LENGTH function [Bit array]" on page 143
- "BIT_SUBSTR function [Bit array]" on page 144
- "COUNT_SET_BITS function [Bit array]" on page 165
- "GET_BIT function [Bit array]" on page 208
- "SET_BIT function [Bit array]" on page 300
- "SET_BITS function [Aggregate]" on page 301

For information about bitwise operators, see "Bitwise operators" on page 14.

See also "sa_get_bits system procedure" on page 873.

Ranking functions

Ranking functions let you compute a rank value for each row in a result set based on an ordering specified in the query.

- "CUME_DIST function [Ranking]" on page 170
- "DENSE_RANK function [Ranking]" on page 189
- "PERCENT_RANK function [Ranking]" on page 263
- "RANK function [Ranking]" on page 272

Data type conversion functions

Data type conversion functions are used to convert arguments from one data type to another, or to test whether they can be converted.

List of functions

The following data type conversion functions are available:

- "CAST function [Data type conversion]" on page 148
- "CONVERT function [Data type conversion]" on page 160
- "HEXTOINT function [Data type conversion]" on page 215
- "INTTOHEX function [Data type conversion]" on page 229
- "ISDATE function [Data type conversion]" on page 230
- "ISNUMERIC function [Miscellaneous]" on page 232

Date and time functions

Date and time functions perform operations on date and time data types or return date or time information.

In this chapter, the term **datetime** is used to mean date or time or timestamp. The specific data type DATETIME is indicated as DATETIME.

For more information about datetime data types, see "Date and time data types" on page 98.

List of functions

The following date and time functions are available:

- "DATE function [Date and time]" on page 172
- "DATEADD function [Date and time]" on page 173
- "DATEDIFF function [Date and time]" on page 174
- "DATEFORMAT function [Date and time]" on page 175
- "DATENAME function [Date and time]" on page 176
- "DATEPART function [Date and time]" on page 177
- "DATETIME function [Date and time]" on page 178
- "DAY function [Date and time]" on page 178
- "DAYNAME function [Date and time]" on page 179
- "DAYS function [Date and time]" on page 179
- "DOW function [Date and time]" on page 191
- "GETDATE function [Date and time]" on page 210
- "HOUR function [Date and time]" on page 216
- "HOURS function [Date and time]" on page 217
- "MINUTE function [Date and time]" on page 247
- "MINUTES function [Date and time]" on page 247
- "MONTH function [Date and time]" on page 249
- "MONTHNAME function [Date and time]" on page 250
- "MONTHS function [Date and time]" on page 250
- "NOW function [Date and time]" on page 258
- "QUARTER function [Date and time]" on page 269
- "SECOND function [Date and time]" on page 298
- "SECONDS function [Date and time]" on page 299
- "TODAY function [Date and time]" on page 325
- "WEEKS function [Date and time]" on page 340
- "YEAR function [Date and time]" on page 349
- "YEARS function [Date and time]" on page 349
- "YMD function [Date and time]" on page 351

Date parts

Many of the date functions use dates built from **date parts**. The following table displays allowed values of date parts.

Date part	Abbreviation	Values
Year	yy	1-9999
Quarter	qq	1-4
Month	mm	1-12
Week	wk	1-54. Weeks begin on Sunday.
Day	dd	1-31
Dayofyear	dy	1-366
Weekday	dw	1-7 (Sunday = 1, ..., Saturday = 7)
Hour	hh	0-23
Minute	mi	0-59
Second	ss	0-59
Millisecond	ms	0-999
Calyearofweek	cyr	Integer. The year in which the week begins. The week containing the first few days of the year may have started in the previous year, depending on the weekday on which the year started. Years starting on Monday through Thursday have no days that are part of the previous year, but years starting on Friday through Sunday start their first week on the first Monday of the year.
Calweekofyear	cwk	1-54. The week number within the year that contains the specified date.
Caldayofweek	cdw	1-7. (Monday = 1, ..., Sunday = 7)

User-defined functions

A user-defined function, or UDF, is a function created by the user of a program or environment. User-defined functions are in contrast to functions that are built in to the program or environment.

There are two mechanisms for creating user-defined functions in SQL Anywhere. You can use the SQL language to write the function, or you can use Java.

User-defined functions in SQL

You can implement your own functions in SQL using the “CREATE FUNCTION statement (web services)” on page 467. The RETURN statement inside the CREATE FUNCTION statement determines the data type of the function.

Once a SQL user-defined function is created, it can be used anywhere a built-in function of the same data type is used.

For more information about creating SQL functions, see “Using procedures, triggers, and batches” [*SQL Anywhere Server - SQL Usage*].

User-defined functions in Java

Java classes provide a more powerful and flexible way of implementing user-defined functions, with the additional advantage that they can be moved from the database server to a client application if desired.

Any class method of an installed Java class can be used as a user-defined function anywhere a built-in function of the same data type is used.

Instance methods are tied to particular instances of a class, and so have different behavior from standard user-defined functions.

For more information about creating Java classes, and on class methods, see “Creating a class” [*SQL Anywhere Server - Programming*].

Deciding whether to create a user-defined functions or a procedure

Functions are similar to procedures. Deciding whether to create a function or a procedure depends on what you want returned, and the object will be called. When deciding whether to create a UDF or a procedure, consider their unique characteristics listed below.

Functions:

- can return a single value of arbitrary type, and allow you to declare the returned type using the RETURNS clause
- can be used in most places an expression can be used
- allow you to define only IN parameters

Procedures:

- can return multiple values using INOUT or OUT parameters
- can return result sets
- can be referenced in the FROM clause of a query, or using a CALL statement, or using a Transact-SQL EXECUTE statement
- can be called using named parameters

Miscellaneous functions

Miscellaneous functions perform operations on arithmetic, string, or date/time expressions, including the return values of other functions.

List of functions

The following miscellaneous functions are available:

- "ARGN function [Miscellaneous]" on page 136
- "COALESCE function [Miscellaneous]" on page 152
- "CONFLICT function [Miscellaneous]" on page 158
- "ERRORMSG function [Miscellaneous]" on page 194
- "ESTIMATE function [Miscellaneous]" on page 195
- "ESTIMATE_SOURCE function [Miscellaneous]" on page 196
- "EXPERIENCE_ESTIMATE function [Miscellaneous]" on page 203
- "EXPLANATION function [Miscellaneous]" on page 204
- "EXPRTYPE function [Miscellaneous]" on page 205
- "GET_IDENTITY function [Miscellaneous]" on page 209
- "GRAPHICAL_PLAN function [Miscellaneous]" on page 211
- "GREATER function [Miscellaneous]" on page 212
- "IDENTITY function [Miscellaneous]" on page 227
- "IFNULL function [Miscellaneous]" on page 227
- "INDEX_ESTIMATE function [Miscellaneous]" on page 228
- "ISNULL function [Miscellaneous]" on page 231
- "LESSER function [Miscellaneous]" on page 237
- "NEWID function [Miscellaneous]" on page 252
- "NULLIF function [Miscellaneous]" on page 259
- "NUMBER function [Miscellaneous]" on page 260
- "PLAN function [Miscellaneous]" on page 265
- "REWRITE function [Miscellaneous]" on page 292
- "ROW_NUMBER function [Miscellaneous]" on page 296
- "SQLDIALECT function [Miscellaneous]" on page 309
- "SQLFLAGGER function [Miscellaneous]" on page 310
- "TRACEBACK function [Miscellaneous]" on page 326
- "TRANSACTSQL function [Miscellaneous]" on page 327
- "VAREXISTS function [Miscellaneous]" on page 339
- "WATCOMSQL function [Miscellaneous]" on page 339

Numeric functions

Numeric functions perform mathematical operations on numerical data types or return numeric information.

List of functions

The following numeric functions are available:

- "ABS function [Numeric]" on page 135
- "ACOS function [Numeric]" on page 135
- "ASIN function [Numeric]" on page 138
- "ATAN function [Numeric]" on page 138
- "ATAN2 function [Numeric]" on page 139
- "CEILING function [Numeric]" on page 149
- "COS function [Numeric]" on page 163
- "COT function [Numeric]" on page 164
- "DEGREES function [Numeric]" on page 188
- "EXP function [Numeric]" on page 202
- "FLOOR function [Numeric]" on page 208
- "LOG function [Numeric]" on page 242
- "LOG10 function [Numeric]" on page 242
- "MOD function [Numeric]" on page 248
- "PI function [Numeric]" on page 264
- "POWER function [Numeric]" on page 266
- "RADIANS function [Numeric]" on page 270
- "RAND function [Numeric]" on page 271
- "REMAINDER function [Numeric]" on page 288
- "ROUND function [Numeric]" on page 294
- "SIGN function [Numeric]" on page 302
- "SIN function [Numeric]" on page 303
- "SQRT function [Numeric]" on page 311
- "TAN function [Numeric]" on page 322
- "TRUNCNUM function [Numeric]" on page 328

Web services functions

HTTP functions assist the handling of HTTP requests within web services. Likewise, SOAP functions assist the handling of SOAP requests within web services.

The following functions are available:

- "HTML_DECODE function [Miscellaneous]" on page 218
- "HTML_ENCODE function [Miscellaneous]" on page 219
- "HTTP_BODY function [HTTP]" on page 223
- "HTTP_DECODE function [HTTP]" on page 220
- "HTTP_ENCODE function [HTTP]" on page 221
- "HTTP_HEADER function [HTTP]" on page 223
- "HTTP_VARIABLE function [HTTP]" on page 225
- "NEXT_HTTP_HEADER function [HTTP]" on page 255
- "NEXT_HTTP_VARIABLE function [HTTP]" on page 256
- "NEXT_SOAP_HEADER function [SOAP]" on page 258
- "SOAP_HEADER function [SOAP]" on page 304

There are also many system procedures available for web services. See “Web services system procedures” on page 830.

String functions

String functions perform conversion, extraction, or manipulation operations on strings, or return information about strings.

When working in a multibyte character set, check carefully whether the function being used returns information concerning characters or bytes.

List of functions

The following string functions are available:

- “ASCII function [String]” on page 137
- “BASE64_DECODE function [String]” on page 141
- “BASE64_ENCODE function [String]” on page 141
- “BYTE_LENGTH function [String]” on page 146
- “BYTE_SUBSTR function [String]” on page 147
- “CHAR function [String]” on page 150
- “CHARINDEX function [String]” on page 150
- “CHAR_LENGTH function [String]” on page 151
- “COMPARE function [String]” on page 153
- “COMPRESS function [String]” on page 155
- “CSCONVERT function [String]” on page 168
- “DECOMPRESS function [String]” on page 186
- “DECRYPT function [String]” on page 187
- “DIFFERENCE function [String]” on page 190
- “ENCRYPT function [String]” on page 193
- “HASH function [String]” on page 214
- “INSERTSTR function [String]” on page 229
- “LCASE function [String]” on page 235
- “LEFT function [String]” on page 235
- “LENGTH function [String]” on page 236
- “LOCATE function [String]” on page 240
- “LOWER function [String]” on page 243
- “LTRIM function [String]” on page 244
- “NCHAR function [String]” on page 252
- “PATINDEX function [String]” on page 262
- “READ_CLIENT_FILE function [String]” on page 273
- “REGEXP_SUBSTR function [String]” on page 274
- “REPEAT function [String]” on page 288
- “REPLACE function [String]” on page 289
- “REPLICATE function [String]” on page 290
- “REVERSE function [String]” on page 291
- “RIGHT function [String]” on page 293
- “RTRIM function [String]” on page 297
- “SIMILAR function [String]” on page 303
- “SORTKEY function [String]” on page 305
- “SOUNDEX function [String]” on page 308
- “SPACE function [String]” on page 309
- “STR function [String]” on page 315
- “STRING function [String]” on page 315
- “STRTOUUID function [String]” on page 316
- “STUFF function [String]” on page 317
- “SUBSTRING function [String]” on page 318
- “TO_CHAR function [String]” on page 323
- “TO_NCHAR function [String]” on page 324

- “TRIM function [String]” on page 327
- “UCASE function [String]” on page 330
- “UNICODE function [String]” on page 331
- “UNISTR function [String]” on page 331
- “UPPER function [String]” on page 333
- “UUIDTOSTR function [String]” on page 335
- “XMLCONCAT function [String]” on page 343
- “XMLELEMENT function [String]” on page 344
- “XMLFOREST function [String]” on page 346
- “XMLGEN function [String]” on page 348

System functions

System functions return system information.

List of functions

The following system functions are available:

- “CONNECTION_EXTENDED_PROPERTY function [String]” on page 156
- “CONNECTION_PROPERTY function [System]” on page 157
- “DATALENGTH function [System]” on page 171
- “DB_ID function [System]” on page 184
- “DB_NAME function [System]” on page 184
- “DB_EXTENDED_PROPERTY function [System]” on page 181
- “DB_PROPERTY function [System]” on page 185
- “EVENT_CONDITION function [System]” on page 198
- “EVENT_CONDITION_NAME function [System]” on page 199
- “EVENT_PARAMETER function [System]” on page 200
- “NEXT_CONNECTION function [System]” on page 253
- “NEXT_DATABASE function [System]” on page 255
- “PROPERTY function [System]” on page 266
- “PROPERTY_DESCRIPTION function [System]” on page 267
- “PROPERTY_NAME function [System]” on page 268
- “PROPERTY_NUMBER function [System]” on page 269
- “SUSER_ID function [System]” on page 321
- “SUSER_NAME function [System]” on page 321
- “TSEQUAL function [System] (deprecated)” on page 329
- “USER_ID function [System]” on page 333
- “USER_NAME function [System]” on page 334

Notes

- Some of the system functions are implemented in SQL Anywhere as stored procedures.
- The db_id, db_name, and datalength functions are implemented as built-in functions.

The implemented system functions are described in the following table.

System function	Description
COL_LENGTH(*table-name*, *column-name* **)**	Returns the defined length of column
COL_NAME(*table-id*, *column-id* [, *database-id*] **)**	Returns the column name
INDEX_COL (*table-name*, *index-id*, *key_#* [, *userid*] **)**	Returns the name of the indexed column
OBJECT_ID (*object-name* **)**	Returns the object ID
OBJECT_NAME (*object-id* [, *database-id*] **)**	Returns the object name

Text and image functions

Text and image functions operate on text and image data types. SQL Anywhere supports only the textptr text and image function.

List of functions

The following text and image function is available:

- "TEXTPTR function [Text and image]" on page 323

SQL functions (A-D)

Each function is listed, and the function type (numeric, character, and so on) is indicated next to it.

For links to all functions of a given type, see "Function types" on page 124.

See also

- "SQL functions (E-O)" on page 193
- "SQL functions (P-Z)" on page 262

ABS function [Numeric]

Returns the absolute value of a numeric expression.

Syntax

ABS(*numeric-expression* **)**

Parameters

- **numeric-expression** The number whose absolute value is to be returned.

Returns

An absolute value of the numeric expression.

Numeric-expression data type	Returns
INT	INT
FLOAT	FLOAT
DOUBLE	DOUBLE
NUMERIC	NUMERIC

Standards and compatibility

- **SQL/2003** SQL foundation feature outside core SQL.

Example

The following statement returns the value 66.

```
SELECT ABS( -66 );
```

ACOS function [Numeric]

Returns the arc-cosine, in radians, of a numeric expression.

Syntax

ACOS(*numeric-expression* **)**

Parameters

- **numeric-expression** The cosine of the angle.

Returns

DOUBLE

Remarks

This function converts its argument to DOUBLE, and performs the computation in double-precision floating point.

See also

- "ASIN function [Numeric]" on page 138
- "ATAN function [Numeric]" on page 138
- "ATAN2 function [Numeric]" on page 139
- "COS function [Numeric]" on page 163

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the arc-cosine value for 0.52.

```
SELECT ACOS( 0.52 );
```

ARGN function [Miscellaneous]

Returns a selected argument from a list of arguments.

Syntax

ARGN(*integer-expression*, *expression* [, ...] **)**

Parameters

- **integer-expression** The position of an argument within the list of expressions.

- **expression** An expression of any data type passed into the function. All supplied expressions must be of the same data type.

Returns

Using the value of the *integer-expression* as n, returns the nth argument (starting at 1) from the remaining list of arguments.

Remarks

While the expressions can be of any data type, they must all be of the same data type. The integer expression must be from one to the number of expressions in the list or NULL is returned. Multiple expressions are separated by a comma.

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value 6.

```
SELECT ARGN( 6, 1,2,3,4,5,6 );
```

ASCII function [String]

Returns the integer ASCII value of the first byte in a string-expression.

Syntax

ASCII(*string-expression* **)**

Parameters

- **string-expression** The string.

Returns

SMALLINT

Remarks

If the string is empty, then ASCII returns zero. Literal strings must be enclosed in quotes. If the database character set is multibyte and the first character of the parameter string consists of more than one byte, the result is NULL.

See also

- "String functions" on page 131

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value 90.

```
SELECT ASCII( 'Z' );
```

ASIN function [Numeric]

Returns the arc-sine, in radians, of a number.

Syntax

ASIN(*numeric-expression* **)**

Parameters

- **numeric-expression** The sine of the angle.

Returns

DOUBLE

Remarks

The SIN and ASIN functions are inverse operations.

This function converts its argument to DOUBLE, and performs the computation in double-precision floating point.

See also

- "ACOS function [Numeric]" on page 135
- "ATAN function [Numeric]" on page 138
- "ATAN2 function [Numeric]" on page 139
- "SIN function [Numeric]" on page 303

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the arc-sine value for 0.52.

```
SELECT ASIN( 0.52 );
```

ATAN function [Numeric]

Returns the arc-tangent, in radians, of a number.

Syntax

ATAN(*numeric-expression* **)**

Remarks

The ATAN and TAN functions are inverse operations.

Parameters

- **numeric-expression** The tangent of the angle.

Returns

DOUBLE

Remarks

This function converts its argument to DOUBLE, and performs the computation in double-precision floating point.

See also

- "ACOS function [Numeric]" on page 135
- "ASIN function [Numeric]" on page 138
- "ATAN2 function [Numeric]" on page 139
- "TAN function [Numeric]" on page 322

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the arc-tangent value for 0.52.

```
SELECT ATAN( 0.52 );
```

ATAN2 function [Numeric]

Returns the arc-tangent, in radians, of the ratio of two numbers.

Syntax

{ **ATN2** | **ATAN2** }**(** *numeric-expression-1*, *numeric-expression-2* **)**

Parameters

- **numeric-expression-1** The numerator in the ratio whose arc-tangent is calculated.
- **numeric-expression-2** The denominator in the ratio whose arc-tangent is calculated.

Returns

DOUBLE

Remarks

This function converts its arguments to DOUBLE, and performs the computation in double-precision floating point.

See also

- "ACOS function [Numeric]" on page 135
- "ASIN function [Numeric]" on page 138
- "ATAN function [Numeric]" on page 138
- "TAN function [Numeric]" on page 322

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the arc-tangent value for the ratio 0.52 to 0.60.

```
SELECT ATAN2( 0.52, 0.60 );
```

AVG function [Aggregate]

Computes the average, for a set of rows, of a numeric expression or of a set of unique values.

Syntax 1

AVG(*numeric-expression* | **DISTINCT** *numeric-expression* **)**

Syntax 2

AVG(*numeric-expression* **) OVER (** *window-spec* **)**

window-spec : see Syntax 2 instructions in the Remarks section below

Parameters

- **numeric-expression** The expression whose average is calculated over a set of rows.
- **DISTINCT clause** Computes the average of the unique numeric values in the input.

Returns

Returns the NULL value for a group containing no rows.

Returns DOUBLE if the argument is DOUBLE, otherwise NUMERIC.

Remarks

This average does not include rows where the *numeric-expression* is the NULL value.

Syntax 2 represents usage as a window function in a SELECT statement. As such, elements of *window-spec* can be specified either in the function syntax (inline), or in conjunction with a WINDOW clause in the SELECT statement. See the *window-spec* definition provided in "WINDOW clause" on page 793.

For more information about using window functions in SELECT statements, including working examples, see "Window functions" [*SQL Anywhere Server - SQL Usage*].

See also

- "SUM function [Aggregate]" on page 320
- "COUNT function [Aggregate]" on page 164

Standards and compatibility

- **SQL/2003** Core feature. Syntax 2 is feature T611.

Example

The following statement returns the value 49988.623200.

```
SELECT AVG( Salary ) FROM Employees;
```

The following statement returns the average product price from the Products table:

```
SELECT AVG( DISTINCT UnitPrice ) FROM Products;
```

BASE64_DECODE function [String]

Decodes data using the MIME base64 format and returns the string as a LONG VARCHAR.

Syntax

BASE64_DECODE(*string-expression* **)**

Parameters

- **string-expression** The string that is to be decoded. Note that the string must be base64-encoded.

Returns

LONG VARCHAR

See also

- "BASE64_ENCODE function [String]" on page 141
- "String functions" on page 131

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following inserts an image into an image table from an embedded SQL program. The input data (host variable) must be base64 encoded.

```
EXEC SQL INSERT INTO images ( image_data ) VALUES ( BASE64_DECODE ( :img ) );
```

BASE64_ENCODE function [String]

Encodes data using the MIME base64 format and returns it as a 7-bit ASCII string.

Syntax

BASE64_ENCODE(*string-expression* **)**

Parameters

- **string-expression** The string that is to be encoded.

Returns

LONG VARCHAR

See also

- "BASE64_DECODE function [String]" on page 141
- "String functions" on page 131

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following retrieves data from a table containing images and returns it in ASCII format. The resulting string can be embedded into an email message, and then decoded by the recipient to retrieve the original image.

```
SELECT BASE64_ENCODE( image_data ) FROM IMAGES;
```

BIT_AND function [Aggregate]

Takes *n* bit arrays and returns a bitwise AND-ing of its arguments using the following logic: for each bit compared, if all bits are 1, return 1; otherwise, return 0.

Syntax

BIT_AND(*bit-expression* **)**

Parameters

- **expression** The expression for which the value is to be determined. This is commonly a column name.

Returns

LONG VARBIT

See also

- "BIT_OR function [Aggregate]" on page 143
- "BIT_XOR function [Aggregate]" on page 145
- "Bitwise operators" on page 14

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following example generates four rows containing a CHAR column, then converts the values to VARBIT.

```
SELECT BIT_AND( CAST(row_value AS VARBIT) )
FROM dbo.sa_split_list('0001,0111,0100,0011')
```

The result 0000 is determined as follows:

1. A bitwise AND is performed between row 1 (0001) and row 2 (0111), resulting in 0001 (both values had a 1 in the fourth bit).
2. A bitwise AND is performed between the result from the previous comparison (0001) and row 3 (0100), resulting in 0000 (neither value had a 1 in the same bit).
3. A bitwise AND is performed between the result from the previous comparison (0000) and row 4 (0011), resulting in 0000 (neither value had a 1 in the same bit).

BIT_LENGTH function [Bit array]

Returns the number of bits stored in the array.

Syntax

BIT_LENGTH(*bit-expression* **)**

Parameters

- **bit-expression** The bit expression for which the length is to be determined.

Returns

INT

See also

- "BIT_LENGTH function [Bit array]" on page 143
- "CHAR_LENGTH function [String]" on page 151

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value 8:

```
SELECT BIT_LENGTH( '01101011' );
```

BIT_OR function [Aggregate]

Takes *n* bit arrays and returns a bitwise OR-ing of its arguments using the following logic: for each bit compared, if any bit is 1, return 1; otherwise, return 0.

Syntax

BIT_OR(*bit-expression* **)**

Parameters

- **expression** The expression for which the value is to be determined. This is commonly a column name.

Returns

LONG VARBIT

See also

- "BIT_AND function [Aggregate]" on page 142
- "BIT_XOR function [Aggregate]" on page 145
- "Bitwise operators" on page 14

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following example generates four rows containing a CHAR column, then converts the values to VARBIT.

```
SELECT BIT_OR( CAST(row_value AS VARBIT) )
FROM dbo.sa_split_list('0001,0111,0100,0011')
```

The result 0111 is determined as follows:

1. A bitwise OR is performed between row 1 (0001) and row 2 (0111), resulting in 0111.
2. A bitwise OR is performed between the result from the previous comparison (0111) and row 3 (0100), resulting in 0111.
3. A bitwise OR is performed between the result from the previous comparison (0111) and row 4 (0011), resulting in 0111.

BIT_SUBSTR function [Bit array]

Returns a sub-array of a bit array.

Syntax

BIT_SUBSTR(*bit-expression* [**,** *start* [**,** *length*]] **)**

Parameters

- **bit-expression** The bit array from which the sub-array is to be extracted.
- **start** The start position of the sub-array to return. A negative starting position specifies the number of bits from the end of the array instead of the beginning. The first bit in the array is at position 1.
- **length** The length of the sub-array to return. A positive length specifies that the sub-array ends *length* bits to the right of the starting position, while a negative length returns, at most, *length* bits up to, and including, the starting position, from the left of the starting position.

Returns

LONG VARBIT

Remarks

Both *start* and *length* can be either positive or negative. Using appropriate combinations of negative and positive numbers, you can get a sub-array from either the beginning or end of the string. Using a negative number for *length* does not impact the order of the bits returned in the sub-array.

If *length* is specified, the sub-array is restricted to that length. If start is zero and length is non-negative, a start value of 1 is used. If start is zero and length is negative, a start value of -1 is used.

If *length* is not specified, selection continues to the end of the array.

The BIT_SUBSTR function is equivalent to, but faster than, the following:

```
CAST( SUBSTR( CAST( bit-expression AS VARCHAR ),
start [, length ] )
AS VARBIT )
```

See also

- "SUBSTRING function [String]" on page 318

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns 1101:

```
SELECT BIT_SUBSTR( '001101', 3 );
```

The following statement returns 10110:

```
SELECT BIT_SUBSTR( '01011011101111011111', 2, 5 );
```

The following statement returns 11111:

```
SELECT BIT_SUBSTR( '01011011101111011111', -5, 5 );
```

BIT_XOR function [Aggregate]

Takes *n* bit arrays and returns a bitwise exclusive OR-ing of its arguments using the following logic: for each bit compared, if there are an odd number of arguments with set bits (odd parity), return 1; otherwise, return 0.

Syntax

BIT_XOR(*bit-expression* **)**

Parameters

- **bit-expression** The expression for which the value is to be determined. This is commonly a column name.

Returns

LONG VARBIT

See also

- "BIT_AND function [Aggregate]" on page 142
- "BIT_OR function [Aggregate]" on page 143
- "Bitwise operators" on page 14

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following example generates four rows containing a CHAR column, then converts the values to VARBIT.

```
SELECT BIT_XOR( CAST(row_value AS VARBIT) )
FROM dbo.sa_split_list('0001,0111,0100,0011')
```

The result 0001 is determined as follows:

1. A bitwise exclusive OR (XOR) is performed between row 1 (0001) and row 2 (0111), resulting in 0110.
2. A bitwise XOR is performed between the result from the previous comparison (0110) and row 3 (0100), resulting in 0010.
3. A bitwise XOR is performed between the result from the previous comparison (0010) and row 4 (0011), resulting in 0001.

BYTE_LENGTH function [String]

Returns the number of bytes in a string.

Syntax

BYTE_LENGTH(*string-expression* **)**

Parameters

- **string-expression** The string whose length is to be calculated.

Returns

INT

Remarks

Trailing white space characters in the *string-expression* are included in the length returned.

The return value of a NULL string is NULL.

If the string is in a multibyte character set, the BYTE_LENGTH value may differ from the number of characters returned by CHAR_LENGTH.

This function supports NCHAR inputs and/or outputs.

See also

- "CHAR_LENGTH function [String]" on page 151
- "DATALENGTH function [System]" on page 171
- "LENGTH function [String]" on page 236
- "String functions" on page 131

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value 12.

```
SELECT BYTE_LENGTH( 'Test Message' );
```

BYTE_SUBSTR function [String]

Returns a substring of a string. The substring is calculated using bytes, not characters.

Syntax

BYTE_SUBSTR(*string-expression*, *start* [, *length*] **)**

Parameters

- **string-expression** The string from which the substring is taken.
- **start** An integer expression indicating the start of the substring. A positive integer starts from the beginning of the string, with the first character being position 1. A negative integer specifies a substring starting from the end of the string, the final character being at position -1.
- **length** An integer expression indicating the length of the substring. A positive *length* specifies the number of bytes to be taken *starting* at the start position. A negative *length* returns at most *length* bytes up to, and including, the starting position, from the left of the starting position.

Returns

The value returned depends on the type of *string-expression*. Also, the arguments you specify determine if the returned value is LONG. For example, LONG is not returned when you specify a constant < 32K for length.

BINARY

VARCHAR

NVARCHAR

Remarks

If *length* is specified, the substring is restricted to that number of bytes. Both *start* and *length* can be either positive or negative. Using appropriate combinations of negative and positive numbers, you can get a substring from either the beginning or end of the string.

If *start* is zero and length is non-negative, a *start* value of 1 is used. If *start* is zero and *length* is negative, a start value of -1 is used.

See also

- "SUBSTRING function [String]" on page 318
- "String functions" on page 131

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value Test.

```
SELECT BYTE_SUBSTR( 'Test Message', 1, 4 );
```

CAST function [Data type conversion]

Returns the value of an expression converted to a supplied data type.

The CAST, CONVERT, HEXTOINT, and INTTOHEX functions can be used to convert to and from hexadecimal values. For more information on using these functions, see "Converting to and from hexadecimal values" on page 10.

Syntax

CAST(*expression* **AS** *datatype* **)**

Parameters

- **expression** The expression to be converted.
- **data type** The target data type.

Returns

The specified data type.

Remarks

If you do not indicate a length for character string types, the database server chooses an appropriate length. If neither precision nor scale is specified for a DECIMAL conversion, the database server selects appropriate values.

If you use the CAST function to truncate strings, the string_rtruncation database option must be set to OFF; otherwise, there will be an error. It is recommended that you use the LEFT function to truncate strings.

See also

- “CONVERT function [Data type conversion]” on page 160
- “LEFT function [String]” on page 235

Standards and compatibility

- **SQL/2003** Core feature.

Example

The following function ensures a string is used as a date:

```
SELECT CAST( '2000-10-31' AS DATE );
```

The value of the expression `1 + 2` is calculated, and the result is then cast into a single-character string.

```
SELECT CAST( 1 + 2 AS CHAR );
```

CEILING function [Numeric]

Returns the first integer that is greater or equal to a given value. For positive numbers, this is known as rounding up.

Syntax

CEILING(*numeric-expression* **)**

Parameters

- **numeric-expression** The number whose ceiling is to be calculated.

Returns

DOUBLE

Remarks

This function converts its argument to DOUBLE, and performs the computation in double-precision floating point.

See also

- “FLOOR function [Numeric]” on page 208

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value 60.

```
SELECT CEILING( 59.84567 );
```

CHAR function [String]

Returns the character with the ASCII value of a number.

Syntax

CHAR(*integer-expression* **)**

Parameters

- **integer-expression** The number to be converted to an ASCII character. The number must be in the range 0 to 255, inclusive.

Returns

VARCHAR

Remarks

The character returned corresponds to the supplied numeric expression in the current database character set, according to a binary sort order.

CHAR returns NULL for integer expressions with values greater than 255 or less than zero.

See also

- "String functions" on page 131

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value Y.

```
SELECT CHAR( 89 );
```

CHARINDEX function [String]

Returns the position of one string in another.

Syntax

CHARINDEX(*string-expression-1*, *string-expression-2* **)**

Parameters

- **string-expression-1** The string for which you are searching.
- **string-expression-2** The string to be searched.

Returns

INT

Remarks

The first character of *string-expression-1* is identified as 1. If the string being searched contains more than one instance of the other string, then the CHARINDEX function returns the position of the first instance.

If the string being searched does not contain the other string, then the CHARINDEX function returns 0.

This function supports NCHAR inputs and/or outputs.

See also

- "SUBSTRING function [String]" on page 318
- "REPLACE function [String]" on page 289
- "LOCATE function [String]" on page 240
- "String functions" on page 131

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns last and first names from the Surname and GivenName columns of the Employees table, but only when the last name includes the letter K:

```
SELECT Surname, GivenName
FROM Employees
WHERE CHARINDEX( 'K', Surname ) = 1;
```

Results returned:

Surname	GivenName
Klobucher	James
Kuo	Felicia
Kelly	Moira

CHAR_LENGTH function [String]

Returns the number of characters in a string.

Syntax

CHAR_LENGTH (*string-expression* **)**

Parameters

- **string-expression** The string whose length is to be calculated.

Returns

INT

Remarks

Trailing white space characters are included in the length returned.

The return value of a NULL string is NULL.

If the string is in a multibyte character set, the value returned by the CHAR_LENGTH function may differ from the number of bytes returned by the BYTE_LENGTH function.

> **Note**
> You can use the CHAR_LENGTH function and the LENGTH function interchangeably for CHAR, VARCHAR, LONG VARCHAR, and NCHAR data types. However, you must use the LENGTH function for BINARY and bit array data types.

This function supports NCHAR inputs and/or outputs.

See also

- "BYTE_LENGTH function [String]" on page 146
- "String functions" on page 131

Standards and compatibility

- **SQL/2003** Core feature.

Example

The following statement returns the value 8.

```
SELECT CHAR_LENGTH( 'Chemical' );
```

COALESCE function [Miscellaneous]

Returns the first non-NULL expression from a list. This function is identical to the ISNULL function.

Syntax

COALESCE(*expression*, *expression* [, ...] **)**

Parameters

- **expression** Any expression.

  At least two expressions must be passed into the function, and all expressions must be comparable.

Returns

ANY

Remarks

The result is NULL only if all the arguments are NULL.

The parameters can be of any scalar type, but not necessarily same type.

For a more detailed description of how the database server processes this function, see "ISNULL function [Miscellaneous]" on page 231.

See also

- "ISNULL function [Miscellaneous]" on page 231

Standards and compatibility

- **SQL/2003** Core feature.

Example

The following statement returns the value 34.

```
SELECT COALESCE( NULL, 34, 13, 0 );
```

COMPARE function [String]

Allows you to compare two character strings based on alternate collation rules.

Syntax

COMPARE(
string-expression-1,
string-expression-2
[**,** { *collation-id*
| *collation-name*[**(***collation-tailoring-string***)**] }]
)

Parameters

- **string-expression-1** The first string expression.
- **string-expression-2** The second string expression.

  The string expression can only contain characters that are encoded in the database's character set.

- **collation-id** A variable or integer constant that specifies the sort order to use. You can only use a *collation-id* for built-in collations. See "SORTKEY function [String]" on page 305.

  If you do not specify a collation name or ID, the default is Default Unicode multilingual.

- **collation-name** A string or a character variable that specifies the name of the collation to use. You can also specify char_collation or db_collation (for example, `COMPARE( 'abc', 'ABC', 'char_collation' );`) to use the database's CHAR collation. Similarly, you can specify nchar_collation to use the database's NCHAR collation. For a list of valid collation names, see "SORTKEY function [String]" on page 305.
- **collation-tailoring-string** Optionally, you can specify collation tailoring options (*collation-tailoring-string*) for additional control over the character comparison. These options take the form of keyword=value pairs in parentheses, following the collation name. For example, `'UCA(locale=es;case=LowerFirst;accent=respect)'`. The syntax for specifying these

options is identical to the syntax defined for the COLLATION clause of the CREATE DATABASE statement. See "Collation tailoring options" [*SQL Anywhere Server - Database Administration*].

> **Note**
> All the collation tailoring options are supported when specifying the UCA collation. For all other collations, only case sensitivity tailoring option is supported.

Returns

An INTEGER, based on the collation rules that you choose:

Value	Meaning
1	*string-expression-1* is greater than *string-expression-2*
0	*string-expression-1* is equal to *string-expression-2*
-1	*string-expression-1* is less than *string-expression-2*

Remarks

The COMPARE function does not equate empty strings and strings containing only spaces, even if the database has blank-padding enabled. The COMPARE function uses the SORTKEY function to generate collation keys for comparison. Therefore, an empty string, a string with one space, and a string with two spaces do not compare equally.

If either *string-expression-1* or *string-expression-2* is NULL, the result is NULL.

See also

- "SORTKEY function [String]" on page 305
- "String functions" on page 131

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following example performs three comparisons using the COMPARE function:

```
SELECT COMPARE( 'abc','ABC','UCA(case=LowerFirst)' ),
       COMPARE( 'abc','ABC','UCA(case=Ignore)' ),
       COMPARE( 'abc','ABC','UCA(case=UpperFirst)' );
```

The values returned are -1, 0, 1, indicating the result of each comparison. The first comparison results in -1, indicating that *string-expression-2* ('ABC') is less than *string-expresssion-1* ('abc'). This is because case sensitivity is set to LowerFirst in the first COMPARE statement.

COMPRESS function [String]

Compresses the string and returns a value of type LONG BINARY.

Syntax

COMPRESS(*string-expression* [**,** **'***compression-algorithm-alias***'**]**)**

Parameters

- **string-expression** The string to be compressed. Binary values can be passed to this function. This parameter is case sensitive, even in case-insensitive databases.
- **compression-algorithm-alias** Alias for the algorithm to use for compression. The supported values are zip and gzip (both are based on the same algorithm, but use different headers and trailers).

  Zip is a widely supported compression algorithm. Gzip is compatible with the gzip utility on Unix, whereas the zip algorithm is not.

  Decompression must be performed with the same algorithm.

  For more information, see "DECOMPRESS function [String]" on page 186.

Returns

LONG BINARY

Remarks

The value returned by the COMPRESS is not human-readable. If the value returned is longer than the original string, its maximum size will not be larger than a 0.1% increase over the original string + 12 bytes. You can decompress a compressed *string-expression* using the DECOMPRESS function.

If you are storing compressed values in a table, the column should be BINARY or LONG BINARY so that character set conversion is not performed on the data.

See also

- "DECOMPRESS function [String]" on page 186
- "String functions" on page 131

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following example returns the length of the binary string created by compressing the string 'Hello World' using the gzip algorithm. This example can be useful when you want to determine whether a value has a shorter length when compressed.

```
SELECT LENGTH( COMPRESS( 'Hello world', 'gzip' ) );
```

CONNECTION_EXTENDED_PROPERTY function [String]

Returns the value of the given property. Allows an optional property-specific string parameter to be specified.

Syntax

CONNECTION_EXTENDED_PROPERTY(
{ *property-id* | *property-name* }
[**,** *property-specific-argument* [**,** *connection-id*]]
)

Parameters

- **property-id** The connection property ID.
- **property-name** The connection property name. Possible property names are CharSet and NcharCharSet.
- **property-specific-argument** Optional property-specific string parameter associated with the following connection properties.
    - **CharSet** Returns the CHAR character set label for the connection as it is known by the specified standard. The possible values include: ASE, IANA, MIME, JAVA, WINDOWS, UTR22, IBM, and ICU. The default is IANA unless the database connection was made through TDS in which case ASE is the default.
    - **NcharCharSet** Returns the NCHAR character set label for the connection as it is known by the specified standard. The possible values are the same as listed above for CharSet.
- **connection-id** The connection ID number of a database connection. The ID number for the current connection is used if a value is not specified.

Returns

Returns extended connection properties. The returned value is a VARCHAR.

Remarks

The CONNECTION_EXTENDED_PROPERTY function is similar to the CONNECTION_PROPERTY function except that it allows an optional property-specific string parameter to be specified. The interpretation of the property-specific argument depends on the property ID or name specified in the first argument.

You can use the CONNECTION_EXTENDED_PROPERTY function to return the value for any connection property. However, extended information is only available for the extended properties.

See also

- "Connection properties" [*SQL Anywhere Server - Database Administration*]
- "CONNECTION_PROPERTY function [System]" on page 157
- "DB_EXTENDED_PROPERTY function [System]" on page 181
- "DB_PROPERTY function [System]" on page 185

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following example returns the CHAR character set of the current connection as it is known by the Java standard:

```
SELECT CONNECTION_EXTENDED_PROPERTY( 'charset', 'Java' );
```

CONNECTION_PROPERTY function [System]

Returns the value of a given connection property as a string.

Syntax

CONNECTION_PROPERTY(
{ *integer-expression-1* | *string-expression* }
[, *integer-expression-2*] **)**

Parameters

- **integer-expression-1** In most cases it is more convenient to supply a string expression as the first argument. If you do supply an integer-expression, it is the connection property ID. You can determine this using the PROPERTY_NUMBER function.
- **string-expression** The connection property Name. Either the property ID or the property name must be specified.

  For a list of connection properties, see "Connection properties" [*SQL Anywhere Server - Database Administration*].
- **integer-expression-2** The connection ID of the current database connection. The current connection is used if this argument is omitted.

Returns

VARCHAR

Remarks

The current connection is used if the second argument is omitted.

See also

- "Connection properties" [*SQL Anywhere Server - Database Administration*]
- "PROPERTY_NUMBER function [System]" on page 269

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the number of prepared statements being maintained.

```
SELECT CONNECTION_PROPERTY( 'PrepStmt' );
```

CONFLICT function [Miscellaneous]

Indicates if a column is a source of conflict for an UPDATE being performed against a consolidated database in a SQL Remote environment.

Syntax

CONFLICT(*column-name* **)**

Parameters

- **column-name**   The name of the column being tested for conflicts.

Returns

Returns TRUE if the column appears in the VERIFY list of an UPDATE statement executed by the SQL Remote Message Agent and if the value provided in the VALUES list of that statement does not match the original value of the column in the row being updated. Otherwise, returns FALSE.

See also

- "CREATE TRIGGER statement" on page 538
- "Default resolution for update conflicts" [*SQL Remote*]

Standards and compatibility

- **SQL/2003**   Vendor extension.

Example

The CONFLICT function is intended for use in SQL Remote RESOLVE UPDATE triggers to avoid error messages. To illustrate the use of the CONFLICT function, consider the following table:

```
CREATE TABLE Admin (
  PKey bigint NOT NULL DEFAULT GLOBAL AUTOINCREMENT,
  TextCol CHAR(20) NULL, PRIMARY KEY ( PKey ) );
```

Assume that consolidated and remote databases both have the following row in the Admin table:

```
1, 'Initial'
```

Now, at the consolidated database, update the row as follows:

```
UPDATE Admin SET TextCol = 'Consolidated Update' WHERE PKey = 1;
```

At the remote database, update the row to a different value as follows:

```
UPDATE Admin SET TextCol = 'Remote Update' WHERE PKey = 1;
```

Next, run dbremote on the remote database. It generates a message file with the following statements in it, to be executed at the consolidated database:

```
UPDATE Admin SET TextCol='Remote Update'
VERIFY ( TextCol )
VALUES ( 'Initial' )
WHERE PKey=1;
```

When the SQL Remote Message Agent runs at the consolidated database and applies this UPDATE statement, SQL Anywhere uses the VERIFY and VALUES clause to determine whether a RESOLVE UPDATE trigger will fire. A RESOLVE UPDATE trigger fires only when the update is executed from the SQL Remote Message Agent against a consolidated database. Here is a RESOLVE UPDATE trigger:

```
CREATE TRIGGER ResolveUpdateAdmin
RESOLVE UPDATE ON Admin
REFERENCING OLD AS OldConsolidated
    NEW AS NewRemote
    REMOTE as OldRemote
FOR EACH ROW BEGIN
  MESSAGE 'OLD';
  MESSAGE OldConsolidated.PKey || ',' || OldConsolidated.TextCol;
  MESSAGE 'NEW';
  MESSAGE NewRemote.PKey || ',' || NewRemote.TextCol;
  MESSAGE 'REMOTE';
  MESSAGE OldRemote.PKey || ',' || OldRemote.TextCol;
END;
```

The RESOLVE UPDATE trigger fires because the current value of the TextCol column at the consolidated database (`'Consolidated Update'`) does not match the value in the VALUES clause for the associated column (`'Initial'`).

This trigger results in a failure because the PKey column was not modified in the UPDATE statement executed on the remote, so there is no OldRemote.PKey value accessible from this trigger.

The CONFLICT function helps to avoid this error by returning the following values:

- If there is no OldRemote.PKey value, return FALSE.
- If there is an OldRemote.PKey value, but it matches OldConsolidated.PKey, return FALSE.
- If there is an OldRemote.PKey value, and it is different than OldConsolidated.PKey, return TRUE.

You can use the CONFLICT function to rewrite the trigger as follows and avoid the error:

```
CREATE TRIGGER ResolveUpdateAdmin
RESOLVE UPDATE ON Admin
REFERENCING OLD AS OldConsolidated
    NEW AS NewRemote
    REMOTE as OldRemote
FOR EACH ROW BEGIN
  message 'OLD';
  message OldConsolidated.PKey || ',' || OldConsolidated.TextCol;
  message 'NEW';
  message NewRemote.PKey || ',' || NewRemote.TextCol;
  message 'REMOTE';
  if CONFLICT( PKey ) then
    message OldRemote.PKey;
  end if;
  if CONFLICT( TextCol ) then
    message OldRemote.TextCol;
  end if;
END;
```

CONVERT function [Data type conversion]

Returns an expression converted to a supplied data type.

The CAST, CONVERT, HEXTOINT, and INTTOHEX functions can be used to convert to and from hexadecimal values. For more information on using these functions, see "Converting to and from hexadecimal values" on page 10.

Syntax

CONVERT(*datatype*, *expression* [, *format-style*] **)**

Parameters

- **datatype** The data type to which the expression is converted.
- **expression** The expression to be converted.
- **format-style** The style code to apply to the outputted value. Use this parameter when converting strings to date or time data types, and vice versa. The table below shows the supported style codes, followed by a representation of the output format produced by that style code. The style codes are separated into two columns, depending on whether the century is included in the output format (for example, 06 versus 2006).

Without century (yy) style codes	With century (yyyy) style codes	Output format
-	0 or 100	Mmm dd yyyy hh:nnAA
1	101	mm/dd/yy[yy]
2	102	[yy]yy.mm.dd
3	103	dd/mm/yy[yy]
4	104	dd.mm.yy[yy]
5	105	dd-mm-yy[yy]
6	106	dd Mmm yy[yy]
7	107	Mmm dd, yy[yy]
8	108	hh:nn:ss
-	9 or 109	Mmm dd yyyy hh:nn:ss:sssAA
10	110	mm-dd-yy[yy]
11	111	[yy]yy/mm/dd
12	112	[yy]yymmdd

Without century (yy) style codes	With century (yyyy) style codes	Output format
-	13 or 113	dd Mmm yyyy hh:nn:ss:sss (24 hour clock, Europe default + milliseconds, 4-digit year)
-	14 or 114	hh:nn:ss:sss (24 hour clock)
-	20 or 120	yyyy-mm-dd hh:nn:ss (24-hour clock, ODBC canonical, 4-digit year)
-	21 or 121	yyyy-mm-dd hh:nn:ss.sss (24 hour clock, ODBC canonical with milliseconds, 4-digit year)

Returns

The data type specified.

Remarks

If no *format-style* argument is provided, style code 0 is used.

For a description of the styles produced by each output symbol (such as Mmm), see "date_format option [database]" [*SQL Anywhere Server - Database Administration*].

See also

- "CAST function [Data type conversion]" on page 148

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statements illustrate the use of format style.

```
SELECT CONVERT( CHAR( 20 ), OrderDate, 104 ) FROM SalesOrders;
```

OrderDate
16.03.2000
20.03.2000
23.03.2000
25.03.2000
...

```
SELECT CONVERT( CHAR( 20 ), OrderDate, 7 ) FROM SalesOrders;
```

OrderDate
Mar 16, 00
Mar 20, 00
Mar 23, 00
Mar 25, 00
...

The following statement illustrates conversion to an integer, and returns the value 5.

```
SELECT CONVERT( integer, 5.2 );
```

CORR function [Aggregate]

Returns the correlation coefficient of a set of number pairs.

Syntax

CORR(*dependent-expression*, *independent-expression* **)**

Parameters

- **dependent-expression** The variable that is affected by the independent variable.
- **independent-expression** The variable that influences the outcome.

Returns

DOUBLE

Remarks

This function converts its arguments to DOUBLE, and performs the computation in double-precision floating point. If the function is applied to an empty set, then it returns NULL.

Both *dependent-expression* and *independent-expression* are numeric. The function is applied to the set of (*dependent-expression*, *independent-expression*) after eliminating the pairs for which either *dependent-expression* or *independent-expression* is NULL. The following computation is made:

COVAR_POP (*y*, *x*) / **STDDEV_POP** (*y*) * **STDDEV_POP** (*x*)

where *y* represents the *dependent-expression* and *x* represents the *independent-expression*.

See also

- “Aggregate functions” on page 124
- “COVAR_POP function [Aggregate]” on page 166
- “STDDEV_POP function [Aggregate]” on page 312

Standards and compatibility

- **SQL/2003** SQL foundation feature outside core SQL.

Example

The following example performs a correlation to discover whether age is associated with income level. This function returns the value 0.440226756459957.

```
SELECT CORR( Salary, ( YEAR( NOW( ) ) - YEAR( BirthDate ) ) ) FROM Employees;
```

COS function [Numeric]

Returns the cosine of the angle in radians given by its argument.

Syntax

COS(*numeric-expression* **)**

Parameters

- **numeric-expression** The angle, in radians.

Returns

This function converts its argument to DOUBLE, performs the computation in double-precision floating point, and returns a DOUBLE as the result. If the parameter is NULL, the result is NULL.

See also

- “ACOS function [Numeric]” on page 135
- “COT function [Numeric]” on page 164
- “SIN function [Numeric]” on page 303
- “TAN function [Numeric]” on page 322

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value of the cosine of an angle 0.52 radians.

```
SELECT COS( 0.52 );
```

COT function [Numeric]

Returns the cotangent of the angle in radians given by its argument.

Syntax

COT(*numeric-expression* **)**

Parameters

- **numeric-expression** The angle, in radians.

Returns

This function converts its argument to DOUBLE, performs the computation in double-precision floating point, and returns a DOUBLE as the result. If the parameter is NULL, the result is NULL.

See also

- "COS function [Numeric]" on page 163
- "SIN function [Numeric]" on page 303
- "TAN function [Numeric]" on page 322

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the cotangent value of 0.52.

```
SELECT COT( 0.52 );
```

COUNT function [Aggregate]

Counts the number of rows in a group depending on the specified parameters.

Syntax 1

COUNT(
*
| *expression*
| **DISTINCT** *expression*
)

Syntax 2

COUNT(
{ * | *expression* }
) OVER (*window-spec* **)**

window-spec : see Syntax 2 instructions in the Remarks section below

Parameters

- ***** Return the number of rows in each group.
- **expression** The expression for which to return the number of rows.
- **DISTINCT expression** The expression for which to return the number of distinct rows.

Returns

INT

Remarks

Rows where the value is the NULL value are not included in the count.

Syntax 2 represents usage as a window function in a SELECT statement. As such, elements of *window-spec* can be specified either in the function syntax (inline), or in conjunction with a WINDOW clause in the SELECT statement. See the *window-spec* definition provided in "WINDOW clause" on page 793.

For more information about using window functions in SELECT statements, including working examples, see "Window functions" [*SQL Anywhere Server - SQL Usage*].

See also

- "AVG function [Aggregate]" on page 140
- "SUM function [Aggregate]" on page 320

Standards and compatibility

- **SQL/2003** Core feature. Syntax 2 is feature T611.

Example

The following statement returns each unique city, and the number of rows with that city value.

```
SELECT City, COUNT( * ) FROM Employees GROUP BY City;
```

COUNT_SET_BITS function [Bit array]

Returns a count of the number of bits set to 1 (TRUE) in the array.

Syntax

COUNT_SET_BITS(*bit-expression* **)**

Parameters

- **bit-expression** The bit array for which to determine the set bits.

Returns

UNSIGNED INT

Remarks

Returns NULL if *bit-expression* is NULL.

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value 4:

```
SELECT COUNT_SET_BITS( '00110011' );
```

The following statement returns the value 12:

```
SELECT COUNT_SET_BITS( '0011001111111111' );
```

COVAR_POP function [Aggregate]

Returns the population covariance of a set of number pairs.

Syntax 1

COVAR_POP(*dependent-expression*, *independent-expression* **)**

Syntax 2

COVAR_POP(*dependent-expression*, *independent-expression* **)**
OVER (*window-spec* **)**

window-spec : see Syntax 2 instructions in the Remarks section below

Parameters

- **dependent-expression** The variable that is affected by the independent variable.
- **independent-expression** The variable that influences the outcome.

Returns

DOUBLE

Remarks

This function converts its arguments to DOUBLE, and performs the computation in double-precision floating point. If the function is applied to an empty set, then it returns NULL.

Both *dependent-expression* and *independent-expression* are numeric. The function is applied to the set of (*dependent-expression*, *independent-expression*) pairs after eliminating all pairs for which either *dependent-expression* or *independent-expression* is NULL. The following computation is then made:

```
( SUM( y * x ) - SUM( x ) * SUM( y ) / n ) / n
```

where *y* represents the *dependent-expression* and *x* represents the *independent-expression*.

For more information about the statistical computation performed, see “Mathematical formulas for the aggregate functions” [*SQL Anywhere Server - SQL Usage*].

Syntax 2 represents usage as a window function in a SELECT statement. As such, elements of *window-spec* can be specified either in the function syntax (inline), or in conjunction with a WINDOW clause in the SELECT statement. See the *window-spec* definition provided in “WINDOW clause” on page 793.

For more information about using window functions in SELECT statements, including working examples, see “Window functions” [*SQL Anywhere Server - SQL Usage*].

See also

- “COVAR_SAMP function [Aggregate]” on page 167
- “SUM function [Aggregate]” on page 320

Standards and compatibility

- **SQL/2003** SQL foundation feature (T621) outside core SQL.

Example

The following example measures the strength of association between employees' age and salary. This function returns the value 73785.84005866646.

```
SELECT COVAR_POP( Salary, ( YEAR( NOW( ) ) - YEAR( BirthDate ) ) )
FROM Employees;
```

COVAR_SAMP function [Aggregate]

Returns the sample covariance of a set of number pairs.

Syntax 1

COVAR_SAMP(*dependent-expression*, *independent-expression* **)**

Syntax 2

COVAR_SAMP(*dependent-expression*, *independent-expression* **)**
OVER (*window-spec* **)**

window-spec : see Syntax 2 instructions in the Remarks section below

Parameters

- **dependent-expression** The variable that is affected by the independent variable.
- **independent-expression** The variable that influences the outcome.

Returns

DOUBLE

Remarks

This function converts its arguments to DOUBLE, and performs the computation in double-precision floating point. If the function is applied to an empty set, then it returns NULL.

Both *dependent-expression* and *independent-expression* are numeric. The function is applied to the set of (*dependent-expression*, *independent-expression*) pairs after eliminating all pairs for which either *dependent-expression* or *independent-expression* is NULL.

For more information about the statistical computation performed, see "Mathematical formulas for the aggregate functions" [*SQL Anywhere Server - SQL Usage*].

Syntax 2 represents usage as a window function in a SELECT statement. As such, elements of *window-spec* can be specified either in the function syntax (inline), or in conjunction with a WINDOW clause in the SELECT statement. See the *window-spec* definition provided in "WINDOW clause" on page 793.

For more information about using window functions in SELECT statements, including working examples, see "Window functions" [*SQL Anywhere Server - SQL Usage*].

See also

- "COVAR_POP function [Aggregate]" on page 166
- "SUM function [Aggregate]" on page 320

Standards and compatibility

- **SQL/2003** SQL foundation feature (T621) outside core SQL.

Example

The following example returns the value 74782.9460054052.

```
SELECT COVAR_SAMP( Salary, ( 2008 - YEAR( BirthDate ) ) )
FROM Employees;
```

CSCONVERT function [String]

Converts strings between character sets.

Syntax

CSCONVERT(
string-expression,
target-charset-string
[, *source-charset-string*] [, *options*]]**)**

Parameters

- **string-expression** The string.
- **target-charset-string** The destination character set. *target-charset-string* can be one of the following:
  - **os_charset** Alias for the character set used by the operating system hosting the database server.
  - **char_charset** Alias for the CHAR character set used by the database.

- ○ **nchar_charset** Alias for the NCHAR character set used by the database.
- ○ **any other supported character set label** You can specify any of the SQL Anywhere supported character set labels.
- ○ **source-charset** The character set used by the original *string-expression*. The default is db_charset (the database character set). *source-charset-string* can be one of the following:
  - **os_charset** Alias for the character set used by the operating system.
  - **char_charset** Alias for the CHAR character set used by the database.
  - **nchar_charset** Alias for the NCHAR character set used by the database.
  - **any other supported character set label** You can specify any of the SQL Anywhere supported character set labels.
- ○ **options** You can specify one of the following options:
  - **Read or write a BOM** By default, the values are set to **read_bom= on** and **write_bom=off**. You can change the values to **read_bom=off** and **write_bom=on**.

Returns

LONG BINARY

Remarks

You can view the list of character sets supported by SQL Anywhere by executing the following command:

```
dbinit -le
```

For more information about the character set labels you can use with this function, see "Supported character sets" [*SQL Anywhere Server - Database Administration*].

See also

- "String functions" on page 131

Standards and compatibility

- **SQL/2003** Vendor extension.

Examples

This fragment converts the mytext column from the Traditional Chinese character set to the Simplified Chinese character set:

```
SELECT CSCONVERT( mytext, 'cp936', 'cp950' )
FROM mytable;
```

This fragment converts the mytext column from the database character set to the Simplified Chinese character set:

```
SELECT CSCONVERT( mytext, 'cp936' )
FROM mytable;
```

If a file name is stored in the database, it is stored in the database character set. If the server is going to read from or write to a file whose name is stored in a database (for example, in an external stored procedure), the file name must be explicitly converted to the operating system character set before the file can be accessed. File names stored in the database and retrieved by the client are converted automatically to the client character set, so explicit conversion is not necessary.

This fragment converts the value in the filename column from the database character set to the operating system character set:

```
SELECT CSCONVERT( filename, 'os_charset' )
FROM mytable;
```

A table contains a list of file names. An external stored procedure takes a file name from this table as a parameter and reads information directly out of that file. The following statement works when character set conversion is not required:

```
SELECT MYFUNC( filename )
FROM mytable;
```

The *mytable* clause indicates a table with a filename column. However, if you need to convert the file name to the character set of the operating system, you would use the following statement.

```
SELECT MYFUNC( csconvert( filename, 'os_charset' ) )
FROM mytable;
```

CUME_DIST function [Ranking]

Computes the relative position of one value among a group of rows.

Syntax

CUME_DIST( ) OVER (*window-spec* **)**

window-spec : see the Remarks section below

Returns

A DOUBLE value between 0 and 1

Remarks

Composite sort keys are not currently allowed in the CUME_DIST function. You can use composite sort keys with any of the other rank functions.

Elements of *window-spec* can be specified either in the function syntax (inline), or in conjunction with a WINDOW clause in the SELECT statement. When used as a window function, you must specify an ORDER BY clause, you may specify a PARTITION BY clause, however, you can not specify a ROWS or RANGE clause. See the *window-spec* definition provided in "WINDOW clause" on page 793.

For more information about using window functions in SELECT statements, including working examples, see "Window functions" [*SQL Anywhere Server - SQL Usage*].

See also

- "DENSE_RANK function [Ranking]" on page 189
- "PERCENT_RANK function [Ranking]" on page 263
- "RANK function [Ranking]" on page 272

Standards and compatibility

- **SQL/2003** SQL/OLAP feature T612.

Example

The following example returns a result set that provides a cumulative distribution of the salaries of employees who live in California.

```
SELECT DepartmentID, Surname, Salary,
CUME_DIST() OVER (PARTITION BY DepartmentID
ORDER BY Salary DESC) "Rank"
FROM Employees
WHERE State IN ('CA');
```

Here is the result set:

DepartmentID	Surname	Salary	Rank
200	Savarino	72300.000	0.333333333333333
200	Clark	45000.000	0.666666666666667
200	Overbey	39300.000	1

DATALENGTH function [System]

Returns the length, in bytes, of the underlying storage for the result of an expression.

Syntax

DATALENGTH(*expression* **)**

Parameters

- **expression** Usually a column name. If *expression* is a string constant, you must enclose it in quotes.

Returns

UNSIGNED INT

Remarks

The return values of the DATALENGTH function are as follows:

Data type	DATALENGTH
SMALLINT	2
INTEGER	4
DOUBLE	8
CHAR	Length of the data
BINARY	Length of the data

This function supports NCHAR inputs and outputs.

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the length of the longest string in the CompanyName column.

```
SELECT MAX( DATALENGTH( CompanyName ) )
FROM Customers;
```

The following statement returns the length of the string '8sdofinsv8s7a7s7gehe4h':

```
SELECT DATALENGTH( '8sdofinsv8s7a7s7gehe4h' );
```

DATE function [Date and time]

Converts the expression into a date, and removes any hours, minutes, or seconds.

For information about controlling the interpretation of date formats, see "date_order option [database]" [*SQL Anywhere Server - Database Administration*].

Syntax

DATE(*expression* **)**

Returns

DATE

Parameters

- **expression** The value to be converted to date format, typically a string.

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value 1999-01-02 as a date.

```
SELECT DATE( '1999-01-02 21:20:53' );
```

The following statement returns the create dates of all the objects listed in the SYSOBJECT system view:

```
SELECT DATE( creation_time ) FROM SYSOBJECT;
```

DATEADD function [Date and time]

Returns the date produced by adding multiple date parts to a date.

Syntax

DATEADD(*date-part*, *numeric-expression*, *date-expression* **)**

date-part :
year
| **quarter**
| **month**
| **week**
| **day**
| **dayofyear**
| **hour**
| **minute**
| **second**
| **millisecond**

Parameters

- **date-part** The date part to be added to the date. For more information about date parts, see "Date parts" on page 126.
- **numeric-expression** The number of date parts to be added to the date. The *numeric-expression* can be any numeric type, but the value is truncated to an integer.
- **date-expression** The date to be modified.

Returns

TIMESTAMP

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value: 1995-11-02 00:00:00.000.

```
SELECT DATEADD( month, 102, '1987/05/02' );
```

DATEDIFF function [Date and time]

Returns the interval between two dates.

Syntax

DATEDIFF(*date-part*, *date-expression-1*, *date-expression-2* **)**

date-part :
year
| **quarter**
| **month**
| **week**
| **day**
| **dayofyear**
| **hour**
| **minute**
| **second**
| **millisecond**

Parameters

- **date-part** Specifies the date part in which the interval is to be measured.

  Choose one of the date objects listed above. For a complete list of date parts, see "Date parts" on page 126.

- **date-expression-1** The starting date for the interval. This value is subtracted from *date-expression-2* to return the number of *date-parts* between the two arguments.

- **date-expression-2** The ending date for the interval. *Date-expression-1* is subtracted from this value to return the number of *date-parts* between the two arguments.

Returns

INT

Remarks

This function calculates the number of date parts between two specified dates. The result is a signed integer value equal to (date2 - date1), in date parts.

The DATEDIFF function results are truncated, not rounded, when the result is not an even multiple of the date part.

When you use **day** as the date part, the DATEDIFF function returns the number of midnights between the two times specified, including the second date but not the first.

When you use **month** as the date part, the DATEDIFF function returns the number of first-of-the-months between two dates, including the second date but not the first.

When you use **week** as the date part, the DATEDIFF function returns the number of Sundays between the two dates, including the second date but not the first.

For the smaller time units there are overflow values:

- **milliseconds** 24 days
- **seconds** 68 years
- **minutes** 4083 years
- **others** No overflow limit

The function returns an overflow error if you exceed these limits.

Standards and compatibility

- **SQL/2003** Transact-SQL extension.

Example

The following statement returns 1.

```
SELECT DATEDIFF( hour, '4:00AM', '5:50AM' );
```

The following statement returns 102.

```
SELECT DATEDIFF( month, '1987/05/02', '1995/11/15' );
```

The following statement returns 0.

```
SELECT DATEDIFF( day, '00:00', '23:59' );
```

The following statement returns 4.

```
SELECT DATEDIFF( day,
   '1999/07/19 00:00',
   '1999/07/23 23:59' );
```

The following statement returns 0.

```
SELECT DATEDIFF( month, '1999/07/19', '1999/07/23' );
```

The following statement returns 1.

```
SELECT DATEDIFF( month, '1999/07/19', '1999/08/23' );
```

DATEFORMAT function [Date and time]

Returns a string representing a date expression in the specified format.

Syntax

DATEFORMAT(*datetime-expression*, *string-expression* **)**

Parameters

- **datetime-expression** The datetime to be converted.
- **string-expression** The format of the converted date.

  For information about date format descriptions, see "timestamp_format option [compatibility]" [*SQL Anywhere Server - Database Administration*].

This function supports NCHAR inputs and/or outputs.

Returns

VARCHAR

Remarks

Any allowable date format can be used for the string-expression.

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value Jan 01, 1989.

```
SELECT DATEFORMAT( '1989-01-01', 'Mmm dd, yyyy' );
```

DATENAME function [Date and time]

Returns the name of the specified part (such as the month June) of a datetime value, as a character string.

Syntax

DATENAME(*date-part*, *date-expression* **)**

Parameters

- **date-part** The date part to be named.

  For a complete listing of allowed date parts, see "Date parts" on page 126.

- **date-expression** The date for which the date part name is to be returned. The date must contain the requested *date-part*.

Returns

VARCHAR

Remarks

The DATENAME function returns a string, even if the result is numeric, such as 23 for the day.

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value May.

```
SELECT DATENAME( month, '1987/05/02' );
```

DATEPART function [Date and time]

Returns the value of part of a datetime value.

Syntax

DATEPART(*date-part*, *date-expression* **)**

Parameters

- **date-part** The date part to be returned.

  For a complete listing of allowed date parts, see "Date parts" on page 126.

- **date-expression** The date for which the part is to be returned.

Returns

INT

Remarks

The date must contain the *date-part* field.

The numbers that correspond week days depend on the setting of the first_day_of_week option. By default Sunday=7.

See also

- "first_day_of_week option [database]" [*SQL Anywhere Server - Database Administration*]
- "SET statement [T-SQL]" on page 737

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value 5.

```
SELECT DATEPART( month , '1987/05/02' );
```

The following example creates a table, TableStatistics, and inserts into it the total number of sales orders per year as stored in the SalesOrders table:

```
CREATE TABLE TableStatistics (
   ID INTEGER NOT NULL DEFAULT AUTOINCREMENT,
   Year INT,
   NumberOrders INT );
INSERT INTO TableStatistics ( Year, NumberOrders )
   SELECT DATEPART( Year, OrderDate ), COUNT(*)
   FROM SalesOrders
   GROUP BY DATEPART( Year, OrderDate );
```

DATETIME function [Date and time]

Converts an expression into a timestamp.

Syntax

DATETIME(*expression* **)**

Parameters

- **expression** The expression to be converted. It is generally a string.

Returns

TIMESTAMP

Remarks

Attempts to convert numerical values return an error.

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns a timestamp with value 1998-09-09 12:12:12.000.

```
SELECT DATETIME( '1998-09-09 12:12:12.000' );
```

DAY function [Date and time]

Returns an integer from 1 to 31.

Syntax

DAY(*date-expression* **)**

Parameters

- **date-expression** The date.

Returns

SMALLINT

Remarks

The integers 1 to 31 correspond to the day of the month in a date.

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value 12.

```
SELECT DAY( '2001-09-12' );
```

DAYNAME function [Date and time]

Returns the name of the day of the week from a date.

Syntax

DAYNAME(*date-expression* **)**

Parameters

- **date-expression** The date.

Returns

VARCHAR

Remarks

The English names are returned as: Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday.

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value Saturday.

```
SELECT DAYNAME ( '1987/05/02' );
```

DAYS function [Date and time]

A function that evaluates days. For specific details, see this function's usage.

Syntax 1: integer

DAYS([*datetime-expression*,] *datetime-expression* **)**

Syntax 2: timestamp

DAYS(*datetime-expression*, *integer-expression* **)**

Parameters

- **datetime-expression** A date and time.
- **integer-expression** The number of days to be added to the *datetime-expression*. If the *integer-expression* is negative, the appropriate number of days is subtracted from the timestamp. If you supply an integer expression, the *datetime-expression* must be explicitly cast as a date or timestamp.

For information about casting data types, see "CAST function [Data type conversion]" on page 148.

Returns

INT when you specify two datetime expressions.

TIMESTAMP when the second argument you specify is an integer.

Remarks

The behavior of this function varies depending on the information you specify:

- If you give a single date, this function returns the number of days since 0000-02-29.

> **Note**
> 0000-02-29 is not meant to imply an actual date; it is the date used by the date algorithm.

- If you give two dates, this function returns the integer number of days between them. Instead, use the DATEDIFF function.
- If you give a date and an integer, this function adds the integer number of days to the specified date. Instead, use the DATEADD function.

This function ignores hours, minutes, and seconds.

See also

- "DATEDIFF function [Date and time]" on page 174
- "DATEADD function [Date and time]" on page 173

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the integer 729889.

```
SELECT DAYS( '1998-07-13 06:07:12' );
```

The following statements return the integer value -366, indicating that the second date is 366 days prior to the first. It is recommended that you use the second example (DATEDIFF).

```
SELECT DAYS( '1998-07-13 06:07:12',
             '1997-07-12 10:07:12' );

SELECT DATEDIFF( day,
   '1998-07-13 06:07:12',
   '1997-07-12 10:07:12' );
```

The following statements return the timestamp 1999-07-14 00:00:00.000. It is recommended that you use the second example (DATEADD).

```
SELECT DAYS( CAST('1998-07-13' AS DATE ), 366 );

SELECT DATEADD( day, 366, '1998-07-13' );
```

DB_EXTENDED_PROPERTY function [System]

Returns the value of the given property. Allows an optional property-specific string parameter to be specified.

Syntax

DB_EXTENDED_PROPERTY(
{ *property-id* | *property-name* }
[**,** *property-specific-argument*
[**,** *database-id* | *database-name*]]
)

Parameters

- **property-id** The database property ID to query.
- **property-name** The database property name to query.

  For a complete list of database properties, see "Database properties" [*SQL Anywhere Server - Database Administration*].
- **property-specific-argument** The following database properties allow you to specify additional arguments, as noted below, to return specific information about the property.
  - **CharSet property** Specify the name of a standard to obtain the default CHAR character set label for the standard. Possible values you can specify are: ASE, IANA, MIME, JAVA, WINDOWS, UTR22, IBM, and ICU. If no standard is specified, IANA is used as the default, unless the database connection was made through TDS, in which case ASE is the default.
  - **CatalogCollation, Collation, and NcharCollation properties** When querying these properties, the following values can be specified as a *property-specific-argument* to return information specific to the collation:
    - **AccentSensitive** Specify AccentSensitive to obtain the accent sensitivity setting for the collation. For example, the following statement returns the accent sensitivity setting for the NCHAR collation:

      ```
      SELECT DB_EXTENDED_PROPERTY( 'NcharCollation', 'AccentSensitive');
      ```

      Possible return values are: Ignore, Respect, and French. For a description of these values, see "Collation tailoring options" [*SQL Anywhere Server - Database Administration*].
    - **CaseSensitivity** Specify CaseSensitivity to obtain the case sensitivity setting for the collation. Possible return values are: Ignore, Respect, UpperFirst, and LowerFirst. For a description of these values, see "Collation tailoring options" [*SQL Anywhere Server - Database Administration*].
    - **PunctuationSensitivity** Specify PunctuationSensitivity to obtain the punctuation sensitivity setting for the collation. Possible return values are: Ignore, Primary, and Quaternary. For a description of these values, see "Collation tailoring options" [*SQL Anywhere Server - Database Administration*].
    - **Properties** Specify Properties to obtain a string containing all the tailoring options specified for the collation. For a description of the keywords and values in the returned string, see "Collation tailoring options" [*SQL Anywhere Server - Database Administration*].

        - **Specification** Specify Specification to obtain a string containing the full collation specification used for the collation. For a description of the keywords and values in the returned string, see "Collation tailoring options" [*SQL Anywhere Server - Database Administration*].

    - **DriveType property** Specify the name of a dbspace, or the file ID for the dbspace, to obtain its drive type. The value returned is one of the following: CD, FIXED, RAMDISK, REMOTE, REMOVABLE, or UNKNOWN. If nothing is specified, the drive type of the system dbspace is returned. If the specified dbspace doesn't exist, the property function returns NULL. If the name of a dbspace is specified and the ID of a database that isn't the database of the current connection is also specified, the function also returns NULL.

    - **File property** Specify a dbspace name to obtain the file name of the database root file, including the path. If nothing is specified, information for the system dbspace is returned. If the specified file doesn't exist, the function returns NULL.

    - **FileSize property** Specify the name of a dbspace, or the file ID for the dbspace, to obtain the size of the specified file. You can also specify temporary to return the size of the temporary dbspace, or translog to return the size of the log file. If nothing is specified, the size of the system dbspace is returned. If the specified file doesn't exist, the function returns NULL.

    - **FreePages property** Specify the name of a dbspace, or the file ID for the dbspace, to obtain the number of free pages. You can also specify temporary to return the number of free pages in the temporary dbspace, or translog to return the number of free pages in the log file. If nothing is specified, the number of free pages in the system dbspace is returned. If the specified file doesn't exist, the function returns NULL.

    - **IOParallelism property** Specify a dbspace name to obtain the estimated number of simultaneous I/O operations supported by the dbspace. If a dbspace is not specified, the current system dbspace is used.

    - **NextScheduleTime property** Specify an event name to obtain its next scheduled execution time.

- **database-id** The database ID number, as returned by the DB_ID function. Typically, the database name is used.

- **database-name** The name of the database, as returned by the DB_NAME function.

Returns

Returns a VARCHAR value. The current database is used if the third argument is omitted.

Remarks

The DB_EXTENDED_PROPERTY function is similar to the DB_PROPERTY function except that it allows an optional *property-specific-argument* string parameter to be specified. The interpretation of *property-specific-argument* depends on the property ID or name specified in the first argument.

When comparing catalog strings such as table names and procedure names, the database server uses the CHAR collation. For the UCA collation, the catalog collation is the same as the CHAR collation but with the tailoring changed to be case-insensitive, accent-insensitive and with punctuation sorted in the primary level. For legacy collations, the catalog collation is the same as the CHAR collation but with the tailoring changed to be case-insensitive. While you cannot explicitly specify the tailoring used for the catalog collation, you can query the Specification property to obtain the full collation specification used by the database server for comparing catalog strings. Querying the Specification property can be useful if you need to exploit the difference between the CHAR and catalog collations. For example, suppose you have a punctuation-insensitive CHAR collation and you want to execute an upgrade script that defines a procedure called my_procedure, and that also attempts to delete an old version named myprocedure. The following statements cannot achieve the desired results because my_procedure is equivalent to myprocedure, using the CHAR collation:

```
CREATE PROCEDURE my_procedure( ) ...;
IF EXISTS ( SELECT * FROM SYS.SYSPROCEDURE WHERE proc_name = 'myprocedure' )
THEN DROP PROCEDURE myprocedure
END IF;
```

Instead, you could execute the following statements to achieve the desired results:

```
CREATE PROCEDURE my_procedure( ) ...;
IF EXISTS ( SELECT * FROM SYS.SYSPROCEDURE
   WHERE COMPARE( proc_name, 'myprocedure',
DB_EXTENDED_PROPERTY( 'CatalogCollation', 'Specification' ) ) = 0 )
THEN DROP PROCEDURE myprocedure
END IF;
```

See also

- “DB_ID function [System]” on page 184
- “DB_NAME function [System]” on page 184
- “Database properties” [*SQL Anywhere Server - Database Administration*]
- “CONNECTION_PROPERTY function [System]” on page 157
- “CONNECTION_EXTENDED_PROPERTY function [String]” on page 156

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the location of the current database:

```
SELECT DB_EXTENDED_PROPERTY( 'File' );
```

The following statement returns the file size of the system dbspace, in pages.

```
SELECT DB_EXTENDED_PROPERTY( 'FileSize' );
```

The following statement returns the file size of the transaction log, in pages.

```
SELECT DB_EXTENDED_PROPERTY( 'FileSize', 'translog' );
```

The following statement returns the case sensitivity setting for the NCHAR collation:

```
SELECT DB_EXTENDED_PROPERTY( 'NcharCollation',' CaseSensitivity' );
```

The following statement returns the tailoring options specified for the database CHAR collation:

```
SELECT DB_EXTENDED_PROPERTY ( 'Collation', 'Properties' );
```

The following statement returns the full collation specification for the database NCHAR collation:

```
SELECT DB_EXTENDED_PROPERTY( 'NCharCollation', 'Specification' );
```

DB_ID function [System]

Returns the database ID number.

Syntax

DB_ID([*database-name*] **)**

Parameters

- **database-name** A string containing the database name. If no *database-name* is supplied, the ID number of the current database is returned.

Returns

INT

See also

- "global_database_id option [database]" [*SQL Anywhere Server - Database Administration*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The statement returns the value 0, when executed against the SQL Anywhere sample database as the sole database on the server.

```
SELECT DB_ID( 'demo' );
```

The following statement returns the value 0 if executed against the only running database.

```
SELECT DB_ID( );
```

DB_NAME function [System]

Returns the name of a database with a given ID number.

Syntax

DB_NAME([*database-id*] **)**

Parameters

- **database-id** The ID of the database. The *database-id* must be a numeric expression.

Returns

VARCHAR

Remarks

If no database ID is supplied, the name of the current database is returned.

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The statement returns the database name demo, when executed against the SQL Anywhere sample database as the sole database on the server.

```
SELECT DB_NAME( 0 );
```

DB_PROPERTY function [System]

Returns the value of the given property.

Syntax

DB_PROPERTY(
{ *property-id* | *property-name* }
[**,** *database-id* | *database-name*]
)

Parameters

- **property-id** The database property ID.
- **property-name** The database property name.
- **database-id** The database ID number, as returned by the DB_ID function. Typically, the database name is used.
- **database-name** The name of the database, as returned by the DB_NAME function.

Returns

VARCHAR

Remarks

Returns a string. The current database is used if the second argument is omitted.

See also

- "DB_ID function [System]" on page 184
- "DB_NAME function [System]" on page 184
- "Database properties" [*SQL Anywhere Server - Database Administration*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the page size of the current database, in bytes.

```
SELECT DB_PROPERTY( 'PageSize' );
```

DECOMPRESS function [String]

Decompresses the string and returns a LONG BINARY value.

Syntax

DECOMPRESS(*string-expression* [**,** *compression-algorithm-alias*] **)**

Parameters

- **string-expression** The string to decompress. Binary values can also be passed to this function. This parameter is case sensitive, even in case-insensitive databases.
- **compression-algorithm-alias** Alias (string) for the algorithm to use for decompression. The supported values are zip and gzip (both are based on the same algorithm, but use different headers and trailers).

  Zip is a widely supported compression algorithm. Gzip is compatible with the gzip utility on Unix, whereas the zip algorithm is not.

  If no algorithm is specified, the function attempts to detect which algorithm was used to compress the string. If the incorrect algorithm is specified, or the correct algorithm cannot be detected, the string is not decompressed.

  For more information about compression, see "COMPRESS function [String]" on page 155.

Returns

LONG BINARY

Remarks

This function can be used to decompress a value that was compressed using the COMPRESS function.

You do not need to use the DECOMPRESS function on values that are stored in a compressed column. Compression and decompression of values in a compressed column are handled automatically by the database server. See "Choosing whether to compress columns" [*SQL Anywhere Server - SQL Usage*].

See also

- "COMPRESS function [String]" on page 155
- "String functions" on page 131

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following example uses the DECOMPRESS function to decompress values from the Attachment column of a fictitious table, TableA:

```
SELECT DECOMPRESS ( Attachment, 'gzip' )
FROM TableA;
```

Since DECOMPRESS returns binary values, if the original values were of a character type, such as LONG VARCHAR, a CAST can be applied to return human-readable values:

```
SELECT CAST ( DECOMPRESS ( Attachment, 'gzip' )
AS LONG VARCHAR ) FROM TableA;
```

DECRYPT function [String]

Decrypts the string using the supplied key and returns a LONG BINARY value.

Syntax

DECRYPT(*string-expression*, *key*
[, *algorithm*]
)

algorithm :
'AES'
| **'AES256'**
| **'AES_FIPS'**
| **'AES256_FIPS'**

Parameters

- **string-expression** The string to be decrypted. Binary values can also be passed to this function. This parameter is case sensitive, even in case-insensitive databases.
- **key** The encryption key (string) required to decrypt the *string-expression*. This must be the same encryption key that was used to encrypt the *string-expression* to obtain the original value that was encrypted. This parameter is case sensitive, even in case-insensitive databases.

Caution
For strongly encrypted databases, be sure to store a copy of the key in a safe location. If you lose the encryption key there is no way to access the data, even with the assistance of technical support. The database must be discarded and you must create a new database.

- **algorithm** This optional parameter specifies the algorithm originally used to encrypt the *string-expression*.

Returns

LONG BINARY

Remarks

For more information about the supported encryption algorithms, see "ENCRYPT function [String]" on page 193.

You can use the DECRYPT function to decrypt a *string-expression* that was encrypted with the ENCRYPT function. This function returns a LONG BINARY value with the same number of bytes as the input string.

To successfully decrypt a *string-expression*, you must use the same encryption key that was used to encrypt the data. If you specify an incorrect encryption key, an error is generated. A lost key will result in inaccessible data, from which there is no recovery.

Note
FIPS is not available on all platforms. For a list of supported platforms, see http://www.sybase.com/detail?id=1061806.

See also

- "ENCRYPT function [String]" on page 193
- "Encrypting portions of a database" [*SQL Anywhere Server - Database Administration*]
- "String functions" on page 131
- "-fips server option" [*SQL Anywhere Server - Database Administration*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following example decrypts a user's password from the user_info table. The CAST function is used to convert the password back to a CHAR data type because the DECRYPT function converts values to the LONG BINARY data type, which is unreadable.

```
SELECT CAST( DECRYPT( user_pwd, '8U3dkA' ) AS CHAR(100) ) FROM user_info;
```

DEGREES function [Numeric]

Converts a number from radians to degrees.

Syntax

DEGREES(*numeric-expression* **)**

Parameters

- **numeric-expression** An angle in radians.

Returns

Returns the degrees of the angle given by *numeric-expression*.

DOUBLE

Remarks

This function converts its argument to DOUBLE, AND performs the computation in double-precision floating point. If the parameter is NULL, the result is NULL.

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value 29.79380534680281.

```
SELECT DEGREES( 0.52 );
```

DENSE_RANK function [Ranking]

Calculates the rank of a value in a partition. In the case of tied values, the DENSE_RANK function does not leave gaps in the ranking sequence.

Syntax

DENSE_RANK( ) OVER (*window-spec* **)**

window-spec : see the Remarks section below

Returns

INTEGER

Remarks

Elements of *window-spec* can be specified either in the function syntax (inline), or in conjunction with a WINDOW clause in the SELECT statement. When used as a window function, you must specify an ORDER BY clause, you may specify a PARTITION BY clause, however, you can not specify a ROWS or RANGE clause. See the *window-spec* definition provided in "WINDOW clause" on page 793.

For more information about using window functions in SELECT statements, including working examples, see "Window functions" [*SQL Anywhere Server - SQL Usage*].

See also

- "CUME_DIST function [Ranking]" on page 170
- "PERCENT_RANK function [Ranking]" on page 263
- "RANK function [Ranking]" on page 272

Standards and compatibility

- **SQL/2003** SQL/OLAP feature T612.

Example

The following example returns a result set that provides a ranking of the employees' salaries in Utah and New York. Although 19 records are returned in the result set, only 18 rankings are listed because of a 7th-place tie between the 7th and 8th employee in the list, who have identical salaries. Instead of ranking the 9th employee as '9', the employee is listed as '8' because the DENSE_RANK function does not leave gaps in the ranks.

```
SELECT DepartmentID, Surname, Salary, State,
DENSE_RANK() OVER (ORDER BY Salary DESC) AS SalaryRank
FROM Employees
WHERE State IN ('NY','UT');
```

Here is the result set:

DepartmentID	Surname	Salary	State	SalaryRank
100	Shishov	72995.000	UT	1
100	Wang	68400.000	UT	2
100	Cobb	62000.000	UT	3
400	Morris	61300.000	UT	4
300	Davidson	57090.000	NY	5
200	Martel	55700.000	NY	6
100	Blaikie	54900.000	NY	7
400	Diaz	54900.000	UT	7
100	Driscoll	48023.000	UT	8
400	Hildebrand	45829.000	UT	9
100	Whitney	45700.000	NY	10
100	Guevara	42998.000	NY	11
100	Soo	39075.000	NY	12
200	Goggin	37900.000	UT	13
400	Wetherby	35745.000	NY	14
400	Ahmed	34992.000	NY	15
500	Rebeiro	34576.000	UT	16
300	Bigelow	31200.000	UT	17
500	Lynch	24903.000	UT	18

DIFFERENCE function [String]

Returns the difference in the SOUNDEX values between the two string expressions.

Syntax

DIFFERENCE (*string-expression-1*, *string-expression-2* **)**

Parameters

- **string-expression-1** The first SOUNDEX argument.
- **string-expression-2** The second SOUNDEX argument.

Returns

SMALLINT

Remarks

The DIFFERENCE function compares the SOUNDEX values of two strings and evaluates the similarity between them, returning a value from 0 through 4, where 4 is the best match.

This function always returns some value. The result is NULL only if one of the arguments are NULL.

See also

- "SOUNDEX function [String]" on page 308
- "String functions" on page 131

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value 3.

```
SELECT DIFFERENCE( 'test', 'chest' );
```

DOW function [Date and time]

Returns a number from 1 to 7 representing the day of the week of a date, where Sunday=1, Monday=2, and so on.

Syntax

DOW(*date-expression* **)**

Parameters

- **date-expression** The date to evaluate.

Returns

SMALLINT

Remarks

The DOW function is not affected by the value specified for the first_day_of_week database option. For example, even if first_day_of_week is set to Monday, the DOW function returns a 2 for Monday.

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value 5.

```
SELECT DOW( '1998-07-09' );
```

The following statement queries the Employees table and returns the employees StartDate, expressed as the number of the day of the week:

```
SELECT DOW( StartDate ) FROM Employees;
```

SQL functions (E-O)

Each function is listed, and the function type (numeric, character, and so on) is indicated next to it.

For links to all functions of a given type, see "Function types" on page 124.

See also

- "SQL functions (A-D)" on page 135
- "SQL functions (P-Z)" on page 262

ENCRYPT function [String]

Encrypts the specified values using the supplied encryption key and returns a LONG BINARY value.

Syntax

ENCRYPT(*string-expression*, *key*
[, *algorithm*]
)

algorithm :
'AES'
| **'AES256'**
| **'AES_FIPS'**
| **'AES256_FIPS'**

Parameters

- **string-expression** The data to be encrypted. Binary values can also be passed to this function. This parameter is case sensitive, even in case-insensitive databases.
- **key** The encryption key used to encrypt the *string-expression*. This same key must be used to decrypt the value to obtain the original value. This parameter is case sensitive, even in case-insensitive databases.

  As with most passwords, it is best to choose a key value that cannot be easily guessed. It is recommended that you choose a value for your key that is at least 16 characters long, contains a mix of uppercase and lowercase, and includes numbers, letters and special characters. You will require this key each time you want to decrypt the data.

  Caution
  For strongly encrypted databases, be sure to store a copy of the key in a safe location. If you lose the encryption key there is no way to access the data, even with the assistance of technical support. The database must be discarded and you must create a new database.

- **algorithm** This optional parameter specifies the algorithm to use when encrypting *string-expression*. The algorithm used for strong encryption is Rijndael: a block encryption algorithm chosen as the new Advanced Encryption Standard (AES) for block ciphers by the National Institute of Standards and Technology (NIST).

  You can specify one of the FIPS algorithms for *algorithm* on any platform that supports FIPS.

If *algorithm* is not specified, AES is used by default. If the database server was started using the -fips server option, AES_FIPS is used as the default instead.

Returns

LONG BINARY

Remarks

The LONG BINARY value returned by this function is at most 31 bytes longer than the input *string-expression*. The value returned by this function is not human-readable. You can use the DECRYPT function to decrypt a *string-expression* that was encrypted with the ENCRYPT function. To successfully decrypt a *string-expression*, you must use the same encryption key and algorithm that were used to encrypt the data. If you specify an incorrect encryption key, an error is generated. A lost key will result in inaccessible data, from which there is no recovery.

If you are storing encrypted values in a table, the column should be BINARY or LONG BINARY so that character set conversion is not performed on the data.

Note
FIPS is not available on all platforms. For a list of supported platforms, see http://www.sybase.com/detail?id=1061806.

See also

- "DECRYPT function [String]" on page 187
- "Encrypting portions of a database" [*SQL Anywhere Server - Database Administration*]
- "-fips server option" [*SQL Anywhere Server - Database Administration*]

Standards and compatibility

- **SQL/2003** SQL foundation feature outside core SQL.

Example

The following trigger encrypts the user_pwd column of the user_info table. This column contains users' passwords, and the trigger fires whenever the password value is changed.

```
CREATE TRIGGER encrypt_updated_pwd
BEFORE UPDATE OF user_pwd
ON user_info
REFERENCING NEW AS new_pwd
FOR EACH ROW
BEGIN
    SET new_pwd.user_pwd=ENCRYPT( new_pwd.user_pwd, '8U3dkA' );
END;
```

ERRORMSG function [Miscellaneous]

Provides the error message for the current error, or for a specified SQLSTATE or SQLCODE value.

Syntax

ERRORMSG([*sqlstate* | *sqlcode*] **)**

sqlstate: *string*

sqlcode: *integer*

Parameters

- **sqlstate** The SQLSTATE value for which the error message is to be returned.
- **sqlcode** The SQLCODE value for which the error message is to be returned.

Returns

A string containing the error message.

VARCHAR

Remarks

If no argument is supplied, the error message for the current state is supplied. Any substitutions (such as table names and column names) are made.

If an argument is supplied, the error message for the supplied SQLSTATE or SQLCODE is returned, with no substitutions. Table names and column names are supplied as placeholders (%1).

See also

- "SQL Anywhere error messages sorted by SQLSTATE" [*Error Messages*]
- "SQL Anywhere error messages sorted by SQLCODE" [*Error Messages*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the error message for SQLCODE -813.

```
SELECT ERRORMSG( -813 );
```

ESTIMATE function [Miscellaneous]

Returns selectivity estimates as a percentage calculated by the query optimizer, based on specified parameters.

Syntax

ESTIMATE(*column-name* [, *value* [, *relation-string*]] **)**

Parameters

- **column-name** The column used in the estimate.

- **value** The value to which the column is compared. The default is NULL.
- **relation-string** The comparison operator used for the comparison, enclosed in single quotes. Possible values for this parameter are: '=' , '>' , '<' , '>=' , '<=' , '<>' , '!=' , '!<' , and '!>'. The default is '='.

Returns

REAL

Remarks

If *value* is NULL then the relation strings '=' and '!=' are interpreted as the IS NULL and IS NOT NULL conditions, respectively.

See also

- “INDEX_ESTIMATE function [Miscellaneous]” on page 228
- “ESTIMATE_SOURCE function [Miscellaneous]” on page 196
- “EXPERIENCE_ESTIMATE function [Miscellaneous]” on page 203

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the percentage of EmployeeID values estimated to be greater than 200. The precise value depends on the actions you have carried out on the database.

```
SELECT FIRST ESTIMATE( EmployeeID, 200, '>' )
   FROM Employees
   ORDER BY 1;
```

ESTIMATE_SOURCE function [Miscellaneous]

Provides the source for selectivity estimates used by the query optimizer.

Syntax

ESTIMATE_SOURCE(
column-name
[, *value*
[, *relation-string*]]
)

Parameters

- **column-name** The name of the column that is being investigated.
- **value** The value to which the column is compared. The default is NULL.
- **relation-string** The comparison operator used for the comparison, enclosed in single quotes. Possible values for this parameter are: '=' , '>' , '<' , '>=' , '<=' , '<>' , '!=' , '!<' , and '!>'. The default is '='.

Returns

The source of the selectivity estimate which can be one of the following:

- **Statistics** is used as the source when you have specified a value, and there is a stored statistic available that estimates the average selectivity of the value in the column. The statistic is available only when the selectivity of the value is a significant enough number that it is stored in the statistics. Currently, a value is deemed significant if it occurs in at least 1% of the rows.
- **Column** is similar to Statistics, except that the selectivity of the value occurs in less than 1% of the rows. In this case, the selectivity that is used is the average of all values that have been stored in the statistics that occur in less than 1% of rows.
- **Guess** is returned when there is no relevant index to use, and no statistics have been collected for the column. In this case, built-in guesses are used.
- **Column-column** is returned when the estimate that is used is the selectivity of a join. In this case, the estimate is calculated as the number of rows in the joined result set divided by the number of rows in the Cartesian product of the two tables.
- **Index** is used as the source when there are no statistics available to estimate the selectivity, but there is an index which can be probed to estimate selectivity.
- **User** is returned when there is a user supplied estimate, and the user_estimates database option is not set to Disabled.

  For more information, see “user_estimates option [database]” [*SQL Anywhere Server - Database Administration*].
- **Computed** is returned when statistics are computed by the optimizer based on other information. For example, SQL Anywhere does not maintain statistics on multiple columns, so if you want an estimate on a multiple column equation, such as x=5 and y=10, and there are statistics on the columns x and y, then the optimizer creates an estimate by multiplying the estimated selectivity for each column.
- **Always** is used when the test is by definition true. For example, if the value is 1=1.
- **Combined** is used when the optimizer uses more than one of the above sources, and combines them.
- **Bounded** can qualify one of the other sources. This indicates that SQL Anywhere has placed an upper and/or lower bound on the estimate. The optimizer does this to keep estimates within logical bounds. For example, it ensures that an estimate is not greater than 100%, or that the selectivity is not less than one row.

Remarks

If *value* is NULL then the relation strings '=' and '!=' are interpreted as the IS NULL and IS NOT NULL conditions, respectively.

See also

- “ESTIMATE function [Miscellaneous]” on page 195
- “INDEX_ESTIMATE function [Miscellaneous]” on page 228

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value Index, which means that the query optimizer probed an index to estimate the selectivity.

```
SELECT FIRST ESTIMATE_SOURCE( EmployeeID, 200, '>' )
   FROM Employees
   ORDER BY 1;
```

EVENT_CONDITION function [System]

Specifies when an event handler is triggered.

Syntax

EVENT_CONDITION(*condition-name* **)**

Parameters

- **condition-name** The condition triggering the event. The possible values are preset in the database, and are case insensitive. Each condition is valid only for certain event types. The conditions and the events for which they are valid are as follows:

Condition name	Units	Valid for...	Comments
DBFreePercent	n/a	DBDiskSpace	
DBFreeSpace	MB	DBDiskSpace	
DBSize	MB	GrowDB	
ErrorNumber	n/a	RAISERROR	
IdleTime	seconds	ServerIdle	
Interval	seconds	All	Time since handler last executed
LogFreePercent	n/a	LogDiskSpace	
LogFreeSpace	MB	LogDiskSpace	
LogSize	MB	GrowLog	
RemainingValues	integer	GlobalAutoincrement	The number of remaining values
TempFreePercent	n/a	TempDiskSpace	
TempFreeSpace	MB	TempDiskSpace	
TempSize	MB	GrowTemp	

Returns

INT

Remarks

The EVENT_CONDITION function returns NULL when not called from an event.

See also

- "CREATE EVENT statement" on page 450

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following event definition uses the EVENT_CONDITION function:

```
CREATE EVENT LogNotifier
TYPE LogDiskSpace
WHERE event_condition( 'LogFreePercent' ) < 50
HANDLER
BEGIN
   MESSAGE 'LogNotifier message'
END;
```

EVENT_CONDITION_NAME function [System]

Lists the possible parameters for EVENT_CONDITION.

Syntax

EVENT_CONDITION_NAME(*integer* **)**

Parameters

- **integer** Must be greater than or equal to zero.

Returns

VARCHAR

Remarks

You can use the EVENT_CONDITION_NAME function to obtain a list of all arguments for the EVENT_CONDITION function by looping over integers until the function returns NULL.

The EVENT_CONDITION_NAME function returns NULL when not called from an event.

See also

- "CREATE EVENT statement" on page 450

Standards and compatibility

- **SQL/2003** Vendor extension.

EVENT_PARAMETER function [System]

Provides context information for event handlers.

Syntax

EVENT_PARAMETER(*context-name* **)**

context-name:
AppInfo
| **ConnectionID**
| **DisconnectReason**
| **EventName**
| **Executions**
| **MirrorServerName**
| **NumActive**
| **ScheduleName**
| **SQLCODE**
| **TableName**
| **User**
| *condition-name*

Parameters

- **context-name** One of the preset strings. The strings must be quoted, are case insensitive, and carry the following information:
  - **AppInfo** The value of the AppInfo connection property for the connection that caused the event to be triggered. Use the following statement to see the value of the property outside the context of the event:

    ```
    SELECT CONNECTION_PROPERTY( 'AppInfo' );
    ```

    This parameter is valid for Connect, Disconnect, ConnectFailed, BackupEnd, and RAISERROR events. The AppInfo string contains the computer name and application name of the client connection for embedded SQL, ODBC, OLE DB, ADO.NET, and iAnywhere JDBC driver connections.
  - **ConnectionId** The connection ID of the connection that caused the event to be triggered.
  - **DisconnectReason** A string indicating the reason the connect was terminated. This parameter is valid only for Disconnect events. Possible results include:
    - **abnormal** A disconnect occurred as a result of the client application terminating abnormally prior to disconnecting from the database, or as a result of a communication failure between the client and server computers.
    - **connect failed** A connection attempt failed.
    - **drop connection** A DROP CONNECTION statement was executed.
    - **from client** The client application disconnected.

    - **inactive** No requests were received for the period specified by the -ti server option.
    - **liveness** No liveness packets were received for the period specified by the -tl server option.
  - **EventName** The name of the event that has been triggered.
  - **Executions** The number of times the event handler has been executed.
  - **MirrorServerName** The name of the mirror or arbiter server that lost its connection to the primary server in a database mirroring system.
  - **NumActive** The number of active instances of an event handler. This is useful if you want to limit an event handler so that only one instance executes at any given time.
  - **ScheduleName** The name of the schedule which caused an event to be fired. If the event was fired manually using TRIGGER EVENT or as a system event, the result will be an empty string. If the schedule was not assigned a name explicitly when it was created, its name will be the name of the event.
  - **SQLCODE** The SQLCODE of the error that occurred during a failed connection. This parameter is valid only for ConnectFailed events.
  - **TableName** The name of the table, for use with RemainingValues.
  - **User** The user ID for the user that caused the event to be triggered.

In addition, you can access any of the valid *condition-name* arguments to the EVENT_CONDITION function from the EVENT_PARAMETER function.

The following table indicates which context-name values are valid for which system event types.

Context-name value	Valid system event types
AppInfo	BackupEnd, "Connect", ConnectFailed, "Disconnect", "RAISERROR", user events
ConnectionID	BackupEnd, "Connect", "Disconnect", Global Autoincrement, "RAISERROR", user events
DisconnectReason	"Disconnect"
EventName	all
Executions	all
NumActive	all
SQLCODE	ConnectFailed
TableName	GlobalAutoincrement
User	BackupEnd, "Connect", ConnectFailed, "Disconnect", GlobalAutoincrement, "RAISERROR", user events

Returns

VARCHAR

Remarks

The maximum size of values passed to an event is limited by the maximum page size for the server (-gp server option). Values that are longer are truncated to be less than the maximum page size.

See also

- "EVENT_CONDITION function [System]" on page 198
- "CREATE EVENT statement" on page 450
- "TRIGGER EVENT statement" on page 767
- "-gp server option" [*SQL Anywhere Server - Database Administration*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following example shows how to pass a string parameter to an event. The event displays the time it was triggered in the database server messages window.

```
CREATE EVENT ev_PassedParameter
HANDLER
BEGIN
  MESSAGE 'ev_PassedParameter - was triggered at ' ||
event_parameter( 'time' );
END;
TRIGGER EVENT ev_PassedParameter( "Time"=string(current timestamp ) );
```

EXP function [Numeric]

Returns the exponential function, e to the power of a number.

Syntax

EXP(*numeric-expression* **)**

Parameters

- **numeric-expression** The exponent.

Returns

DOUBLE

Remarks

The EXP function returns the exponential of the value specified by *numeric-expression*.

This function converts its argument to DOUBLE, performs the computation in double-precision floating point, and returns a DOUBLE as the result. If the parameter is NULL, the result is NULL.

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The statement returns the value 3269017.3724721107.

```
SELECT EXP( 15 );
```

EXPERIENCE_ESTIMATE function [Miscellaneous]

Returns selectivity estimates from the frequency table as a percentage calculated by the query optimizer, based on specified parameters.

Syntax

EXPERIENCE_ESTIMATE(
column-name
[**,** *value*
[**,** *relation-string*]]
)

Parameters

- **column-name** The name of the column that is being investigated.
- **value** The value to which the column is compared.
- **relation-string** The comparison operator used for the comparison. Possible values for this parameter are: '=' , '>' , '<' , '>=' , '<=' , '<>' , '!=' , '!<' , and '!>'. The default is '='.

Returns

REAL

Remarks

If *value* is NULL then the relation strings = and != are interpreted as the IS NULL and IS NOT NULL conditions, respectively.

See also

- “ESTIMATE function [Miscellaneous]” on page 195
- “INDEX_ESTIMATE function [Miscellaneous]” on page 228
- “ESTIMATE_SOURCE function [Miscellaneous]” on page 196

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns NULL.

```
SELECT DISTINCT EXPERIENCE_ESTIMATE( EmployeeID, 200, '>' )
FROM Employees;
```

EXPLANATION function [Miscellaneous]

Returns the plan optimization strategy of a SQL statement.

Syntax

EXPLANATION(
string-expression
[, *cursor-type*]
[, *update-status*]
)

Parameters

- **string-expression** The SQL statement, which is commonly a SELECT statement, but can also be an UPDATE or DELETE statement.
- **cursor-type** A cursor type, expressed as a string. Possible values are asensitive, insensitive, sensitive, or keyset-driven. If *cursor-type* is not specified, asensitive is used by default.
- **update-status** A string parameter accepting one of the following values indicating how the optimizer should treat the given cursor:

Value	Description
READ-ONLY	The cursor is read-only.
READ-WRITE (default)	The cursor can be read or written to.
FOR UPDATE	The cursor can be read or written to. This is the same as READ-WRITE.

Returns

LONG VARCHAR

Remarks

The optimization is returned as a string.

This information can help you decide which indexes to add or how to structure your database for better performance.

See also

- "Execution plans in UltraLite" [*UltraLite - Database Management and Reference*]
- "Reading execution plans" [*SQL Anywhere Server - SQL Usage*]
- "PLAN function [Miscellaneous]" on page 265
- "GRAPHICAL_PLAN function [Miscellaneous]" on page 211

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement passes a SELECT statement as a string parameter and returns the plan for executing the query.

```
SELECT EXPLANATION( 'SELECT * FROM Departments WHERE DepartmentID > 100' );
```

The following statement returns a string containing the short form of the text plan for an INSENSITIVE cursor over the query 'select * from Departments where ....'.

```
SELECT EXPLANATION( 'SELECT * FROM Departments WHERE DepartmentID > 100',
   'insensitive', 'read-only' );
```

EXPRTYPE function [Miscellaneous]

Returns a string that identifies the data type of an expression.

Syntax

EXPRTYPE(*string-expression*, *integer-expression* **)**

Parameters

- **string-expression** A SELECT statement. The expression whose data type is to be queried must appear in the select list. If the string is not a valid SELECT statement, NULL is returned.
- **integer-expression** The position in the select list of the desired expression. The first item in the select list is numbered 1. If the integer-expression value does not correspond to a SELECT list item, NULL is returned.

Returns

LONG VARCHAR

See also

- "SQL data types" on page 77
- "sa_describe_query system procedure" on page 864

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns `smallint` when executed against the SQL Anywhere sample database.

```
SELECT EXPRTYPE( 'SELECT LineID FROM SalesOrderItems', 1 );
```

FIRST_VALUE function [Aggregate]

Returns values from the first row of a window.

Syntax

FIRST_VALUE(*expression* [{ **RESPECT** | **IGNORE** } **NULLS**] **)**
OVER (*window-spec* **)**

window-spec : see the Remarks section below

Parameters

- **expression** The expression to evaluate. For example, a column name.

Returns

Data type of the argument.

Remarks

The FIRST_VALUE function allows you to select the first value (according to some ordering) in a table, without having to use a self-join. This is valuable when you want to use the first value as the baseline in calculations.

The FIRST_VALUE function takes the first record from the window. Then, the *expression* is computed against the first record and results are returned.

If IGNORE NULLS is specified, the first non-NULL value of *expression* is returned. If RESPECT NULLS is specified (the default), the first value is returned whether or not it is NULL.

The FIRST_VALUE function is different from most other aggregate functions in that it can only be used with a window specification.

Elements of *window-spec* can be specified either in the function syntax (inline), or in conjunction with a WINDOW clause in the SELECT statement. See the *window-spec* definition provided in “WINDOW clause” on page 793.

For more information about using window functions in SELECT statements, including working examples, see “Window functions” [*SQL Anywhere Server - SQL Usage*].

See also

- “Window aggregate functions” [*SQL Anywhere Server - SQL Usage*]
- “LAST_VALUE function [Aggregate]” on page 233

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following example returns the relationship, as a percentage, between each employee's salary and that of the most recently hired employee in the same department:

```
SELECT DepartmentID, EmployeeID,
       100 * Salary / ( FIRST_VALUE( Salary ) OVER (
                          PARTITION BY DepartmentID  ORDER BY StartDate
DESC ) )
           AS percentage
    FROM Employees;
```

DepartmentID	EmployeeID	percentage
500	1658	100
500	1615	110.4284624
500	1570	138.8427097
500	1013	109.5851905
500	921	167.4497049
500	868	113.2393688
500	750	137.7344095
500	703	222.8679276
500	191	119.6642975
400	1751	100
400	1740	99.705647
400	1684	130.969936
400	1643	83.9734797
400	1607	175.1828989
400	1576	197.0164609
...	...	...

Employee 1658 is the first row for department 500, indicating that they are the most recent hire in that department and their percentage is 100%. Percentages for the remaining department 500 employees are calculated relative to that of employee 1658. For example, employee 1570 earns approximately 139% of what employee 1658 earns.

If another employee in the same department makes the same salary as the most recent hire, they will have a percentage of 100 as well.

FLOOR function [Numeric]

Returns the largest integer not greater than a number.

Syntax

FLOOR(*numeric-expression* **)**

Parameters

- **numeric-expression** The value, usually a FLOAT.

Returns

DOUBLE

Remarks

This function converts its arguments to DOUBLE, and performs the computation in double-precision floating point.

See also

- "CEILING function [Numeric]" on page 149

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statements returns a Floor value of 123

```
SELECT FLOOR (123);
```

The following statements returns a Floor value of 123

```
SELECT FLOOR (123.45);
```

The following statements returns a Floor value of -124

```
SELECT FLOOR (-123.45);
```

GET_BIT function [Bit array]

Returns the value (1 or 0) of a specified bit in a bit array.

Syntax

GET_BIT(*bit-expression*, *position* **)**

Parameters

- **bit-expression** The bit array containing the bit.

- **position** The position of the bit for which to return the status.

Returns

BIT

Remarks

The positions in the array are counted from the left side, starting at 1.

If *position* exceeds the length of the array, 0 (false) is returned.

See also

- "Bitwise operators" on page 14
- "SET_BIT function [Bit array]" on page 300
- "SET_BITS function [Aggregate]" on page 301
- "sa_get_bits system procedure" on page 873

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value 1:

```
SELECT GET_BIT( '00110011' , 4 );
```

The following statement returns the value 0:

```
SELECT GET_BIT( '00110011' , 5 );
```

GET_IDENTITY function [Miscellaneous]

Allocates values to an autoincrement column. This is an alternative to using autoincrement to generate numbers.

Syntax

GET_IDENTITY(*table_name* [**,** *number_to_allocate*] **)**

Parameters

- **table_name** A string indicating the name of the table, including, optionally, the owner name.
- **number_to_allocate** The starting number to allocate for the identity. Default is 1.

Returns

UNSIGNED BIGINT

Remarks

Using autoincrement or global autoincrement is still the most efficient way to generate IDs, but this function is provided as an alternative. The function assumes that the table has an autoincrement column defined. It returns the next available value that would be generated for the table's autoincrement column, and reserves that value so that no other connection will use it by default.

The function returns an error if the table is not found, and returns NULL if the table has no autoincrement column. If there is more than one autoincrement column, it uses the first one it finds.

number_to_allocate is the number of values to reserve. If *number_to_allocate* is greater than 1, the function also reserves the remaining values. The next allocation uses the current number plus the value of *number_to_allocate*. This allows the application to execute the GET_IDENTITY function less frequently.

No COMMIT is required after executing the GET_IDENTITY function, and so it can be called using the same connection that is used to insert rows. If ID values are required for several tables, they can be obtained using a single SELECT that includes multiple calls to the GET_IDENTITY function, as in the example.

The GET_IDENTITY function is non-deterministic function; successive calls to it may return different values. The optimizer does not cache the results of the GET_IDENTITY function.

For more information about non-deterministic functions, see “Function caching” [*SQL Anywhere Server - SQL Usage*].

See also

- “CREATE TABLE statement” on page 523
- “ALTER TABLE statement” on page 389
- “NUMBER function [Miscellaneous]” on page 260

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the next available value for the Customers table autoincrement column (ID). The number returned and the following nine values are reserved:

```
SELECT GET_IDENTITY( 'Customers', 10 );
```

GETDATE function [Date and time]

Returns the current year, month, day, hour, minute, second and fraction of a second.

Syntax

GETDATE( )

Returns

TIMESTAMP

Remarks

The accuracy is limited by the accuracy of the system clock.

The information the GETDATE function returns is equivalent to the information returned by the NOW function and the CURRENT TIMESTAMP special value.

See also

- "NOW function [Date and time]" on page 258
- "CURRENT TIMESTAMP special value" on page 59

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the system date and time.

```
SELECT GETDATE( );
```

GRAPHICAL_PLAN function [Miscellaneous]

Returns the plan optimization strategy of a SQL statement in XML format, as a string.

Syntax

GRAPHICAL_PLAN(
string-expression
[**,** *statistics-level*
[**,** *cursor-type*
[**,** *update-status*]]] **)**

Parameters

- **string-expression** The SQL statement, which is commonly a SELECT statement but which may also be an UPDATE or DELETE statement.
- **statistics-level** An integer. *Statistics-level* can be one of the following values:

Value	Description
0	Optimizer estimates only (default).
2	Detailed statistics including node statistics.
3	Detailed statistics.

cursor-type A cursor type, expressed as a string. Possible values are: asensitive, insensitive, sensitive, or keyset-driven. If *cursor-type* is not specified, asensitive is used by default.

update-status A string parameter accepting one of the following values indicating how the optimizer should treat the given cursor:

Value	Description
READ-ONLY	The cursor is read-only.
READ-WRITE (default)	The cursor can be read or written to.
FOR UPDATE	The cursor can be read or written to. This is exactly the same as READ-WRITE.

Returns

LONG VARCHAR

See also

- "PLAN function [Miscellaneous]" on page 265
- "EXPLANATION function [Miscellaneous]" on page 204

Standards and compatibility

- **SQL/2003** Vendor extension.

Examples

The following Interactive SQL example passes a SELECT statement as a string parameter and returns the plan for executing the query. It saves the plan in the file *plan.saplan* which can be opened and read using Interactive SQL.

```
SELECT GRAPHICAL_PLAN(  'SELECT * FROM Departments WHERE DepartmentID >
100' );
OUTPUT TO 'plan.saplan' FORMAT TEXT QUOTE '' HEXADECIMAL ASIS;
```

The following statement returns a string containing the graphical plan for a keyset-driven, updatable cursor over the query `SELECT * FROM Departments WHERE DepartmentID > 100`. It also causes the server to annotate the plan with actual execution statistics, in addition to the estimated statistics that were used by the optimizer.

```
SELECT GRAPHICAL_PLAN(
   'SELECT * FROM Departments WHERE DepartmentID > 100',
   2,
   'keyset-driven', 'for update' );
```

GREATER function [Miscellaneous]

Returns the greater of two parameter values.

Syntax

GREATER(*expression-1*, *expression-2* **)**

Parameters

- **expression-1** The first parameter value to be compared.
- **expression-2** The second parameter value to be compared.

Returns

ANY

Remarks

If the parameters are equal, the first is returned.

See also

- "LESSER function [Miscellaneous]" on page 237

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value 10.

```
SELECT GREATER( 10, 5 ) FROM dummy;
```

GROUPING function [Aggregate]

Identifies whether a column in a GROUP BY operation result set is NULL because it is part of a subtotal row, or NULL because of the underlying data.

Syntax

GROUPING(*group-by-expression* **)**

Parameters

- **group-by-expression** An expression appearing as a grouping column in the result set of a query that uses a GROUP BY clause. This function can be used to identify subtotal rows added to the result set by a ROLLUP or CUBE operation.

Returns

- **1** Indicates that *group-by-expression* is NULL because it is part of a subtotal row. The column is not a prefix column for that row.
- **0** Indicates that *group-by-expression* is a prefix column of a subtotal row.

See also

- “Using ROLLUP” [*SQL Anywhere Server - SQL Usage*]
- “Using CUBE” [*SQL Anywhere Server - SQL Usage*]
- “GROUP BY GROUPING SETS” [*SQL Anywhere Server - SQL Usage*]
- “SELECT statement” on page 728
- “Detecting placeholder NULLs using the GROUPING function” [*SQL Anywhere Server - SQL Usage*]

Standards and compatibility

- **SQL/2003** SQL foundation feature (T611) outside core SQL.

Example

For examples of this function in use, see “Detecting placeholder NULLs using the GROUPING function” [*SQL Anywhere Server - SQL Usage*].

HASH function [String]

Returns the specified value in hashed form.

Syntax

HASH(*string-expression*[**,** *algorithm*] **)**

Parameters

- **string-expression** The value to be hashed. This parameter is case sensitive, even in case-insensitive databases.
- **algorithm** The algorithm to use for the hash. Possible values include: MD5, SHA1, SHA1_FIPS, SHA256, SHA256_FIPS. By default, the MD5 algorithm is used.

Note
The FIPS algorithms are only for use on systems using FIPS 140-2 certified software from Certicom.

Returns

VARCHAR

Remarks

Using a hash converts the value to a byte sequence that is unique to each value passed to the function.

If the server was started with the -fips option, the algorithm used, or the behavior, may be different, as follows:

- SHA1_FIPS is used if SHA1 is specified
- SHA256_FIPS is used if SHA256 is specified
- an error is returned if MD5 is specified

Following are the return types, depending on the algorithm used:

- MD5 returns a VARCHAR(32)
- SHA1 returns a VARCHAR(40)
- SHA1_FIPS returns a VARCHAR(40)
- SHA256 returns a VARCHAR(40)
- SHA256_FIPS returns a VARCHAR(40)

Caution
All the algorithms are one-way hashes. It is not possible to re-create the original string from the hash.

See also

- "String functions" on page 131
- "-fips server option" [*SQL Anywhere Server - Database Administration*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following example creates a table called user_info to store information about the users of an application, including their user ID and password. One row is also inserted into the table. The password is hashed using the HASH function and the SHA256 algorithm. Storing hashed passwords in this way can be useful if you do not want to store passwords in clear text, yet you have an external application that needs to compare passwords.

```
CREATE TABLE user_info (
  employee_id   INTEGER NOT NULL PRIMARY KEY,
  user_name CHAR(80),
  user_pwd CHAR(80) );
INSERT INTO user_info
  VALUES ( '1', 's_phillips', HASH( 'mypass', 'SHA256' ) );
```

HEXTOINT function [Data type conversion]

Returns the decimal integer equivalent of a hexadecimal string.

The CAST, CONVERT, HEXTOINT, and INTTOHEX functions can be used to convert to and from hexadecimal values. For more information on using these functions, see "Converting to and from hexadecimal values" on page 10.

Syntax

HEXTOINT(*hexadecimal-string* **)**

Parameters

- **hexadecimal-string** The string to be converted to an integer.

Returns

The HEXTOINT function returns the platform-independent SQL INTEGER equivalent of the hexadecimal string. The hexadecimal value represents a negative integer if the 8th digit from the right is one of the digits 8-9 and the uppercase or lowercase letters A-F and the previous leading digits are all uppercase or lowercase letter F. The following is not a valid use of HEXTOINT since the argument represents a positive integer value that cannot be represented as a signed 32-bit integer:

```
SELECT HEXTOINT( '0x0080000001' );
```

INT

Remarks

The HEXTOINT function accepts string literals or variables consisting only of digits and the uppercase or lowercase letters A-F, with or without a 0x prefix. The following are all valid uses of HEXTOINT:

```
SELECT HEXTOINT( '0xFFFFFFFF' );
SELECT HEXTOINT( '0x00000100' );
SELECT HEXTOINT( '100' );
SELECT HEXTOINT( '0xffffffff80000001' );
```

The HEXTOINT function removes the 0x prefix, if present. If the data exceeds 8 digits, it must represent a value that can be represented as a signed 32-bit integer value.

This function supports NCHAR inputs and/or outputs.

See also

- "INTTOHEX function [Data type conversion]" on page 229

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value 420.

```
SELECT HEXTOINT( '1A4' );
```

HOUR function [Date and time]

Returns the hour component of a datetime.

Syntax

HOUR(*datetime-expression* **)**

Parameters

- **datetime-expression** The datetime.

Returns

SMALLINT

Remarks

The value returned is a number from 0 to 23 corresponding to the datetime hour.

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value 21:

```
SELECT HOUR( '1998-07-09 21:12:13' );
```

HOURS function [Date and time]

A function that evaluates hours. For specific details, see this function's usage.

Syntax 1: integer

HOURS ([*datetime-expression*,] *datetime-expression* **)**

Syntax 2: timestamp

HOURS (*datetime-expression*, *integer-expression* **)**

Parameters

- **datetime-expression** A date and time.
- **integer-expression** The number of hours to be added to the *datetime-expression*. If *integer-expression* is negative, the appropriate number of hours is subtracted from the datetime. If you supply an integer expression, the *datetime-expression* must be explicitly cast as a DATETIME data type.

  For information about casting data types, see "CAST function [Data type conversion]" on page 148.

Returns

INT

TIMESTAMP

Remarks

The information you specify changes the behavior of this function:

- If you give a single date, this function returns the number of hours since 0000-02-29.

  Note
  0000-02-29 is not meant to imply an actual date; it is the date used by the date algorithm.

- If you give two time stamps, this function returns the integer number of hours between them. Instead, use the DATEDIFF function.

- If you give a date and an integer, this function adds the integer number of hours to the specified timestamp. Instead, use the DATEADD function.

See also

- "DATEDIFF function [Date and time]" on page 174
- "DATEADD function [Date and time]" on page 173

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statements return the value 4, signifying that the second timestamp is four hours after the first. It is recommended that you use the second example (DATEDIFF).

```
SELECT HOURS( '1999-07-13 06:07:12',
   '1999-07-13 10:07:12' );

SELECT DATEDIFF( hour,
   '1999-07-13 06:07:12',
   '1999-07-13 10:07:12' );
```

The following statement returns the value 17517342.

```
SELECT HOURS( '1998-07-13 06:07:12' );
```

The following statements return the datetime 1999-05-13 02:05:07.000. It is recommended that you use the second example (DATEADD).

```
SELECT HOURS(
   CAST( '1999-05-12 21:05:07' AS DATETIME ), 5 );

SELECT DATEADD( hour, 5, '1999-05-12 21:05:07' );
```

HTML_DECODE function [Miscellaneous]

Decodes special character entities that appear in HTML literal strings.

Syntax

HTML_DECODE(*string* **)**

Parameters

- **string** Arbitrary literal string used in an HTML document.

Returns

LONG VARCHAR

LONG NVARCHAR

Remarks

This function returns the string argument after making the following set of substitutions:

Characters	**Substitution**
&quot;	"
&#39;	'
&amp;	&
&lt;	<
&gt;	>
&#x*hexadecimal-number*;	Unicode codepoint, specified as a hexadecimal number. For example, &#x27; returns a single apostrophe.
&#*decimal-number*;	Unicode codepoint, specified as a decimal number. For example, &#8482; returns the trademark symbol.

When a Unicode codepoint is specified, if the value can be converted to a character in the database character set, it is converted to a character. Otherwise, it is returned uninterpreted.

SQL Anywhere supports all character entity references specified in the HTML 4.01 Specification. See http://www.w3.org/TR/html4/.

See also

- "-xs server option" [*SQL Anywhere Server - Database Administration*]
- "SQL Anywhere web services" [*SQL Anywhere Server - Programming*]
- "HTML_ENCODE function [Miscellaneous]" on page 219
- "Web services functions" on page 130
- "Web services system procedures" on page 830

Standards and compatibility

- **SQL/2003** Vendor extension.

HTML_ENCODE function [Miscellaneous]

Encodes special characters within strings to be inserted into HTML documents.

Syntax

HTML_ENCODE(*string* **)**

Parameters

- **string** Arbitrary string to be used in an HTML document.

Returns

LONG VARCHAR

LONG NVARCHAR

Remarks

This function returns the string argument after making the following set of substitutions:

Characters	Substitution
"	&quot;
'	&#39;
&	&amp;
<	&lt;
>	&gt;
codes *nn* less than 0x20	&#x*nn*;

This function supports NCHAR inputs and/or outputs.

See also

- "-xs server option" [*SQL Anywhere Server - Database Administration*]
- "SQL Anywhere web services" [*SQL Anywhere Server - Programming*]
- "HTML_DECODE function [Miscellaneous]" on page 218
- "Web services functions" on page 130
- "Web services system procedures" on page 830

Standards and compatibility

- **SQL/2003** Vendor extension.

HTTP_DECODE function [HTTP]

Decodes special characters within strings for use with HTTP.

Syntax

HTTP_DECODE(*string* **)**

Parameters

- **string** Arbitrary string to be used in an HTTP request.

Returns

LONG VARCHAR

LONG NVARCHAR

Remarks

This function returns the string argument after replacing all character sequences of the form %*nn*, where *nn* is a hexadecimal value, with the character with code *nn*. In addition, all plus signs (+) are replaced with spaces.

See also

- "-xs server option" [*SQL Anywhere Server - Database Administration*]
- "SQL Anywhere web services" [*SQL Anywhere Server - Programming*]
- "HTTP_ENCODE function [HTTP]" on page 221
- "HTTP_HEADER function [HTTP]" on page 223
- "HTTP_VARIABLE function [HTTP]" on page 225
- "NEXT_HTTP_HEADER function [HTTP]" on page 255
- "NEXT_HTTP_VARIABLE function [HTTP]" on page 256
- "Web services functions" on page 130
- "Web services system procedures" on page 830

Standards and compatibility

- **SQL/2003** Vendor extension.

HTTP_ENCODE function [HTTP]

Encodes special characters in strings for use with HTTP.

Syntax

HTTP_ENCODE(*string* **)**

Parameters

- **string** Arbitrary string to be used in an HTTP request.

Returns

LONG VARCHAR

LONG NVARCHAR

Remarks

This function returns the string argument after making the following set of substitutions. In addition, all characters with hexadecimal codes less than 20 or greater than 7E are replaced with %*nn*, where *nn* is the character code.

Character	Substitution
space	%20
"	%22

Character	Substitution
#	%23
%	%25
&	%26
,	%2C
;	%3B
<	%3C
>	%3E
[	%5B
\	%5C
]	%5D
`	%60
{	%7B
|	%7C
}	%7D
character codes *nn* that are less than 0x20 and greater than 0x7f	%*nn*

This function supports NCHAR inputs and/or outputs.

See also

- "-xs server option" [*SQL Anywhere Server - Database Administration*]
- "SQL Anywhere web services" [*SQL Anywhere Server - Programming*]
- "HTTP_DECODE function [HTTP]" on page 220
- "HTTP_HEADER function [HTTP]" on page 223
- "HTTP_VARIABLE function [HTTP]" on page 225
- "NEXT_HTTP_HEADER function [HTTP]" on page 255
- "NEXT_HTTP_VARIABLE function [HTTP]" on page 256
- "Web services functions" on page 130
- "Web services system procedures" on page 830

Standards and compatibility

- **SQL/2003** Vendor extension.

HTTP_BODY function [HTTP]

Returns the body of the HTTP request in binary form. For example, in a POST request, this is the raw POST data.

Syntax

HTTP_BODY( )

Parameters

None

Returns

LONG VARCHAR

The HTTP request is returned in binary form; no character set conversion is performed on it.

Remarks

If the request body does not exist, or if the function is not called from a web service, a NULL value is returned.

See also

- "SQL Anywhere web services" [*SQL Anywhere Server - Programming*]
- "-xs server option" [*SQL Anywhere Server - Database Administration*]
- "sa_http_php_page system procedure" on page 884
- "sa_http_php_page_interpreted system procedure" on page 885
- "sa_http_header_info system procedure" on page 883
- "sa_set_http_header system procedure" on page 944
- "sa_set_http_option system procedure" on page 946
- "Web services functions" on page 130
- "Web services system procedures" on page 830

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

For more information about HTTP requests and web services, see "SQL Anywhere web services" [*SQL Anywhere Server - Programming*].

HTTP_HEADER function [HTTP]

Returns the value of an HTTP header.

Syntax

HTTP_HEADER(*header-field-name* **)**

Parameters

- **header-field-name** The name of an HTTP header field.

Returns

LONG VARCHAR

Remarks

This function returns the value of the named HTTP header field, or NULL if not called from an HTTP service. It is used when processing an HTTP request via a web service.

If a header for the given *header-field-name* does not exist, the return value is NULL. The return value is also NULL when the function is not called from a web service.

Some headers that may be of interest when processing an HTTP web service request include the following. More information about these headers is available at http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html.org/Protocols/rfc2616/rfc2616-sec14.html.

- **Cookie** The cookie value(s), if any, stored by the client, that are associated with the requested URI.
- **Referer** The URL of the page that contained the link to the requested URI.
- **Host** The name or IP of the host that submitted the request.
- **User-Agent** The name of the client application.
- **Accept-Encoding** A list of encodings for the response that are acceptable to the client application.

These special headers are always defined when processing an HTTP web service request:

- **@HttpMethod** Returns the type of request being processed. Possible values include HEAD, GET, or POST.
- **@HttpURI** The full URI of the request, as it was specified in the HTTP request.
- **@HttpVersion** The HTTP version of the request (for example, 1.0, or 1.1).
- **@HttpQueryString** Returns the query portion of the requested URI if it exists.

These special headers allow access to the first line of a client request (also known as the request line).

See also

- "-xs server option" [*SQL Anywhere Server - Database Administration*]
- "sa_set_http_header system procedure" on page 944
- "Working with HTTP headers" [*SQL Anywhere Server - Programming*]
- "SQL Anywhere web services" [*SQL Anywhere Server - Programming*]
- "HTTP_DECODE function [HTTP]" on page 220
- "HTTP_ENCODE function [HTTP]" on page 221
- "HTTP_HEADER function [HTTP]" on page 223
- "HTTP_VARIABLE function [HTTP]" on page 225
- "NEXT_HTTP_HEADER function [HTTP]" on page 255
- "NEXT_HTTP_VARIABLE function [HTTP]" on page 256
- "Web services functions" on page 130
- "Web services system procedures" on page 830

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

When used within a stored procedure that is called by an HTTP web service, the following example gets the Cookie header value:

```
SET cookie_value = HTTP_HEADER( 'Cookie' );
```

When used within a stored procedure that is called by an HTTP web service, the following example returns the value of the first HTTP header.

```
DECLARE header_name LONG VARCHAR;
DECLARE header_value LONG VARCHAR;
SET header_name = NEXT_HTTP_HEADER( NULL );
SET header_value = HTTP_HEADER( header_name );
```

HTTP_VARIABLE function [HTTP]

Returns the value of an HTTP variable.

Syntax

HTTP_VARIABLE(*var-name* [[, *instance*] , *http-header-field*] **)**

Parameters

- **var-name** The name of an HTTP variable.
- **instance** If more than one variable has the same name, the instance number of the field instance, or NULL to get the first one. Useful for select lists that permit multiple selections.
- **http-header-field** In a multi-part request, a header field name associated with the named field as specified in *var-name*.

Returns

This function returns the value of the named HTTP variable. It is used when processing an HTTP request within a web service.

LONG VARCHAR

Remarks

If a header for the given *var-name* does not exist, the return value is NULL.

When the web service request is a POST, and the variable data is posted as multipart/form-data, the HTTP server receives HTTP headers for each individual variable. When the *http-header-field* parameter is specified, the HTTP_VARIABLE function returns the associated multipart/form-data header value from the POST request for the particular variable.

All input data goes through character set translation between the client (for example, a browser) character set, and the character set of the database. However, if @BINARY is specified for *http-header-field*, the variable input value is returned without going through character set translation. This may be useful when receiving binary data, such as image data, from a client.

This function returns NULL when not called from a web service.

See also

- "-xs server option" [*SQL Anywhere Server - Database Administration*]
- "Working with variables" [*SQL Anywhere Server - Programming*]
- "HTTP_DECODE function [HTTP]" on page 220
- "HTTP_ENCODE function [HTTP]" on page 221
- "HTTP_HEADER function [HTTP]" on page 223
- "NEXT_HTTP_HEADER function [HTTP]" on page 255
- "NEXT_HTTP_VARIABLE function [HTTP]" on page 256
- "SQL Anywhere web services" [*SQL Anywhere Server - Programming*]
- "Web services functions" on page 130
- "Web services system procedures" on page 830

Standards and compatibility

- **SQL/2003** Vendor extension.

Examples

When used within a stored procedure that is called by an HTTP web service, the following statements request the Content-Disposition and Content-Type headers of the image variable:

```
SET v_name = HTTP_VARIABLE( 'image', NULL, 'Content-Disposition' );
SET v_type = HTTP_VARIABLE( 'image', NULL, 'Content-Type' );
```

When used within a stored procedure that is called by an HTTP web service, the following statement requests the value of the image variable in its current character set, that is, without going through character set translation:

```
SET v_image = HTTP_VARIABLE( 'image', NULL, '@BINARY' );
```

IDENTITY function [Miscellaneous]

Generates integer values, starting at 1, for each successive row in a query. Its implementation is identical to that of the NUMBER function.

Syntax

IDENTITY(*expression* **)**

Parameters

- **expression** An expression. The expression is parsed, but is ignored during the execution of the function.

Returns

INT

Remarks

For a description of how to use the IDENTITY function, see "NUMBER function [Miscellaneous]" on page 260.

See also

- "NUMBER function [Miscellaneous]" on page 260

Standards and compatibility

- **SQL/2003** Transact-SQL extension.

Example

The following statement returns a sequentially-numbered list of employees.

```
SELECT IDENTITY( 10 ), Surname FROM Employees;
```

IFNULL function [Miscellaneous]

If the first expression is the NULL value, then the value of the second expression is returned. If the first expression is not NULL, the value of the third expression is returned. If the first expression is not NULL and there is no third expression, NULL is returned.

Syntax

IFNULL(*expression-1*, *expression-2* [, *expression-3*] **)**

Parameters

- **expression-1** The expression to be evaluated. Its value determines whether *expression-2* or *expression-3* is returned.
- **expression-2** The return value if *expression-1* is NULL.

- **expression-3** The return value if *expression-1* is not NULL.

Returns

The data type returned depends on the data type of *expression-2* and *expression-3*.

Standards and compatibility

- **SQL/2003** Transact-SQL extension.

Example

The following statement returns the value -66.

```
SELECT IFNULL( NULL, -66 );
```

The following statement returns NULL, because the first expression is not NULL and there is no third expression.

```
SELECT IFNULL( -66, -66 );
```

INDEX_ESTIMATE function [Miscellaneous]

Returns selectivity estimates from the index as a percentage calculated by the query optimizer, based on specified parameters.

Syntax

INDEX_ESTIMATE(*column-name* [**,** *value* [**,** *relation-string*]] **)**

Parameters

- **column-name** The column used in the estimate.
- **value** The value to which the column is compared. The default is NULL.
- **relation-string** The comparison operator used for the comparison, enclosed in single quotes. Possible values for this parameter are: '=' , '>' , '<' , '>=' , '<=' , '<>' , '!=' , '!<' , and '!>'. The default is '='.

Returns

REAL

Remarks

If *value* is NULL then the relation strings '=' and '!=' are interpreted as the IS NULL and IS NOT NULL conditions, respectively.

See also

- "ESTIMATE function [Miscellaneous]" on page 195
- "ESTIMATE_SOURCE function [Miscellaneous]" on page 196
- "EXPERIENCE_ESTIMATE function [Miscellaneous]" on page 203

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the percentage of EmployeeID values estimated to be greater than 200.

```
SELECT INDEX_ESTIMATE( EmployeeID, 200, '>' )
FROM Employees;
```

INSERTSTR function [String]

Inserts a string into another string at a specified position.

Syntax

INSERTSTR(*integer-expression*, *string-expression-1*, *string-expression-2* **)**

Parameters

- **integer-expression** The position after which the string is to be inserted. Use zero to insert a string at the beginning.
- **string-expression-1** The string into which the other string is to be inserted.
- **string-expression-2** The string to be inserted.

Returns

LONG VARCHAR

Remarks

This function supports NCHAR inputs and/or outputs.

See also

- “STUFF function [String]” on page 317
- “String functions” on page 131

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value backoffice.

```
SELECT INSERTSTR( 0, 'office ', 'back' );
```

INTTOHEX function [Data type conversion]

Returns a string containing the hexadecimal equivalent of an integer.

Syntax

INTTOHEX(*integer-expression* **)**

Parameters

- **integer-expression** The integer to be converted to hexadecimal.

Returns

VARCHAR

Remarks

The CAST, CONVERT, HEXTOINT, and INTTOHEX functions can be used to convert to and from hexadecimal values. For more information, see "Converting to and from hexadecimal values" on page 10.

See also

- "HEXTOINT function [Data type conversion]" on page 215

Standards and compatibility

- **SQL/2003** Transact-SQL extension.

Example

The following statement returns the value 0000009c.

```
SELECT INTTOHEX( 156 );
```

ISDATE function [Data type conversion]

Tests if a string argument can be converted to a date.

Syntax

ISDATE(*string* **)**

Parameters

- **string** The string to be analyzed to determine if the string represents a valid date.

Returns

INT

Remarks

If a conversion is possible, the function returns 1; otherwise, 0 is returned. If the argument is NULL, 0 is returned.

This function supports NCHAR inputs and/or outputs.

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following example imports data from an external file, exports rows which contain invalid values, and copies the remaining rows to a permanent table.

```
CREATE GLOBAL TEMPORARY TABLE MyData(
      person VARCHAR(100),
      birth_date VARCHAR(30),
      height_in_cms VARCHAR(10)
   ) ON COMMIT PRESERVE ROWS;
   LOAD TABLE MyData FROM 'exported.dat';
   UNLOAD
      SELECT * FROM MyData
      WHERE ISDATE( birth_date ) = 0
  OR ISNUMERIC( height_in_cms ) = 0
   TO 'badrows.dat';
   INSERT INTO PermData
      SELECT person, birth_date, height_in_cms
      FROM MyData
      WHERE ISDATE( birth_date ) = 1
  AND ISNUMERIC( height_in_cms ) = 1;
   COMMIT;
   DROP TABLE MyData;
```

ISNULL function [Miscellaneous]

Returns the first non-NULL expression from a list. This function is identical to the COALESCE function.

Syntax

ISNULL(*expression*, *expression* [, ...] **)**

Parameters

- **expression** An expression to be tested against NULL.

  At least two expressions must be passed into the function, and all expressions must be comparable.

Returns

The return type for this function depends on the expressions specified. That is, when the database server evaluates the function, it first searches for a data type in which all the expressions can be compared. When found, the database server compares the expressions and then returns the result in the type used for the comparison. If the database server cannot find a common comparison type, an error is returned.

See also

- "COALESCE function [Miscellaneous]" on page 152

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value -66.

```
SELECT ISNULL( NULL ,-66, 55, 45, NULL, 16 );
```

ISNUMERIC function [Miscellaneous]

Determines if a string argument is a valid number.

Syntax

ISNUMERIC(*string* **)**

Parameters

- **string** The string to be analyzed to determine if the string represents a valid number.

Returns

INT

Remarks

ISNUMERIC returns 1 when the input string evaluates to a valid integer or floating point number; otherwise it returns 0. The function also returns 0 if the string contains only blanks or is NULL.

Following are values that also cause the ISNUMERIC function to return 0:

- Values that use the letter d or D as the exponent separator. For example, 1d2.
- Special values such as NAN, 0x12, INF, and INFINITY.
- NULL (for example, `SELECT ISNUMERIC( NULL );`)

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following example imports data from an external file, exports rows that contain invalid values, and copies the remaining rows to a permanent table. In this example, the ISNUMERIC statement validates that the values in height_in_cms values are numeric.

```
CREATE GLOBAL TEMPORARY TABLE MyData(
      person VARCHAR(100),
      birth_date VARCHAR(30),
      height_in_cms VARCHAR(10)
   ) ON COMMIT PRESERVE ROWS;
   LOAD TABLE MyData FROM 'exported.dat';
   UNLOAD
      SELECT *
      FROM MyData
      WHERE ISDATE( birth_date ) = 0
  OR ISNUMERIC( height_in_cms ) = 0
   TO 'badrows.dat';
```

```
 INSERT INTO PermData
    SELECT person, birth_date, height_in_cms
    FROM MyData
    WHERE ISDATE( birth_date ) = 1
AND ISNUMERIC( height_in_cms ) = 1;
 COMMIT;
 DROP TABLE MyData;
```

LAST_VALUE function [Aggregate]

Returns values from the last row of a window.

Syntax

LAST_VALUE(*expression*[{ **RESPECT** | **IGNORE** } **NULLS**] **)**
OVER (*window-spec* **)**

window-spec : see the Remarks section below

Parameters

- **expression** The expression to evaluate. For example, a column name.

Returns

Data type of the argument.

Remarks

The LAST_VALUE function allows you to select the last value (according to some ordering) in a table, without having to use a self-join. This is valuable when you want to use the last value as the baseline in calculations.

The LAST_VALUE function takes the last record from the partition after doing the ORDER BY. Then, the *expression* is computed against the last record and results are returned.

If IGNORE NULLS is specified, the last non-NULL value of *expression* is returned. If RESPECT NULLS is specified (the default), the last value is returned whether or not it is NULL.

The LAST_VALUE function is different from most other aggregate functions in that it can only be used with a window specification.

Elements of *window-spec* can be specified either in the function syntax (inline), or in conjunction with a WINDOW clause in the SELECT statement. See the *window-spec* definition provided in "WINDOW clause" on page 793.

For more information about using window functions in SELECT statements, including working examples, see "Window functions" [*SQL Anywhere Server - SQL Usage*].

See also

- "Window aggregate functions" [*SQL Anywhere Server - SQL Usage*]
- "FIRST_VALUE function [Aggregate]" on page 206

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following example returns the salary of each employee, plus the name of the employee with the highest salary in the same department:

```
SELECT GivenName + ' ' + Surname AS employee_name,
       Salary, DepartmentID,
       LAST_VALUE( employee_name ) OVER Salary_Window AS highest_paid
FROM Employees
WINDOW Salary_Window AS ( PARTITION BY DepartmentID ORDER BY Salary
                          RANGE BETWEEN UNBOUNDED PRECEDING
                          AND UNBOUNDED FOLLOWING );
```

employee_name	Salary	DepartmentID	highest_paid
Michael Lynch	24903	500	Jose Martinez
Joseph Barker	27290	500	Jose Martinez
Sheila Romero	27500	500	Jose Martinez
Felicia Kuo	28200	500	Jose Martinez
Jeannette Bertrand	29800	500	Jose Martinez
Jane Braun	34300	500	Jose Martinez
Anthony Rebeiro	34576	500	Jose Martinez
Charles Crowley	41700	500	Jose Martinez
Jose Martinez	55500.8	500	Jose Martinez
Doug Charlton	28300	400	Scott Evans
Elizabeth Lambert	29384	400	Scott Evans
Joyce Butterfield	34011	400	Scott Evans
Robert Nielsen	34889	400	Scott Evans
Alex Ahmed	34992	400	Scott Evans
Ruth Wetherby	35745	400	Scott Evans
...	...	...	...

Jose Martinez makes the highest salary in department 500, and Scott Evans makes the highest salary in department 400.

LCASE function [String]

Converts all characters in a string to lowercase. This function is identical the LOWER function.

Syntax

LCASE(*string-expression* **)**

Parameters

- **string-expression** The string to be converted to lowercase.

Returns

CHAR

NCHAR

LONG VARCHAR

VARCHAR

NVARCHAR

Remarks

The LCASE function is similar to the LOWER function.

See also

- "LOWER function [String]" on page 243
- "UCASE function [String]" on page 330
- "UPPER function [String]" on page 333
- "String functions" on page 131

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value chocolate.

```
SELECT LCASE( 'ChoCOlatE' );
```

LEFT function [String]

Returns multiple characters from the beginning of a string.

Syntax

LEFT(*string-expression*, *integer-expression* **)**

Parameters

- **string-expression** The string.
- **integer-expression** The number of characters to return.

Returns

LONG VARCHAR

LONG NVARCHAR

Remarks

If the string contains multibyte characters, and the proper collation is being used, the number of bytes returned may be greater than the specified number of characters.

You can specify an *integer-expression* that is larger than the value in the argument string expression. In this case, the entire value is returned.

This function supports NCHAR inputs and/or outputs. Whenever possible, if the input string uses character length semantics the return value will be described in terms of character length semantics.

See also

- "RIGHT function [String]" on page 293
- "String functions" on page 131

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the first 5 characters of each Surname value in the Customers table.

```
SELECT LEFT( Surname, 5) FROM Customers;
```

LENGTH function [String]

Returns the number of characters in the specified string.

Syntax

LENGTH(*string-expression* **)**

Parameters

- **string-expression** The string.

Returns

INT

Remarks

Use this function to determine the length of a string. For example, specify a column name for *string-expression* to determine the length of values in the column.

If the string contains multibyte characters, and the proper collation is being used, LENGTH returns the number of characters, not the number of bytes. If the string is of data type BINARY, the LENGTH function behaves as the BYTE_LENGTH function.

Note
You can use the LENGTH function and the CHAR_LENGTH function interchangeably for CHAR, VARCHAR, LONG VARCHAR, and NCHAR data types. However, you must use the LENGTH function for BINARY and bit array data types.

This function supports NCHAR inputs and/or outputs.

See also

- "BYTE_LENGTH function [String]" on page 146
- "International languages and character sets" [*SQL Anywhere Server - Database Administration*]
- "String functions" on page 131

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value 9.

```
SELECT LENGTH( 'chocolate' );
```

LESSER function [Miscellaneous]

Returns the lesser of two parameter values.

Syntax

LESSER(*expression-1*, *expression-2* **)**

Parameters

- **expression-1** The first parameter value to be compared.
- **expression-2** The second parameter value to be compared.

Returns

The return type for this function depends on the expressions specified. That is, when the database server evaluates the function, it first searches for a data type in which all the expressions can be compared. When found, the database server compares the expressions and then returns the result in the type used for the comparison. If the database server cannot find a common comparison type, an error is returned.

Remarks

If the parameters are equal, the first value is returned.

See also

- "GREATER function [Miscellaneous]" on page 212

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value 5.

```
SELECT LESSER( 10, 5 ) FROM dummy;
```

LIST function [Aggregate]

Returns a comma-delimited list of values.

Syntax

LIST(
[**DISTINCT**] *string-expression*
[**,** *delimiter-string*]
[**ORDER BY** *order-by-expression* [**ASC** | **DESC**], ...] **)**

Parameters

- **string-expression** A string expression, usually a column name. For each row in the column, the value is added to a comma-separated list. When DISTINCT is specified, only unique values are added.
- **delimiter-string** A delimiter string for the list items. The default setting is a comma. There is no delimiter if a value of NULL or an empty string is supplied. The *delimiter-string* must be a constant.
- **order-by-expression** Order the items returned by the function. There is no comma preceding this argument, which makes it easy to use in the case where no *delimiter-string* is supplied.

  order-by-expression cannot be an integer literal. However, it can be a variable that contains an integer literal. Also, multiple LIST functions in the same query block are not allowed to use different *order-by-expression* arguments.

Returns

LONG VARCHAR

LONG NVARCHAR

Remarks

NULL values are not added to the list. LIST (X) returns the concatenation (with delimiters) of all the non-NULL values of X for each row in the group. If there does not exist at least one row in the group with a definite X-value, then LIST(X) returns the empty string.

A LIST function cannot be used as a window function, but it can be used as input to a window function.

This function supports NCHAR inputs and/or outputs.

Standards and compatibility

- **SQL/2003** Vendor extension.

See also

- "sa_split_list system procedure" on page 951

Examples

The following statement returns the value 487 Kennedy Court, 547 School Street.

```
SELECT LIST( Street ) FROM Employees
WHERE GivenName = 'Thomas';
```

The following statement lists employee IDs. Each row in the result set contains a comma-delimited list of employee IDs for a single department.

```
SELECT LIST( EmployeeID )
FROM Employees
GROUP BY DepartmentID;
```

LIST(EmployeeID)
102,105,160,243,247,249,266,278,...
129,195,299,467,641,667,690,856,...
148,390,586,757,879,1293,1336,...
184,207,318,409,591,888,992,1062,...
191,703,750,868,921,1013,1570,...

The following statement sorts the employee IDs by the last name of the employee:

```
SELECT LIST( EmployeeID ORDER BY Surname ) AS "Sorted IDs"
FROM Employees
GROUP BY DepartmentID;
```

Sorted IDs '1751,591,1062,1191,992,888,318,184,1576,207,1684,1643,1607,1740,409,1507'

Sorted IDs
1013,191,750,921,868,1658,...
1751,591,1062,1191,992,888,318,...
1336,879,586,390,757,148,1483,...

Sorted IDs
1039,129,1142,195,667,1162,902,...
160,105,1250,247,266,249,445,...

The following statement returns semicolon-separated lists. Note the position of the ORDER BY clause and the list separator:

```
SELECT LIST( EmployeeID, ';' ORDER BY Surname ) AS "Sorted IDs"
FROM Employees
GROUP BY DepartmentID;
```

Sorted IDs
1013;191;750;921;868;1658;703;...
1751;591;1062;1191;992;888;318;...
1336;879;586;390;757;148;1483;...
1039;129;1142;195;667;1162;902; ...
160;105;1250;247;266;249;445;...

Be sure to distinguish the previous statement from the following statement, which returns comma-separated lists of employee IDs sorted by a compound sort-key of `( Surname, ';' )`:

```
SELECT LIST( EmployeeID ORDER BY Surname, ';' ) AS "Sorted IDs"
FROM Employees
GROUP BY DepartmentID;
```

LOCATE function [String]

Returns the position of one string within another.

Syntax

LOCATE(*string-expression-1*, *string-expression-2* [, *integer-expression*] **)**

Parameters

- **string-expression-1** The string to be searched.
- **string-expression-2** The string to be searched for. This string is limited to 255 bytes.
- **integer-expression** The character position in the string to begin the search. The first character is position 1. If the starting offset is negative, the locate function returns the last matching string offset rather than the first. A negative offset indicates how much of the end of the string is to be excluded from the search. The number of bytes excluded is calculated as (-1 * offset) -1.

Returns

INT

Remarks

If *integer-expression* is specified, the search starts at that offset into the string.

The first string can be a long string (longer than 255 bytes), but the second is limited to 255 bytes. If a long string is given as the second argument, the function returns a NULL value. If the string is not found, 0 is returned. Searching for a zero-length string will return 1. If any of the arguments are NULL, the result is NULL.

If multibyte characters are used, with the appropriate collation, then the starting position and the return value may be different from the *byte* positions.

This function supports NCHAR inputs and/or outputs.

See also

- "String functions" on page 131
- "CHARINDEX function [String]" on page 150

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value 8.

```
SELECT LOCATE(
   'office party this week - rsvp as soon as possible',
   'party',
   2 );
```

The following statement:

```
BEGIN
   DECLARE STR LONG VARCHAR;
   DECLARE POS INT;
   SET str = 'c:\test\functions\locate.sql';
   SET pos = LOCATE( str, '\', -1 );
   select str, pos,
      SUBSTR( str, 1, pos -1 ) AS path,
      SUBSTR( str, pos +1 ) AS filename;
END;
```

returns the following output:

str	pos	path	filename
c:\test\functions\locate.sql	18	c:\test\functions	locate.sql

LOG function [Numeric]

Returns the natural logarithm of a number.

Syntax

LOG(*numeric-expression* **)**

Parameters

- **numeric-expression** The number.

Returns

This function converts its argument to DOUBLE, performs the computation in double-precision floating point, and returns a DOUBLE as the result. If the parameter is NULL, the result is NULL.

Remarks

The argument is an expression that returns the value of any built-in numeric data type.

See also

- "LOG10 function [Numeric]" on page 242

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the natural logarithm of 50.

```
SELECT LOG( 50 );
```

LOG10 function [Numeric]

Returns the base 10 logarithm of a number.

Syntax

LOG10(*numeric-expression* **)**

Parameters

- **numeric-expression** The number.

Returns

This function converts its argument to DOUBLE, and performs the computation in double-precision floating point. If the parameter is NULL, the result is NULL.

Remarks

The argument is an expression that returns the value of any built-in numeric data type.

See also

- “LOG function [Numeric]” on page 242

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the base 10 logarithm for 50.

```
SELECT LOG10( 50 );
```

LOWER function [String]

Converts all characters in a string to lowercase. This function is identical the LCASE function.

Syntax

LOWER(*string-expression* **)**

Parameters

- **string-expression** The string to be converted.

Returns

CHAR

NCHAR

LONG VARCHAR

VARCHAR

NVARCHAR

Remarks

The LCASE function is identical to the LOWER function.

See also

- “LCASE function [String]” on page 235
- “UCASE function [String]” on page 330
- “UPPER function [String]” on page 333
- “String functions” on page 131

Standards and compatibility

- **SQL/2003** Core feature.

Example

The following statement returns the value chocolate.

```
SELECT LOWER( 'chOCOLate' );
```

LTRIM function [String]

Removes leading and trailing blanks from a string.

Syntax

LTRIM(*string-expression* **)**

Parameters

- **string-expression** The string to be trimmed.

Returns

VARCHAR

NVARCHAR

LONG VARCHAR

LONG NVARCHAR

Remarks

The actual length of the result is the length of the expression minus the number of characters removed. If all the characters are removed, the result is an empty string.

If the parameter can be null, the result can be null.

If the parameter is null, the result is the null value.

This function supports NCHAR inputs and/or outputs.

See also

- "RTRIM function [String]" on page 297
- "TRIM function [String]" on page 327
- "String functions" on page 131

Standards and compatibility

- **SQL/2003** Vendor extension.

  The TRIM specifications defined by the SQL/2003 standard (LEADING and TRAILING) are supplied by the SQL Anywhere LTRIM and RTRIM functions respectively.

Example

The following statement returns the value Test Message with all leading blanks removed.

```
SELECT LTRIM( '     Test Message' );
```

MAX function [Aggregate]

Returns the maximum *expression* value found in each group of rows.

Syntax 1

MAX(*expression* | **DISTINCT** *expression* **)**

Syntax 2

MAX(*expression* **) OVER (** *window-spec* **)**

window-spec : see Syntax 2 instructions in the Remarks section below

Parameters

- **expression** The expression for which the maximum value is to be calculated. This is commonly a column name.
- **DISTINCT expression** Returns the same as MAX(*expression*), and is included for completeness.

Returns

The same data type as the argument.

Remarks

Rows where *expression* is NULL are ignored. Returns NULL for a group containing no rows.

Syntax 2 represents usage as a window function in a SELECT statement. As such, elements of *window-spec* can be specified either in the function syntax (inline), or in conjunction with a WINDOW clause in the SELECT statement. See the *window-spec* definition provided in "WINDOW clause" on page 793.

For more information about using window functions in SELECT statements, including working examples, see "Window functions" [*SQL Anywhere Server - SQL Usage*].

This function supports NCHAR inputs and/or outputs.

See also

- "MIN function [Aggregate]" on page 246

Standards and compatibility

- **SQL/2003** Core feature. Syntax 2 is feature T611.

Example

The following statement returns the value 138948.000, representing the maximum salary in the Employees table.

```
SELECT MAX( Salary )
FROM Employees;
```

MIN function [Aggregate]

Returns the minimum expression value found in each group of rows.

Syntax 1

MIN(*expression* | **DISTINCT** *expression* **)**

Syntax 2

MIN(*expression* **) OVER (** *window-spec* **)**

window-spec : see Syntax 2 instructions in the Remarks section below

Parameters

- **expression** The expression for which the minimum value is to be calculated. This is commonly a column name.
- **DISTINCT expression** Returns the same as MIN(*expression*), and is included for completeness.

Returns

The same data type as the argument.

Remarks

Rows where *expression* is NULL are ignored. Returns NULL for a group containing no rows.

Syntax 2 represents usage as a window function in a SELECT statement. As such, elements of *window-spec* can be specified either in the function syntax (inline), or in conjunction with a WINDOW clause in the SELECT statement. See the *window-spec* definition provided in "WINDOW clause" on page 793.

For more information about using window functions in SELECT statements, including working examples, see "Window functions" [*SQL Anywhere Server - SQL Usage*].

This function supports NCHAR inputs and/or outputs.

See also

- "MAX function [Aggregate]" on page 245

Standards and compatibility

- **SQL/2003** Core feature. Syntax 2 is feature T611.

Example

The following statement returns the value 24903.000, representing the minimum salary in the Employees table.

```
SELECT MIN( Salary )
FROM Employees;
```

MINUTE function [Date and time]

Returns a minute component of a datetime value.

Syntax

MINUTE(*datetime-expression* **)**

Parameters

- **datetime-expression** The datetime value.

Returns

SMALLINT

Remarks

The value returned is a number from number from 0 to 59 corresponding to the datetime minute.

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value 22.

```
SELECT MINUTE( '1998-07-13 12:22:34' );
```

MINUTES function [Date and time]

The information you specify changes the behavior of this function:

- If you give a single date, this function returns the number of minutes since 0000-02-29.

  Note
  0000-02-29 is not meant to imply an actual date; it is the date used by the date algorithm.

- If you give two time stamps, this function returns the integer number of minutes between them. Instead, use the DATEDIFF function.
- If you give a date and an integer, this function adds the integer number of minutes to the specified timestamp. Instead, use the DATEADD function.

Syntax 1: integer

MINUTES([*datetime-expression*,] *datetime-expression* **)**

Syntax 2: timestamp

MINUTES(*datetime-expression*, *integer-expression* **)**

Parameters

- **datetime-expression** A date and time.
- **integer-expression** The number of minutes to be added to the *datetime-expression*. If *integer-expression* is negative, the appropriate number of minutes is subtracted from the datetime value. If you supply an integer expression, the *datetime-expression* must be explicitly cast as a DATETIME data type.

Returns

INT

TIMESTAMP

Remarks

Since this function returns an integer, overflow can occur when syntax 1 is used with time stamps greater than or equal to 4083-03-23 02:08:00.

See also

- "CAST function [Data type conversion]" on page 148

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statements return the value 240, signifying that the second timestamp is 240 minutes after the first. It is recommended that you use the second example (DATEDIFF).

```
SELECT MINUTES( '1999-07-13 06:07:12',
    '1999-07-13 10:07:12' );

SELECT DATEDIFF( minute,
     '1999-07-13 06:07:12',
     '1999-07-13 10:07:12' );
```

The following statement returns the value 1051040527.

```
SELECT MINUTES( '1998-07-13 06:07:12' );
```

The following statements return the timestamp 1999-05-12 21:10:07.000. It is recommended that you use the second example (DATEADD).

```
SELECT MINUTES( CAST( '1999-05-12 21:05:07'
AS DATETIME ), 5);

SELECT DATEADD( minute, 5, '1999-05-12 21:05:07' );
```

MOD function [Numeric]

Returns the remainder when one whole number is divided by another.

Syntax

MOD(*dividend*, *divisor* **)**

Parameters

- **dividend** The dividend, or numerator of the division.
- **divisor** The divisor, or denominator of the division.

Returns

SMALLINT

INT

NUMERIC

Remarks

Division involving a negative dividend gives a negative or zero result. The sign of the divisor has no effect.

See also

- "REMAINDER function [Numeric]" on page 288

Standards and compatibility

- **SQL/2003** SQL foundation feature outside core SQL.

Example

The following statement returns the value 2.

```
SELECT MOD( 5, 3 );
```

MONTH function [Date and time]

Returns a month of the given date.

Syntax

MONTH(*date-expression* **)**

Parameters

- **date-expression** A datetime value.

Returns

SMALLINT

Remarks

The value returned is a number from number from 1 to 12 corresponding to the datetime month.

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value 7.

```
SELECT MONTH( '1998-07-13' );
```

MONTHNAME function [Date and time]

Returns the name of the month from a date.

Syntax

MONTHNAME(*date-expression* **)**

Parameters

- **date-expression** The datetime value.

Returns

VARCHAR

Remarks

The MONTHNAME function returns a string, even if the result is numeric, such as 2 for the month of February.

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value September.

```
SELECT MONTHNAME( '1998-09-05' );
```

MONTHS function [Date and time]

The information you specify changes the behavior of this function:

- If you give a single date, this function returns the number of months since 0000-02.

  Note
  0000-02 is not meant to imply an actual date; it is the date used by the date algorithm.

- If you give two time stamps, this function returns the integer number of months between them. Instead, use the DATEDIFF function.

- If you give a date and an integer, this function adds the integer number of minutes to the specified timestamp. Instead, use the DATEADD function.

Syntax 1: integer

MONTHS([*datetime-expression*,] *datetime-expression* **)**

Syntax 2: timestamp

MONTHS(*datetime-expression*, *integer-expression* **)**

Parameters

- **datetime-expression** A date and time.
- **integer-expression** The number of months to be added to the *datetime-expression*. If *integer-expression* is negative, the appropriate number of months is subtracted from the datetime value. If you supply an *integer-expression*, the *datetime-expression* must be explicitly cast as a datetime data type.

  For information about casting data types, see "CAST function [Data type conversion]" on page 148.

Returns

INT

TIMESTAMP

Remarks

The value of MONTHS is calculated from the number of first days of the month between the two dates.

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statements return the value 2, signifying that the second date is two months after the first. It is recommended that you use the second example (DATEDIFF).

```
SELECT MONTHS( '1999-07-13 06:07:12',
   '1999-09-13 10:07:12' );

SELECT DATEDIFF( month,
   '1999-07-13 06:07:12',
   '1999-09-13 10:07:12' );
```

The following statement returns the value 23981.

```
SELECT MONTHS( '1998-07-13 06:07:12' );
```

The following statements return the timestamp 1999-10-12 21:05:07.000. It is recommended that you use the second example (DATEADD).

```
SELECT MONTHS( CAST( '1999-05-12 21:05:07'
AS DATETIME ), 5);

SELECT DATEADD( month, 5, '1999-05-12 21:05:07' );
```

NCHAR function [String]

Returns an NCHAR string containing one character whose Unicode code point is given in the parameter, or NULL if the value is not a valid code point value.

Syntax

NCHAR(*integer* **)**

Parameters

- **integer** The number to be converted to the corresponding Unicode code point.

Returns

NVARCHAR

See also

- "CONNECTION_EXTENDED_PROPERTY function [String]" on page 156
- "TO_NCHAR function [String]" on page 324
- "TO_CHAR function [String]" on page 323
- "UNICODE function [String]" on page 331
- "UNISTR function [String]" on page 331

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following example returns the ALEF Arabic letter, which is Unicode code point U+627:

```
SELECT NCHAR( 1575 );
```

NEWID function [Miscellaneous]

Generates a UUID (Universally Unique Identifier) value. A UUID is the same as a GUID (Globally Unique Identifier).

Syntax

NEWID( )

Parameters

There are no parameters associated with the NEWID function.

Returns

UNIQUEIDENTIFIER

Remarks

The NEWID function can be used in a DEFAULT clause for a column.

UUIDs can be used to uniquely identify rows in a table. A value produced on one computer does not match a value produced on another computer, so they can be used as keys in synchronization and replication environments.

UUIDs contain hyphens for compatibility with other RDBMSs.

The NEWID function is non-deterministic; successive calls may return different values. The query optimizer does not cache the results of the NEWID function.

For more information about non-deterministic functions, see "Function caching" [*SQL Anywhere Server - SQL Usage*].

See also

- "The NEWID default" [*SQL Anywhere Server - SQL Usage*]
- "STRTOUUID function [String]" on page 316
- "UUIDTOSTR function [String]" on page 335

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement creates a table named mytab with two columns. Column pk has a unique identifier data type, and assigns the NEWID function as the default value. Column c1 has an integer data type.

```
CREATE TABLE mytab(
   pk UNIQUEIDENTIFIER PRIMARY KEY DEFAULT NEWID(),
   c1 INT );
```

The following statement returns a unique identifier as a string:

```
SELECT NEWID();
```

For example, the value returned might be 96603324-6FF6-49DE-BF7D-F44C1C7E6856.

NEXT_CONNECTION function [System]

Returns an identifying number for the next connection.

Syntax

NEXT_CONNECTION([*connection-id*] [, *database-id*] **)**

Returns

INT

Parameters

- **connection-id** An integer, usually returned from a previous call to NEXT_CONNECTION. If *connection-id* is NULL, NEXT_CONNECTION returns the most recent connection ID.
- **database-id** An integer representing one of the databases on the current server. If you supply no *database-id*, the current database is used. If you supply NULL, then NEXT_CONNECTION returns the next connection regardless of database.

Remarks

NEXT_CONNECTION can be used to enumerate the connections to a database. Connection IDs are generally created in monotonically increasing order. This function returns the next connection ID in reverse order.

To get the connection ID value for the most recent connection, enter NULL as the *connection-id*. To get the subsequent connection, enter the previous return value. The function returns NULL when there are no more connections in the order.

NEXT_CONNECTION is useful if you want to disconnect all the connections created before a specific time. However, because NEXT_CONNECTION returns the connection IDS in reverse order, connections made after the function is started are not returned. If you want to ensure that all connections are disconnected, prevent new connections from being created before you run NEXT_CONNECTION.

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns an identifier for the first connection on the current database. The identifier is an integer value like 10.

```
SELECT NEXT_CONNECTION( NULL );
```

The following statement returns a value like 5.

```
SELECT NEXT_CONNECTION( 10 );
```

The following call returns the next connection ID in reverse order from the specified *connection-id* on the current database.

```
SELECT NEXT_CONNECTION( connection-id );
```

The following call returns the next connection ID in reverse order from the specified *connection-id* (regardless of database).

```
SELECT NEXT_CONNECTION( connection-id, NULL );
```

The following call returns the next connection ID in reverse order from the specified *connection-id* on the specified database.

```
SELECT NEXT_CONNECTION( connection-id, database-id );
```

The following call returns the first (earliest) connection (regardless of database).

```
SELECT NEXT_CONNECTION( NULL, NULL );
```

The following call returns the first (earliest) connection on the specified database.

```
SELECT NEXT_CONNECTION( NULL, database-id );
```

NEXT_DATABASE function [System]

Returns an identifying number for a database.

Syntax

NEXT_DATABASE({ **NULL** | *database-id* } **)**

Parameters

- **database-id** An integer that specifies the ID number of the database.

Returns

INT

Remarks

The NEXT_DATABASE function is used to enumerate the databases running on a database server. To get the first database pass NULL; to get each subsequent database, pass the previous return value. The function returns NULL when there are no more databases. The database ID numbers are not returned in a particular order, but you can tell the order in which connections were made to the server using the database ID. The first database that connects to the server is assigned the value 0, and for subsequent connections to the server, the database IDs are incremented by a value of 1.

Standards and compatibility

- **SQL/2003** Transact-SQL extension.

Example

The following statement returns the value 0, the first database value.

```
SELECT NEXT_DATABASE( NULL );
```

The following statement returns NULL, indicating that there are no more databases on the server.

```
SELECT NEXT_DATABASE( 0 );
```

NEXT_HTTP_HEADER function [HTTP]

Get the next HTTP header name.

Syntax

NEXT_HTTP_HEADER(*header-name* **)**

Parameters

header-name The name of the previous header. If header-name is NULL, this function returns the name of the first HTTP header.

Returns

LONG VARCHAR

Remarks

This function iterates over the HTTP headers included within a request and returns the next HTTP header name. Calling it with NULL causes it to return the name of the first header. Subsequent headers are retrieved by passing the function the name of the previous header. This function returns NULL when called with the name of the last header, or when not called from a web service.

Calling this function repeatedly returns all the header fields exactly once, but not necessarily in the order they appear in the HTTP request.

See also

- "-xs server option" [*SQL Anywhere Server - Database Administration*]
- "Working with HTTP headers" [*SQL Anywhere Server - Programming*]
- "HTTP_DECODE function [HTTP]" on page 220
- "HTTP_ENCODE function [HTTP]" on page 221
- "HTTP_HEADER function [HTTP]" on page 223
- "HTTP_VARIABLE function [HTTP]" on page 225
- "NEXT_HTTP_VARIABLE function [HTTP]" on page 256
- "SQL Anywhere web services" [*SQL Anywhere Server - Programming*]
- "Web services functions" on page 130
- "Web services system procedures" on page 830
- "Working with HTTP headers" [*SQL Anywhere Server - Programming*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

When used within a stored procedure that is called by an HTTP web service, the following example returns the name of the first HTTP header.

```
BEGIN
DECLARE header_name LONG VARCHAR;
SET header_name  = NULL;
SET header_name = NEXT_HTTP_HEADER( header_name )
END;
```

NEXT_HTTP_VARIABLE function [HTTP]

Get the next HTTP variable name.

Syntax

NEXT_HTTP_VARIABLE(*var-name* **)**

Parameters

- **var-name** The name of the previous variable. If *var-name* is NULL, this function returns the name of the first HTTP variable.

Returns

LONG VARCHAR

Remarks

This function iterates over the HTTP variables included within a request. Calling it with NULL causes it to return the name of the first variable. Subsequent variables are retrieved by passing the function the name of the previous variable. This function returns NULL when called with the name of the final variable, when not called from a web service.

Calling this function repeatedly returns all the variables exactly once, but not necessarily in the order they appear in the HTTP request. The variables url or url1, url2, ..., url10 are included if URL PATH is set to ON or ELEMENTS, respectively.

See also

- "-xs server option" [*SQL Anywhere Server - Database Administration*]
- "Working with variables" [*SQL Anywhere Server - Programming*]
- "HTTP_DECODE function [HTTP]" on page 220
- "HTTP_ENCODE function [HTTP]" on page 221
- "HTTP_HEADER function [HTTP]" on page 223
- "HTTP_VARIABLE function [HTTP]" on page 225
- "NEXT_HTTP_HEADER function [HTTP]" on page 255
- "SQL Anywhere web services" [*SQL Anywhere Server - Programming*]
- "Web services functions" on page 130
- "Web services system procedures" on page 830

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

When used within a stored procedure that is called by an HTTP web service, the following example returns the name of the first HTTP variable.

```
BEGIN
DECLARE variable_name LONG VARCHAR;
SET variable_name = NULL;
SET variable_name = NEXT_HTTP_VARIABLE( variable_name )
END;
```

NEXT_SOAP_HEADER function [SOAP]

Returns the next header key in a SOAP request header.

Syntax

NEXT_SOAP_HEADER(*header-key* **)**

Parameters

header-key The XML local name of the top level XML element for the given header entry.

Returns

LONG VARCHAR

Remarks

If you specify NULL for the *header-key*, the function returns the header key for the first header entry found in the SOAP header.

This function returns NULL if called with the last *header-key*.

See also

- "-xs server option" [*SQL Anywhere Server - Database Administration*]
- "Working with SOAP headers" [*SQL Anywhere Server - Programming*]
- "SQL Anywhere web services" [*SQL Anywhere Server - Programming*]
- "SOAP_HEADER function [SOAP]" on page 304
- "Web services functions" on page 130
- "Web services system procedures" on page 830

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

When used within a stored procedure that is called by an HTTP web service, the following statement returns the first header key found in the SOAP header.

```
SET header_key = NEXT_SOAP_HEADER( NULL );
```

NOW function [Date and time]

Returns the current year, month, day, hour, minute, second, and fraction of a second. The accuracy is limited by the accuracy of the system clock.

Syntax

NOW(*)

Returns

TIMESTAMP

Remarks

The information the NOW function returns is equivalent to the information returned by the GETDATE function and the CURRENT TIMESTAMP special value.

See also

- "GETDATE function [Date and time]" on page 210
- "CURRENT TIMESTAMP special value" on page 59

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the current date and time.

```
SELECT NOW( * );
```

NULLIF function [Miscellaneous]

Provides an abbreviated CASE expression by comparing expressions.

Syntax

NULLIF(*expression-1*, *expression-2* **)**

Parameters

- **expression-1** An expression to be compared.
- **expression-2** An expression to be compared.

Returns

Data type of the first argument.

Remarks

NULLIF compares the values of the two expressions.

If the first expression equals the second expression, NULLIF returns NULL.

If the first expression does not equal the second expression, or if the second expression is NULL, NULLIF returns the first expression.

The NULLIF function provides a short way to write some CASE expressions.

See also

- "CASE expressions" on page 18

Standards and compatibility

- **SQL/2003** Core feature.

Example

The following statement returns the value a:

```
SELECT NULLIF( 'a', 'b' );
```

The following statement returns NULL.

```
SELECT NULLIF( 'a', 'a' );
```

NUMBER function [Miscellaneous]

Generates numbers starting at 1 for each successive row in the results of the query. The NUMBER function is primarily intended for use in select lists.

Due to limitations imposed by the NUMBER function (described in the Remarks section below), use the "ROW_NUMBER function [Miscellaneous]" on page 296, instead. The ROW_NUMBER function provides the same functionality, but without the limitations of the NUMBER function.

Syntax

NUMBER(*)

Returns

INT

Remarks

You can use NUMBER(*) in a select list to provide a sequential numbering of the rows in the result set. NUMBER(*) returns the value of the ANSI row number of each result row. This means that the NUMBER function can return positive or negative values, depending on how the application scrolls through the result set. For insensitive cursors, the value of NUMBER(*) will always be positive because the entire result set is materialized at OPEN.

In addition, the row number may be subject to change for some cursor types. The value is fixed for insensitive cursors and scroll cursors. If there are concurrent updates, it may change for dynamic and sensitive cursors.

A syntax error is generated if you use the NUMBER function in: a DELETE statement, a WHERE clause, a HAVING clause, an ORDER BY clause, a subquery, a query involving aggregation, any constraint, a GROUP BY clause, a DISTINCT clause, a query expression (UNION, EXCEPT, INTERSECT), or a derived table.

NUMBER(*) can be used in a view (subject to the above restrictions), but the view column corresponding to the expression involving NUMBER(*) can be referenced at most once in the query or outer view, and the view cannot participate as a NULL-supplying table in a left outer join or full outer join.

In embedded SQL, care should be exercised when using a cursor that references a query containing a NUMBER(*) function. In particular, this function returns negative numbers when a database cursor is positioned using relative to the end of the cursor (an absolute position with a negative offset).

You can use NUMBER in the right-hand side of an assignment in the SET clause of an UPDATE statement. For example, `SET x = NUMBER(*)`.

The NUMBER function can also be used to generate primary keys when using the INSERT from SELECT statement (see "INSERT statement" on page 649), although using an AUTOINCREMENT clause is a preferred mechanism for generating sequential primary keys.

For information about the AUTOINCREMENT clause, see "CREATE TABLE statement" on page 523.

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns a sequentially-numbered list of departments.

```
SELECT NUMBER( * ), DepartmentName
FROM Departments
WHERE DepartmentID > 5
ORDER BY DepartmentName;
```

SQL functions (P-Z)

Each function is listed, and the function type (numeric, character, and so on) is indicated next to it.

For links to all functions of a given type, see "Function types" on page 124.

See also

- "SQL functions (A-D)" on page 135
- "SQL functions (E-O)" on page 193

PATINDEX function [String]

Returns an integer representing the starting position of the first occurrence of a pattern in a string.

Syntax

PATINDEX('%*pattern***%'**, *string-expression* **)**

Parameters

- **pattern** The pattern to be searched for. If the leading percent wildcard is omitted, the PATINDEX function returns one (1) if the pattern occurs at the beginning of the string, and zero if not.

  The pattern uses the same wildcards as the LIKE comparison. These are as follows:

Wildcard	Matches
_ (underscore)	Any one character
% (percent)	Any string of zero or more characters
[]	Any single character in the specified range or set
[^]	Any single character not in the specified range or set

- **string-expression** The string to be searched for the pattern.

Returns

INT

Remarks

The PATINDEX function returns the starting position of the first occurrence of the pattern. If the pattern is not found, it returns zero (0).

See also

- "LIKE search condition" on page 41
- "LOCATE function [String]" on page 240
- "String functions" on page 131

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value 2.

```
SELECT PATINDEX( '%hoco%', 'chocolate' );
```

The following statement returns the value 11.

```
SELECT PATINDEX( '%4_5_', '0a1A 2a3A 4a5A' );
```

The following statement returns 14 which is the first non-alphanumeric character in the string expression. Note that the pattern `'%[^a-z0-9]%'` can be used instead of `'%[^a-zA-Z0-9]%'` if the database is case insensitive.

```
SELECT PATINDEX( '%[^a-zA-Z0-9]%', 'SQLAnywhere11 has many new features' );
```

To get the first alphanumeric word in a string, you can use something like the following:

```
SELECT LEFT( @string, PATINDEX( '%[^a-zA-Z0-9]%', @string ) );
```

PERCENT_RANK function [Ranking]

For any row X, defined by the function's arguments and ORDER BY specification, the PERCENT_RANK function determines the rank of row X - 1, divided by the number of rows in the group.

Syntax

PERCENT_RANK( ) OVER (*window-spec* **)**

window-spec : see the Remarks section below

Returns

The PERCENT_RANK function returns a DOUBLE value between 0 and 1.

Remarks

Elements of *window-spec* can be specified either in the function syntax (inline), or in conjunction with a WINDOW clause in the SELECT statement. When used as a window function, you must specify an ORDER BY clause, you may specify a PARTITION BY clause, however, you can not specify a ROWS or RANGE clause. See the *window-spec* definition provided in "WINDOW clause" on page 793.

For more information about using window functions in SELECT statements, including working examples, see "Window functions" [*SQL Anywhere Server - SQL Usage*].

See also

- "CUME_DIST function [Ranking]" on page 170
- "DENSE_RANK function [Ranking]" on page 189
- "RANK function [Ranking]" on page 272

Standards and compatibility

- **SQL/2003** SQL/OLAP feature T612.

Example

The following example returns a result set that shows the ranking of New York employees' salaries by gender. The results are ranked in descending order and are partitioned by gender.

```
SELECT DepartmentID, Surname, Salary, Sex,
PERCENT_RANK() OVER (PARTITION BY Sex
ORDER BY Salary DESC) "Rank"
FROM Employees
WHERE State IN ('NY');
```

DepartmentID	Surname	Salary	Sex	Rank
200	Martel	55700.000	M	0
100	Guevara	42998.000	M	0.333333333
100	Soo	39075.000	M	0.666666667
400	Ahmed	34992.000	M	1
300	Davidson	57090.000	F	0
400	Blaikie	54900.000	F	0.333333333
100	Whitney	45700.000	F	0.666666667
400	Wetherby	35745.000	F	1

PI function [Numeric]

Returns the numeric value PI.

Syntax

PI(*)

Returns

DOUBLE

Standards and compatibility

- **SQL/2003** Vendor extension.

Remarks

This function returns a DOUBLE value.

Example

The following statement returns the value 3.141592653...

```
SELECT PI( * );
```

PLAN function [Miscellaneous]

Returns the long plan optimization strategy of a SQL statement, as a string.

Syntax

PLAN(*string-expression*, [*cursor-type*], [*update-status*] **)**

Parameters

- **string-expression** The SQL statement, which is commonly a SELECT statement but which may also be an UPDATE or DELETE statement.
- **cursor-type** A string. *cursor-type* can be asensitive (default), insensitive, sensitive, or keyset-driven.
- **update-status** A string parameter accepting one of the following values indicating how the optimizer should treat the given cursor:

Value	Description
READ-ONLY	The cursor is read-only.
READ-WRITE (default)	The cursor can be read or written to.
FOR UPDATE	The cursor can be read or written to. This is exactly the same as READ-WRITE.

Returns

LONG VARCHAR

See also

- "EXPLANATION function [Miscellaneous]" on page 204
- "GRAPHICAL_PLAN function [Miscellaneous]" on page 211

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement passes a SELECT statement as a string parameter and returns the plan for executing the query.

```
SELECT PLAN(
   'SELECT * FROM Departments WHERE DepartmentID > 100' );
```

This information can help with decisions about indexes to add or how to structure your database for better performance.

The following statement returns a string containing the text plan for an INSENSITIVE cursor over the query `SELECT * FROM Departments WHERE DepartmentID > 100;`.

```
SELECT PLAN(
   'SELECT * FROM Departments WHERE DepartmentID > 100',
   'insensitive',
   'read-only' );
```

POWER function [Numeric]

Calculates one number raised to the power of another.

Syntax

POWER(*numeric-expression-1*, *numeric-expression-2* **)**

Parameters

- **numeric-expression-1** The base.
- **numeric-expression-2** The exponent.

Returns

DOUBLE

Remarks

This function converts its arguments to DOUBLE, and performs the computation in double-precision floating point. If any argument is NULL, the result is a NULL value.

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value 64.

```
SELECT POWER( 2, 6 );
```

PROPERTY function [System]

Returns the value of the specified database server property as a string.

Syntax

PROPERTY({ *property-id* | *property-name* } **)**

Parameters

- **property-id** An integer that is the property-number of the database server property. This number can be determined from the PROPERTY_NUMBER function. The *property-id* is commonly used when looping through a set of properties.
- **property-name** A string giving the name of the database property.

Returns

VARCHAR

Remarks

Each property has both a number and a name, but the number is subject to change between releases, and should not be used as a reliable identifier for a given property.

See also

- "Database server properties" [*SQL Anywhere Server - Database Administration*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the name of the current database server:

```
SELECT PROPERTY( 'Name' );
```

PROPERTY_DESCRIPTION function [System]

Returns a description of a property.

Syntax

PROPERTY_DESCRIPTION({ *property-id* | *property-name* } **)**

Parameters

- **property-id** An integer that is the property-number of the database property. This number can be determined from the PROPERTY_NUMBER function. The *property-id* is commonly used when looping through a set of properties.
- **property-name** A string giving the name of the database property.

Returns

VARCHAR

Remarks

Each property has both a number and a name, but the number is subject to change between releases, and should not be used as a reliable identifier for a given property.

See also

- “Connection, database, and database server properties” [*SQL Anywhere Server - Database Administration*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the description Number of index insertions.

```
SELECT PROPERTY_DESCRIPTION( 'IndAdd' );
```

PROPERTY_NAME function [System]

Returns the name of the property with the supplied property ID for the specified connection level.

Syntax

PROPERTY_NAME(*property-id* [**,** *property-scope*] **)**

property-scope:
NULL
| **'server'**
| **'database'**
| **'db'**
| **'connection'**
| **'conn'**

Parameters

- **property-id** The property ID of the database property.
- **property-scope** The scope of the property, or NULL.

Returns

VARCHAR

See also

- “Connection properties” [*SQL Anywhere Server - Database Administration*]
- “Database server properties” [*SQL Anywhere Server - Database Administration*]
- “Database properties” [*SQL Anywhere Server - Database Administration*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the server-level property associated with property ID 102.

```
SELECT PROPERTY_NAME( 102, 'server' );
```

PROPERTY_NUMBER function [System]

Returns the property number of the property with the supplied property-name.

Syntax

PROPERTY_NUMBER(*property-name* **)**

Parameters

- **property-name**   A property name.

Returns

INT

Remarks

Each property has both a number and a name, but the number is subject to change between releases, and should not be used as a reliable identifier for a given property. In cases where either property number or property name can be used, it is preferable to use the property name. Always use the PROPERTY_NUMBER function to ensure that the property number is current for the server being used.

See also

- "Connection, database, and database server properties" [*SQL Anywhere Server - Database Administration*]

Standards and compatibility

- **SQL/2003**   Vendor extension.

Example

The following statement returns an integer value. The actual value changes from release to release.

```
SELECT PROPERTY_NUMBER( 'PAGESIZE' );
```

QUARTER function [Date and time]

Returns a number indicating the quarter of the year from the supplied date expression.

Syntax

QUARTER(*date-expression* **)**

Parameters

- **date-expression**   The date.

Returns

INT

Remarks

The quarters are as follows:

Quarter	Period (inclusive)
1	January 1 to March 31
2	April 1 to June 30
3	July 1 to September 30
4	October 1 to December 31

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value 2.

```
SELECT QUARTER( '1987/05/02' );
```

RADIANS function [Numeric]

Converts a number from degrees to radians.

Syntax

RADIANS(*numeric-expression* **)**

Parameters

- **numeric-expression** A number, in degrees. This angle is converted to radians.

Returns

DOUBLE

Remarks

This function converts its argument to DOUBLE, and performs the computation in double-precision floating point.

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns a value of approximately 0.5236.

```
SELECT RADIANS( 30 );
```

RAND function [Numeric]

Returns a random number in the interval 0 to 1, with an optional seed.

Syntax

RAND([*integer-expression*] **)**

Parameters

- **integer-expression** An optional seed used to create a random number. This argument allows you to create repeatable random number sequences.

Returns

DOUBLE

Remarks

The RAND function is a multiplicative linear congruential random number generator. See Park and Miller (1988), CACM 31(10), pp. 1192-1201 and Press et al. (1992), Numerical Recipes in C (2nd edition, Chapter 7, pp. 279). The result of calling the RAND function is a pseudo-random number *n* where $0 < n < 1$ (neither 0.0 nor 1.0 can be the result).

When a connection is made to the server, the random number generator seeds an initial value. Each connection is uniquely seeded so that it sees a different random sequence from other connections. You can also specify a seed value (*integer-expression*) as an argument. Normally, you should only do this once before requesting a sequence of random numbers through successive calls to the RAND function. If you initialize the seed value more than once, the sequence is restarted. If you specify the same seed value, the same sequence is generated. Seed values that are close in value generate similar initial sequences, with divergence further out in the sequence.

Never combine the sequence generated from one seed value with the sequence generated from a second seed value, in an attempt to obtain statistically random results. In other words, do not reset the seed value at any time during the generation of a sequence of random values.

The RAND function is treated as a non-deterministic function. The query optimizer does not cache the results of the RAND function.

For more information about non-deterministic functions, see “Function caching” [*SQL Anywhere Server - SQL Usage*].

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statements produce eleven random results. Each subsequent call to the RAND function where a seed is not specified continues to produce different results:

```
SELECT RAND( 1 );
SELECT RAND( ), RAND( ), RAND( ), RAND( ), RAND( );
SELECT RAND( ), RAND( ), RAND( ), RAND( ), RAND( );
```

The following example produces two sets of results with identical sequences, since the seed value is specified twice:

```
SELECT RAND( 1 ), RAND( ), RAND( ), RAND( ), RAND( );
SELECT RAND( 1 ), RAND( ), RAND( ), RAND( ), RAND( );
```

The following example produces five results that are near each other in value, and are not random in terms of distribution. For this reason, calling the RAND function more than once with similar seed values is not recommended:

```
SELECT RAND( 1 ), RAND( 2 ), RAND( 3 ), RAND( 4 ), RAND( 5 );
```

The following example produces five identical results, and should be avoided:

```
SELECT RAND( 1 ), RAND( 1 ), RAND( 1 ), RAND( 1 ), RAND( 1 );
```

RANK function [Ranking]

Calculates the value of a rank in a group of values. In the case of ties, the RANK function leaves a gap in the ranking sequence.

Syntax

RANK( ) OVER (*window-spec* **)**

window-spec : see the Remarks section below

Returns

INTEGER

Remarks

Elements of *window-spec* can be specified either in the function syntax (inline), or in conjunction with a WINDOW clause in the SELECT statement. When used as a window function, you must specify an ORDER BY clause, you may specify a PARTITION BY clause, however, you can not specify a ROWS or RANGE clause. See the *window-spec* definition provided in “WINDOW clause” on page 793.

For more information about using window functions in SELECT statements, including working examples, see “Window functions” [*SQL Anywhere Server - SQL Usage*].

See also

- “CUME_DIST function [Ranking]” on page 170
- “DENSE_RANK function [Ranking]” on page 189
- “ROW_NUMBER function [Miscellaneous]” on page 296
- “PERCENT_RANK function [Ranking]” on page 263

Standards and compatibility

- **SQL/2003** SQL/OLAP feature T612.

Example

The following example provides a rank in descending order of employees' salaries in Utah and New York. Notice that the 7th and 8th employees have an identical salary and therefore share the 7th place ranking. The employee that follows receives the 9th place ranking, which leaves a gap in the ranking sequence (no 8th place ranking).

```
SELECT Surname, Salary, State,
RANK() OVER (ORDER BY Salary DESC) "Rank"
FROM Employees WHERE State IN ('NY','UT');
```

Surname	Salary	State	Rank
Shishov	72995.000	UT	1
Wang	68400.000	UT	2
Cobb	62000.000	UT	3
Morris	61300.000	UT	4
Davidson	57090.000	NY	5
Martel	55700.000	NY	6
Blaikie	54900.000	NY	7
Diaz	54900.000	NY	7
Driscoll	48023.690	UT	9
Hildebrand	45829.000	UT	10
Whitney	45700.000	NY	11
...	...	...	...
Lynch	24903.000	UT	19

READ_CLIENT_FILE function [String]

Reads data from the specified file on the client computer.

Syntax

READ_CLIENT_FILE(*client-filename-expression* **)**

Parameters

- **client-filename-expression** CHAR value indicating the name of the file on the client computer. The path is resolved on the client computer relative to the current working directory of the client application.

Returns

LONG BINARY

Remarks

The value returned by the READ_CLIENT_FILE function represents the contents of the specified client file. You can use the function in syntax wherever a BINARY expression is allowed.

Since the data returns as a binary string, if the data is in another character set, or is compressed, or is encrypted, you may also need to perform character set conversion, decompression, or decryption on it.

During evaluation of READ_CLIENT_FILE, the database server initiates the transfer of the specified file from the client. The client, upon receiving the transfer request, obtains a shared lock on the client file, and holds the lock until the database server requests the client to terminate the request.

Reading of the file is performed by the client software library, and the transfer of data is done using the command sequence communication protocol.

Permissions

When reading from a file on a client computer:

- READCLIENTFILE authority is required. See "READCLIENTFILE authority" [*SQL Anywhere Server - Database Administration*].
- Read permissions are required on the directory being read from.
- The allow_read_client_file database option must be enabled. See "allow_read_client_file option [database]" [*SQL Anywhere Server - Database Administration*].
- The read_client_file secured feature must be enabled. See "-sf server option" [*SQL Anywhere Server - Database Administration*].

Standards and compatibility

- **SQL/2003** Vendor extension.

See also

- "Accessing data on client computers" [*SQL Anywhere Server - SQL Usage*]
- "READCLIENTFILE authority" [*SQL Anywhere Server - Database Administration*]
- "DECOMPRESS function [String]" on page 186
- "DECRYPT function [String]" on page 187
- "CSCONVERT function [String]" on page 168

REGEXP_SUBSTR function [String]

Extracts substrings from strings using regular expressions.

Syntax

REGEXP_SUBSTR(*expression*,
regular-expression
[, *start-offset* [, *occurrence-number* [, *escape-expression*]]] **)**

Parameters

- **expression** The string to be searched.
- **regular-expression** The pattern you are trying to match. For more information about regular expression syntax, see "Regular expressions overview" on page 20.
- **start-offset** The offset into *expression* at which to start searching. *start-offset* is expressed as a positive integer, and reflects the number of characters to count when starting from the left side of the string. The default is 1 (the start of the string).
- **occurrence-number** In the case of multiple matches within *expression*, specify an integer indicating the occurrence to locate. For example, 3 finds the third occurrence. The default is 1.
- **escape-expression** The escape character to use for *regular-expression*. The default is the backslash character (\).

Returns

LONG VARCHAR

Remarks

REGEXP_SUBSTR returns NULL if *regular-expression* is not found.

Similar to the REGEXP search condition, the REGEXP_SUBSTR function uses code points for matching and range evaluation. This means that database case sensitivity does not impact results. For more information on how REGEXP_SUBSTR performs matching and set evaluation, see "LIKE, REGEXP, and SIMILAR TO: differences in character comparisons" on page 39.

When matching against a character class that contains only a sub-character class, include the outer square brackets and the square brackets for the sub-character class (for example, `REGEXP_SUBSTR(`*`expression`*`, '[[:digit:]]')`). For more on sub-character class matching, see "Regular expressions: Special sub-character classes" on page 24.

See also

- "Regular expressions syntax" on page 21
- "REGEXP search condition" on page 45

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following example breaks values in the Employees.Street column into street number and street name:

```
SELECT REGEXP_SUBSTR( Street, '^\S+' ) as street_num,
 REGEXP_SUBSTR( Street, '(?<=^\S+\s+).*$' ) AS street_name
    FROM Employees;
```

street_num	street_name
9	East Washington Street
7	Pleasant Street
539	Pond Street
1244	Great Plain Avenue
...	...

To determine whether the IP address of the current connection is in a range of IP addresses (in this case, 10.25.101.xxx or 10.25.102.xxx), you can execute the following statement:

```
IF REGEXP_SUBSTR( CONNECTION_PROPERTY( 'NodeAddress' ), '\\d+\\.\\d+\\.\\d
+' )
   IN ( '10.25.101' , '10.25.102' ) THEN
      MESSAGE 'In range' TO CLIENT;
ELSE
      MESSAGE 'Out of range' TO CLIENT;
END IF;
```

REGR_AVGX function [Aggregate]

Computes the average of the independent variable of the regression line.

Syntax 1

REGR_AVGX(*dependent-expression* , *independent-expression* **)**

Syntax 2

REGR_AVGX(*dependent-expression* , *independent-expression* **)**
OVER (*window-spec* **)**

window-spec : see Syntax 2 instructions in the Remarks section below

Parameters

- **dependent-expression** The variable that is affected by the independent variable.
- **independent-expression** The variable that influences the outcome.

Returns

DOUBLE

Remarks

This function converts its arguments to DOUBLE, and performs the computation in double-precision floating point. If the function is applied to an empty set, then it returns NULL.

The function is applied to the set of (*dependent-expression* and *independent-expression*) pairs after eliminating all pairs for which either *dependent-expression* or *independent-expression* is NULL. The function is computed simultaneously during a single pass through the data. After eliminating NULL values, the following computation is then made, where *x* represents the *independent-expression*:

```
AVG( x )
```

For more information about the statistical computation performed, see "Mathematical formulas for the aggregate functions" [*SQL Anywhere Server - SQL Usage*].

Syntax 2 represents usage as a window function in a SELECT statement. As such, elements of *window-spec* can be specified either in the function syntax (inline), or in conjunction with a WINDOW clause in the SELECT statement. See the *window-spec* definition provided in "WINDOW clause" on page 793.

For more information about using window functions in SELECT statements, including working examples, see "Window functions" [*SQL Anywhere Server - SQL Usage*].

See also

- "AVG function [Aggregate]" on page 140
- "REGR_COUNT function [Aggregate]" on page 279
- "REGR_INTERCEPT function [Aggregate]" on page 280
- "REGR_COUNT function [Aggregate]" on page 279
- "REGR_SLOPE function [Aggregate]" on page 282
- "REGR_SXX function [Aggregate]" on page 284
- "REGR_SXY function [Aggregate]" on page 285
- "REGR_SYY function [Aggregate]" on page 286
- "REGR_AVGY function [Aggregate]" on page 277

Standards and compatibility

- **SQL/2003** SQL foundation feature (T621) outside core SQL.

Example

The following example calculates the average of the dependent variable, employee age.

```
SELECT REGR_AVGX( Salary, ( 2008 - YEAR( BirthDate ) ) )
FROM Employees;
```

REGR_AVGY function [Aggregate]

Computes the average of the dependent variable of the regression line.

Syntax 1

REGR_AVGY(*dependent-expression* , *independent-expression* **)**

Syntax 2

REGR_AVGY(*dependent-expression* , *independent-expression* **)**
OVER (*window-spec* **)**

window-spec : see Syntax 2 instructions in the Remarks section below

Parameters

- **dependent-expression** The variable that is affected by the independent variable.
- **independent-expression** The variable that influences the outcome.

Returns

DOUBLE

Remarks

This function converts its arguments to DOUBLE, and performs the computation in double-precision floating point. If the function is applied to an empty set, then it returns NULL.

The function is applied to the set of (*dependent-expression* and *independent-expression*) pairs after eliminating all pairs for which either *dependent-expression* or *independent-expression* is NULL. The function is computed simultaneously during a single pass through the data. After eliminating NULL values, the following computation is then made, where *y* represents the *dependent-expression*:

```
AVG( y )
```

For more information about the statistical computation performed, see "Mathematical formulas for the aggregate functions" [*SQL Anywhere Server - SQL Usage*].

Syntax 2 represents usage as a window function in a SELECT statement. As such, elements of *window-spec* can be specified either in the function syntax (inline), or in conjunction with a WINDOW clause in the SELECT statement. See the *window-spec* definition provided in "WINDOW clause" on page 793.

For more information about using window functions in SELECT statements, including working examples, see "Window functions" [*SQL Anywhere Server - SQL Usage*].

See also

- "REGR_COUNT function [Aggregate]" on page 279
- "REGR_INTERCEPT function [Aggregate]" on page 280
- "REGR_COUNT function [Aggregate]" on page 279
- "REGR_SLOPE function [Aggregate]" on page 282
- "REGR_SXX function [Aggregate]" on page 284
- "REGR_SXY function [Aggregate]" on page 285
- "REGR_SYY function [Aggregate]" on page 286
- "REGR_AVGX function [Aggregate]" on page 276
- "AVG function [Aggregate]" on page 140

Standards and compatibility

- **SQL/2003** SQL foundation feature (T621) outside core SQL.

Example

The following example calculates the average of the independent variable, employee salary.

```
SELECT REGR_AVGY( Salary, ( YEAR( NOW( )) - YEAR( BirthDate ) ) )
FROM Employees;
```

REGR_COUNT function [Aggregate]

Returns an integer that represents the number of non-NULL number pairs used to fit the regression line.

Syntax 1

REGR_COUNT(*dependent-expression* , *independent-expression* **)**

Syntax 2

REGR_COUNT(*dependent-expression* , *independent-expression* **)**
OVER (*window-spec* **)**

window-spec : see Syntax 2 instructions in the Remarks section below

Parameters

- **dependent-expression** The variable that is affected by the independent variable.
- **independent-expression** The variable that influences the outcome.

Returns

INTEGER

Remarks

Syntax 2 represents usage as a window function in a SELECT statement. As such, elements of *window-spec* can be specified either in the function syntax (inline), or in conjunction with a WINDOW clause in the SELECT statement. See the *window-spec* definition provided in "WINDOW clause" on page 793.

For more information about using window functions in SELECT statements, including working examples, see "Window functions" [*SQL Anywhere Server - SQL Usage*].

For more information about the statistical computation performed, see "Mathematical formulas for the aggregate functions" [*SQL Anywhere Server - SQL Usage*].

See also

- "REGR_INTERCEPT function [Aggregate]" on page 280
- "REGR_COUNT function [Aggregate]" on page 279
- "REGR_SLOPE function [Aggregate]" on page 282
- "REGR_SXX function [Aggregate]" on page 284
- "REGR_SXY function [Aggregate]" on page 285
- "REGR_SYY function [Aggregate]" on page 286
- "REGR_AVGY function [Aggregate]" on page 277
- "REGR_AVGX function [Aggregate]" on page 276
- "COUNT function [Aggregate]" on page 164
- "AVG function [Aggregate]" on page 140
- "SUM function [Aggregate]" on page 320

Standards and compatibility

- **SQL/2003** SQL foundation feature (T621) outside core SQL.

Example

The following example returns the number of non-NULL pairs that were used to fit the regression line.

```
SELECT REGR_COUNT( Salary, ( YEAR( NOW() ) - YEAR( BirthDate ) ) )
FROM Employees;
```

REGR_INTERCEPT function [Aggregate]

Computes the y-intercept of the linear regression line that best fits the dependent and independent variables.

Syntax 1

REGR_INTERCEPT(*dependent-expression* , *independent-expression* **)**

Syntax 2

REGR_INTERCEPT(*dependent-expression* , *independent-expression* **)**
OVER (*window-spec* **)**

window-spec : see Syntax 2 instructions in the Remarks section below

Parameters

- **dependent-expression** The variable that is affected by the independent variable.
- **independent-expression** The variable that influences the outcome.

Returns

DOUBLE

Remarks

This function converts its arguments to DOUBLE, and performs the computation in double-precision floating point. If the function is applied to an empty set, then it returns NULL.

The function is applied to the set of (*dependent-expression* and *independent-expression*) pairs after eliminating all pairs for which either *dependent-expression* or *independent-expression* is NULL. The function is computed simultaneously during a single pass through the data. After eliminating NULL values, the following computation is then made, where *y* represents the *dependent-expression* and *x* represents the *independent-expression*:

```
AVG( y ) - REGR_SLOPE( y, x ) * AVG( x )
```

For more information about the statistical computation performed, see "Mathematical formulas for the aggregate functions" [*SQL Anywhere Server - SQL Usage*].

Syntax 2 represents usage as a window function in a SELECT statement. As such, elements of *window-spec* can be specified either in the function syntax (inline), or in conjunction with a WINDOW clause in the SELECT statement. See the *window-spec* definition provided in "WINDOW clause" on page 793.

For more information about using window functions in SELECT statements, including working examples, see "Window functions" [*SQL Anywhere Server - SQL Usage*].

See also

- "REGR_COUNT function [Aggregate]" on page 279
- "REGR_COUNT function [Aggregate]" on page 279
- "REGR_SLOPE function [Aggregate]" on page 282
- "REGR_SXX function [Aggregate]" on page 284
- "REGR_SXY function [Aggregate]" on page 285
- "REGR_SYY function [Aggregate]" on page 286
- "REGR_AVGY function [Aggregate]" on page 277
- "REGR_AVGX function [Aggregate]" on page 276
- "REGR_SLOPE function [Aggregate]" on page 282
- "AVG function [Aggregate]" on page 140

Standards and compatibility

- **SQL/2003** SQL foundation feature (T621) outside core SQL.

Example

The following example returns the y-intercept of the linear regression line.

```
SELECT REGR_INTERCEPT( Salary, ( YEAR( NOW( )) - YEAR( BirthDate ) ) )
FROM Employees;
```

REGR_R2 function [Aggregate]

Computes the coefficient of determination (also referred to as *R-squared* or the *goodness of fit* statistic) for the regression line.

Syntax 1

REGR_R2(*dependent-expression* , *independent-expression* **)**

Syntax 2

REGR_R2(*dependent-expression* , *independent-expression* **)**
OVER (*window-spec* **)**

window-spec : see Syntax 2 instructions in the Remarks section below

Parameters

- **dependent-expression** The variable that is affected by the independent variable.
- **independent-expression** The variable that influences the outcome.

Returns

DOUBLE

Remarks

This function converts its arguments to DOUBLE, and performs the computation in double-precision floating point. If the function is applied to an empty set, then it returns NULL.

The function is applied to the set of (*dependent-expression* and *independent-expression*) pairs after eliminating all pairs for which either *dependent-expression* or *independent-expression* is NULL.

For more information about the statistical computation performed, see "Mathematical formulas for the aggregate functions" [*SQL Anywhere Server - SQL Usage*].

Syntax 2 represents usage as a window function in a SELECT statement. As such, elements of *window-spec* can be specified either in the function syntax (inline), or in conjunction with a WINDOW clause in the SELECT statement. See the *window-spec* definition provided in "WINDOW clause" on page 793.

For more information about using window functions in SELECT statements, including working examples, see "Window functions" [*SQL Anywhere Server - SQL Usage*].

See also

- "REGR_COUNT function [Aggregate]" on page 279
- "REGR_INTERCEPT function [Aggregate]" on page 280
- "REGR_SLOPE function [Aggregate]" on page 282
- "REGR_SXX function [Aggregate]" on page 284
- "REGR_SXY function [Aggregate]" on page 285
- "REGR_SYY function [Aggregate]" on page 286
- "REGR_AVGX function [Aggregate]" on page 276
- "REGR_AVGY function [Aggregate]" on page 277

Standards and compatibility

- **SQL/2003** SQL foundation feature (T621) outside core SQL.

Example

The following example returns the coefficient of determination for the regression line.

```
SELECT REGR_R2( Salary, ( YEAR( NOW( )) - YEAR( BirthDate ) ) )
FROM Employees;
```

REGR_SLOPE function [Aggregate]

Computes the slope of the linear regression line fitted to non-NULL pairs.

Syntax 1

REGR_SLOPE(*dependent-expression* , *independent-expression* **)**

Syntax 2

REGR_SLOPE(*dependent-expression* , *independent-expression* **)**
OVER (*window-spec* **)**

window-spec : see Syntax 2 instructions in the Remarks section below

Parameters

- **dependent-expression** The variable that is affected by the independent variable.

- **independent-expression** The variable that influences the outcome.

Returns

DOUBLE

Remarks

This function converts its arguments to DOUBLE, and performs the computation in double-precision floating point. If the function is applied to an empty set, then it returns NULL.

The function is applied to the set of (*dependent-expression* and *independent-expression*) pairs after eliminating all pairs for which either *dependent-expression* or *independent-expression* is NULL. The function is computed simultaneously during a single pass through the data. After eliminating NULL values, the following computation is then made, where *y* represents the *dependent-expression* and *x* represents the *independent-expression*:

```
COVAR_POP( y, x ) / VAR_POP( x )
```

For more information about the statistical computation performed, see “Mathematical formulas for the aggregate functions” [*SQL Anywhere Server - SQL Usage*].

Syntax 2 represents usage as a window function in a SELECT statement. As such, elements of *window-spec* can be specified either in the function syntax (inline), or in conjunction with a WINDOW clause in the SELECT statement. See the *window-spec* definition provided in “WINDOW clause” on page 793.

For more information about using window functions in SELECT statements, including working examples, see “Window functions” [*SQL Anywhere Server - SQL Usage*].

See also

- “REGR_COUNT function [Aggregate]” on page 279
- “REGR_INTERCEPT function [Aggregate]” on page 280
- “REGR_COUNT function [Aggregate]” on page 279
- “REGR_SXX function [Aggregate]” on page 284
- “REGR_SXY function [Aggregate]” on page 285
- “REGR_SYY function [Aggregate]” on page 286
- “REGR_AVGX function [Aggregate]” on page 276
- “REGR_AVGY function [Aggregate]” on page 277
- “COVAR_POP function [Aggregate]” on page 166
- “VAR_POP function [Aggregate]” on page 336

Standards and compatibility

- **SQL/2003** SQL foundation feature (T621) outside core SQL.

Example

The following example returns the value 935.3429749445614.

```
SELECT REGR_SLOPE( Salary, ( YEAR( NOW() ) - YEAR( BirthDate ) ) )
FROM Employees;
```

REGR_SXX function [Aggregate]

Returns the sum of squares of the independent expressions used in a linear regression model. The REGR_SXX function can be used to evaluate the statistical validity of a regression model.

Syntax 1

REGR_SXX(*dependent-expression* , *independent-expression* **)**

Syntax 2

REGR_SXX(*dependent-expression* , *independent-expression* **)**
OVER (*window-spec* **)**

window-spec : see Syntax 2 instructions in the Remarks section below

Parameters

- **dependent-expression** The variable that is affected by the independent variable.
- **independent-expression** The variable that influences the outcome.

Returns

DOUBLE

Remarks

This function converts its arguments to DOUBLE, and performs the computation in double-precision floating point. If the function is applied to an empty set, then it returns NULL.

The function is applied to the set of (*dependent-expression* and *independent-expression*) pairs after eliminating all pairs for which either *dependent-expression* or *independent-expression* is NULL. The function is computed simultaneously during a single pass through the data. After eliminating NULL values, the following computation is then made, where *y* represents the *dependent-expression* and *x* represents the *independent-expression*:

```
REGR_COUNT( y, x ) * VAR_POP( x )
```

For more information about the statistical computation performed, see "Mathematical formulas for the aggregate functions" [*SQL Anywhere Server - SQL Usage*].

Syntax 2 represents usage as a window function in a SELECT statement. As such, elements of *window-spec* can be specified either in the function syntax (inline), or in conjunction with a WINDOW clause in the SELECT statement. See the *window-spec* definition provided in "WINDOW clause" on page 793.

For more information about using window functions in SELECT statements, including working examples, see "Window functions" [*SQL Anywhere Server - SQL Usage*].

See also

- "REGR_COUNT function [Aggregate]" on page 279
- "REGR_INTERCEPT function [Aggregate]" on page 280
- "REGR_COUNT function [Aggregate]" on page 279
- "REGR_AVGX function [Aggregate]" on page 276
- "REGR_AVGY function [Aggregate]" on page 277
- "REGR_SXY function [Aggregate]" on page 285
- "REGR_SYY function [Aggregate]" on page 286
- "VAR_POP function [Aggregate]" on page 336

Standards and compatibility

- **SQL/2003** SQL foundation feature (T621) outside core SQL.

Example

The following example returns the value 5916.4800000000105.

```
SELECT REGR_SXX( Salary, ( YEAR( NOW() ) - YEAR( BirthDate ) ) )
FROM Employees;
```

REGR_SXY function [Aggregate]

Returns the sum of products of the dependent and independent variables. The REGR_SXY function can be used to evaluate the statistical validity of a regression model.

Syntax 1

REGR_SXY(*dependent-expression* , *independent-expression* **)**

Syntax 2

REGR_SXY(*dependent-expression* , *independent-expression* **)**
OVER (*window-spec* **)**

window-spec : see Syntax 2 instructions in the Remarks section below

Parameters

- **dependent-expression** The variable that is affected by the independent variable.
- **independent-expression** The variable that influences the outcome.

Returns

DOUBLE

Remarks

This function converts its arguments to DOUBLE, and performs the computation in double-precision floating point, and returns a DOUBLE as the result. If the function is applied to an empty set, then it returns NULL.

The function is applied to the set of (*dependent-expression* and *independent-expression*) pairs after eliminating all pairs for which either *dependent-expression* or *independent-expression* is NULL. The function is computed simultaneously during a single pass through the data. After eliminating NULL values, the following computation is then made, where *y* represents the *dependent-expression* and *x* represents the *independent-expression*:

```
REGR_COUNT( y, x ) * COVAR_POP( y, x )
```

For more information about the statistical computation performed, see “Mathematical formulas for the aggregate functions” [*SQL Anywhere Server - SQL Usage*].

Syntax 2 represents usage as a window function in a SELECT statement. As such, elements of *window-spec* can be specified either in the function syntax (inline), or in conjunction with a WINDOW clause in the SELECT statement. See the *window-spec* definition provided in “WINDOW clause” on page 793.

For more information about using window functions in SELECT statements, including working examples, see “Window functions” [*SQL Anywhere Server - SQL Usage*].

See also

- “REGR_COUNT function [Aggregate]” on page 279
- “REGR_INTERCEPT function [Aggregate]” on page 280
- “REGR_COUNT function [Aggregate]” on page 279
- “REGR_SLOPE function [Aggregate]” on page 282
- “REGR_AVGX function [Aggregate]” on page 276
- “REGR_AVGY function [Aggregate]” on page 277
- “REGR_SXX function [Aggregate]” on page 284
- “REGR_SYY function [Aggregate]” on page 286

Standards and compatibility

- **SQL/2003** SQL foundation feature (T621) outside core SQL.

Example

The following example returns the sum of products of the dependent and independent variables.

```
SELECT REGR_SXY( Salary, ( YEAR( NOW( )) - YEAR( BirthDate ) ) )
FROM Employees;
```

REGR_SYY function [Aggregate]

Returns values that can evaluate the statistical validity of a regression model.

Syntax 1

REGR_SYY(*dependent-expression* , *independent-expression* **)**

Syntax 2

REGR_SYY(*dependent-expression* , *independent-expression* **)**
OVER (*window-spec* **)**

window-spec : see Syntax 2 instructions in the Remarks section below

Parameters

- **dependent-expression** The variable that is affected by the independent variable.
- **independent-expression** The variable that influences the outcome.

Returns

DOUBLE

Remarks

This function converts its arguments to DOUBLE, and performs the computation in double-precision floating point. If the function is applied to an empty set, then it returns NULL.

The function is applied to the set of (*dependent-expression* and *independent-expression*) pairs after eliminating all pairs for which either *dependent-expression* or *independent-expression* is NULL. The function is computed simultaneously during a single pass through the data. After eliminating NULL values, the following computation is then made, where *y* represents the *dependent-expression* and *x* represents the *independent-expression*:

```
REGR_COUNT( y, x ) * VAR_POP( y )
```

For more information about the statistical computation performed, see "Mathematical formulas for the aggregate functions" [*SQL Anywhere Server - SQL Usage*].

Syntax 2 represents usage as a window function in a SELECT statement. As such, elements of *window-spec* can be specified either in the function syntax (inline), or in conjunction with a WINDOW clause in the SELECT statement. See the *window-spec* definition provided in "WINDOW clause" on page 793.

For more information about using window functions in SELECT statements, including working examples, see "Window functions" [*SQL Anywhere Server - SQL Usage*].

See also

- "REGR_COUNT function [Aggregate]" on page 279
- "REGR_INTERCEPT function [Aggregate]" on page 280
- "REGR_COUNT function [Aggregate]" on page 279
- "REGR_AVGX function [Aggregate]" on page 276
- "REGR_AVGY function [Aggregate]" on page 277
- "REGR_SLOPE function [Aggregate]" on page 282
- "REGR_SXX function [Aggregate]" on page 284
- "REGR_SXY function [Aggregate]" on page 285

Standards and compatibility

- **SQL/2003** SQL foundation feature (T621) outside core SQL.

Example

The following example returns the value 26, 708, 672,843.3002.

```
SELECT REGR_SYY( Salary, ( YEAR( NOW( )) - YEAR( BirthDate ) ) )
FROM Employees;
```

REMAINDER function [Numeric]

Returns the remainder when one whole number is divided by another.

Syntax

REMAINDER(*dividend*, *divisor* **)**

Parameters

- **dividend** The dividend, or numerator of the division.
- **divisor** The divisor, or denominator of the division.

Returns

INTEGER

NUMERIC

Remarks

Alternatively, try using the MOD function.

See also

- "MOD function [Numeric]" on page 248

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value 2.

```
SELECT REMAINDER( 5, 3 );
```

REPEAT function [String]

Concatenates a string a specified number of times.

Syntax

REPEAT(*string-expression*, *integer-expression* **)**

Parameters

- **string-expression** The string to be repeated.
- **integer-expression** The number of times the string is to be repeated. If *integer-expression* is negative, an empty string is returned.

Returns

LONG VARCHAR

LONG NVARCHAR

Remarks

If the actual length of the result string exceeds the maximum for the return type, an error occurs. The result is truncated to the maximum string size allowed.

Alternatively, try using the REPLICATE function.

This function supports NCHAR inputs and/or outputs.

See also

- "REPLICATE function [String]" on page 290
- "String functions" on page 131

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value repeatrepeatrepeat.

```
SELECT REPEAT( 'repeat', 3 );
```

REPLACE function [String]

Replaces a string with another string, and returns the new results.

Syntax

REPLACE(*original-string*, *search-string*, *replace-string* **)**

Parameters

If any argument is NULL, the function returns NULL.

- **original-string** The string to be searched. This can be any length.
- **search-string** The string to be searched for and replaced with *replace-string*. This string is limited to 255 bytes. If *search-string* is an empty string, the original string is returned unchanged.
- **replace-string** The replacement string, which replaces *search-string*. This can be any length. If *replacement-string* is an empty string, all occurrences of *search-string* are deleted.

Returns

LONG VARCHAR

LONG NVARCHAR

Remarks

This function replaces all occurrences.

Comparisons are case-sensitive on case-sensitive databases.

This function supports NCHAR inputs and/or outputs.

See also

- "SUBSTRING function [String]" on page 318
- "CHARINDEX function [String]" on page 150
- "String functions" on page 131

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value xx.def.xx.ghi.

```
SELECT REPLACE( 'abc.def.abc.ghi', 'abc', 'xx' );
```

The following statement generates a result set containing ALTER PROCEDURE statements which, when executed, would repair stored procedures that reference a table that has been renamed. (To be useful, the table name must be unique.)

```
SELECT REPLACE(
   REPLACE( proc_defn, 'OldTableName', 'NewTableName' ),
   'CREATE PROCEDURE',
   'ALTER PROCEDURE')
FROM SYS.SYSPROCEDURE
WHERE proc_defn LIKE '%OldTableName%';
```

REPLICATE function [String]

Concatenates a string a specified number of times.

Syntax

REPLICATE(*string-expression*, *integer-expression* **)**

Parameters

- **string-expression** The string to be repeated.
- **integer-expression** The number of times the string is to be repeated.

Returns

LONG VARCHAR

LONG NVARCHAR

Remarks

If the actual length of the result string exceeds the maximum for the return type, an error occurs. The result is truncated to the maximum string size allowed.

Alternatively, try using the REPEAT function.

This function supports NCHAR inputs and/or outputs.

See also

- "REPEAT function [String]" on page 288
- "String functions" on page 131

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value repeatrepeatrepeat.

```
SELECT REPLICATE( 'repeat', 3 );
```

REVERSE function [String]

Returns the reverse of a character expression.

Syntax

REVERSE(*string-expression* **)**

Parameters

- **string-expression** The string to be reversed.

Returns

LONG VARCHAR

LONG NVARCHAR

Remarks

This function supports NCHAR inputs and/or outputs.

See also

- "String functions" on page 131

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value cba.

```
SELECT REVERSE( 'abc' );
```

REWRITE function [Miscellaneous]

Returns a rewritten SELECT, UPDATE, or DELETE statement.

Syntax

REWRITE(*select-statement* [**, 'ANSI'**] **)**

Parameters

- **select-statement** The SQL statement to which the rewrite optimizations are applied to generate the function's results.

Returns

LONG VARCHAR

Remarks

You can use the REWRITE function without the ANSI argument to help understand how the optimizer generated the access plan for a given query. In particular, you can find how SQL Anywhere has rewritten the conditions in the statement's WHERE, ON, and HAVING clauses, and then determine if applicable indexes exist that can be exploited to improve the request's execution time.

The statement that is returned by REWRITE may not match the semantics of the original statement. This is because several rewrite optimizations introduce internal mechanisms that cannot be translated directly into SQL. For example, the server's use of row identifiers to perform duplicate elimination cannot be translated into SQL.

The rewritten query from the REWRITE function is not intended to be executable. It is a tool for analyzing performance issues by showing what gets passed to the optimizer after the rewrite phase.

There are some rewrite optimizations that are not reflected in the output of REWRITE. They include LIKE optimization, optimization for minimum or maximum functions, upper/lower elimination, and predicate subsumption.

If ANSI is specified, REWRITE returns the ANSI equivalent to the statement. In this case, only the following rewrite optimizations are applied:

- Transact-SQL outer joins are rewritten as ANSI SQL outer joins.
- Duplicate correlation names are eliminated.
- KEY and NATURAL joins are rewritten as ANSI SQL joins.

See also

- “Semantic query transformations” [*SQL Anywhere Server - SQL Usage*]
- “extended_join_syntax option [database]” [*SQL Anywhere Server - Database Administration*]
- “Transact-SQL outer joins (*= or =*)” [*SQL Anywhere Server - SQL Usage*]
- “Key joins” [*SQL Anywhere Server - SQL Usage*]
- “Natural joins” [*SQL Anywhere Server - SQL Usage*]
- “Duplicate correlation names in joins (star joins)” [*SQL Anywhere Server - SQL Usage*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

In the following example, two rewrite optimizations are performed on a query. The first is the un-nesting of the subquery into a join between the Employees and SalesOrders tables. The second optimization simplifies the query by eliminating the primary key - foreign key join between Employees and SalesOrders. Part of this rewrite optimization is to replace the join predicate e.EmployeeID=s.SalesRepresentative with the predicate s.SalesRepresentative IS NOT NULL.

```
SELECT REWRITE( 'SELECT s.ID, s.OrderDate
    FROM SalesOrders s
    WHERE EXISTS ( SELECT *
       FROM Employees e
          WHERE e.EmployeeID = s.SalesRepresentative)' ) FROM dummy;
```

The query returns a single column result set containing the rewritten query:

```
'SELECT s.ID, s.OrderDate FROM SalesOrders s WHERE s.SalesRepresentative IS
NOT NULL'
```

The next example of REWRITE uses the ANSI argument.

```
SELECT REWRITE( 'SELECT DISTINCT s.ID, s.OrderDate, e.GivenName, e.EmployeeID
    FROM SalesOrders s, Employees e
          WHERE e.EmployeeID *= s.SalesRepresentative', 'ANSI' ) FROM dummy;
```

The result is the ANSI equivalent of the statement. In this case, the Transact-SQL outer join is converted to an ANSI outer join. The query returns a single column result set (broken into separate lines for readability):

```
'SELECT DISTINCT s.ID, s.OrderDate, e.GivenName, e.EmployeeID
 FROM Employees as e
 LEFT OUTER JOIN  SalesOrders as s
 ON e.EmployeeID = s.SalesRepresentative';
```

RIGHT function [String]

Returns the rightmost characters of a string.

Syntax

RIGHT(*string-expression*, *integer-expression* **)**

Parameters

- **string-expression** The string to be left-truncated.
- **integer-expression** The number of characters at the end of the string to return.

Returns

LONG VARCHAR

LONG NVARCHAR

Remarks

If the string contains multibyte characters, and the proper collation is being used, the number of bytes returned may be greater than the specified number of characters.

You can specify an *integer-expression* that is larger than the value in the column. In this case, the entire value is returned.

This function supports NCHAR inputs and/or outputs. Whenever possible, if the input string uses character length semantics the return value will be described in terms of character length semantics.

See also

- "LEFT function [String]" on page 235
- "International languages and character sets" [*SQL Anywhere Server - Database Administration*]
- "String functions" on page 131

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the last 5 characters of each Surname value in the Customers table.

```
SELECT RIGHT( Surname, 5) FROM Customers;
```

ROUND function [Numeric]

Rounds the *numeric-expression* to the specified *integer-expression* amount of places after the decimal point.

Syntax

ROUND(*numeric-expression*, *integer-expression* **)**

Parameters

- **numeric-expression** The number, passed into the function, to be rounded.
- **integer-expression** A positive integer specifies the number of significant digits to the right of the decimal point at which to round. A negative expression specifies the number of significant digits to the left of the decimal point at which to round.

Returns

NUMERIC

Remarks

The result of this function is either numeric or double. When there is a numeric result and the integer *integer-expression* is a negative value, the precision is increased by one.

See also

- "TRUNCNUM function [Numeric]" on page 328

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value 123.200.

```
SELECT ROUND( 123.234, 1 );
```

ROWID function [Miscellaneous]

Returns an unsigned 64-bit value that uniquely identifies a row within a table.

Syntax

ROWID(*correlation-name* **)**

Parameters

- **correlation-name** The correlation name of a table used in the query. The correlation name should refer to a base table, a temporary table, a global temporary table or a proxy table (permitted only when the underlying proxy server supports a similar function). The argument of the ROWID function should not refer to a view, derived table, common table expression or a procedure.

Returns

UNSIGNED BIGINT

Remarks

Returns the row identifier of the row in the table corresponding to the given correlation name.

The value returned by the function is not necessarily constant between queries as various operations performed on the database may result in changes to the row identifiers of a table. In particular, the REORGANIZE TABLE statement is likely to result in changes to row identifiers. Additionally, row identifiers may be reused after a row has been deleted. So, users should refrain from using the ROWID function in ordinary situations; retrieval by primary key value should be used instead. It is recommended that ROWID be used only in diagnostic situations.

Although the result of this function is an UNSIGNED BIGINT, the results of most arithmetic operations on this value have no particular meaning. For example, you should not expect that adding one to a row identifier will give you the row identifier of the next row. Also, only equality and IN predicates are sargable if they involve the use of ROWID. If necessary, predicates involving ROWID, such as `ROWID( T ) = `*`literal`*, may be used to cast to a 64-bit UNSIGNED INTEGER value. If the conversion cannot be performed a data exception will occur. If the value of *literal* is an invalid row identifier then the comparison predicate evaluates to FALSE.

The ROWID function cannot be used inside a CHECK constraint on either a table or a column, nor can it be used in the COMPUTE expression for a computed column.

See also

- “ROW_NUMBER function [Miscellaneous]” on page 296

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the row identifier of the row in Employee where id is equal to 105:

```
SELECT ROWID( Employees ) FROM Employees WHERE Employees.EmployeeID = 105;
```

The following statement returns a list of the locks on rows in the Employees table along with the contents of those rows:

```
SELECT *
  FROM sa_locks() S JOIN Employees WITH( NOLOCK )
     ON ROWID( Employees ) = S.row_identifier
  WHERE S.table_name = 'Employees';
```

ROW_NUMBER function [Miscellaneous]

Assigns a unique number to each row. Use this function instead of the NUMBER function.

Syntax

ROW_NUMBER( ) OVER (*window-spec* **)**

window-spec : see the Remarks section below

Returns

INTEGER

Remarks

Elements of *window-spec* can be specified either in the function syntax (inline), or in conjunction with a WINDOW clause in the SELECT statement. When used as a window function, you must specify an ORDER BY clause, you may specify a PARTITION BY clause, however, you can not specify a ROWS or RANGE clause. See the *window-spec* definition provided in “WINDOW clause” on page 793.

For more information about using window functions in SELECT statements, including working examples, see “Window functions” [*SQL Anywhere Server - SQL Usage*].

See also

- “NUMBER function [Miscellaneous]” on page 260
- “RANK function [Ranking]” on page 272
- “ROWID function [Miscellaneous]” on page 295

Standards and compatibility

- **SQL/2003** SQL/OLAP feature T612.

Example

The following example returns a result set that provides unique row numbers for each employee in New York and Utah. Because the query is ordered by Salary in descending order, the first row number is given to the employee with the highest salary in the data set. Although two employees have identical salaries, the tie is not resolved because the two employees are assigned unique row numbers.

```
SELECT Surname, Salary, State,
ROW_NUMBER() OVER (ORDER BY Salary DESC) "Rank"
FROM Employees WHERE State IN ('NY','UT');
```

Surname	Salary	State	Rank
Shishov	72995.000	UT	1
Wang	68400.000	UT	2
Cobb	62000.000	UT	3
Morris	61300.000	UT	4
Davidson	57090.000	NY	5
Martel	55700.000	NY	6
Blaikie	54900.000	NY	7
Diaz	54900.000	NY	8
Driscoll	48023.690	UT	9
Hildebrand	45829.000	UT	10
...	...	...	...
Lynch	24903.000	UT	19

RTRIM function [String]

Removes leading and trailing blanks from a string.

Syntax

RTRIM(*string-expression* **)**

Parameters

- **string-expression** The string to be trimmed.

Returns

VARCHAR

NVARCHAR

LONG VARCHAR

LONG NVARCHAR

Remarks

The actual length of the result is the length of the expression minus the number of characters removed. If all the characters are removed, the result is an empty string.

If the argument is null, the result is the null value.

This function supports NCHAR inputs and/or outputs.

See also

- "TRIM function [String]" on page 327
- "LTRIM function [String]" on page 244
- "String functions" on page 131

Standards and compatibility

- **SQL/2003** Vendor extension.

  The TRIM specifications defined by the SQL/2003 standard (LEADING and TRAILING) are supplied by the SQL Anywhere LTRIM and RTRIM functions respectively.

Example

The following statement returns the string Test Message, with all trailing blanks removed.

```
SELECT RTRIM( 'Test Message     ' );
```

SECOND function [Date and time]

Returns a second of the given date.

Syntax

SECOND(*datetime-expression* **)**

Parameters

- **datetime-expression** The datetime value.

Returns

SMALLINT

Remarks

Returns a number from 0 to 59 corresponding to the second component of the given datetime value.

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value 25.

```
SELECT SECOND( '1998-07-13 21:21:25' );
```

SECONDS function [Date and time]

The behavior of this function can vary depending on what you supply:

- If you give a single date, this function returns the number of seconds since 0000-02-29.

> **Note**
> 0000-02-29 is not meant to imply an actual date; it is the date used by the date algorithm.

- If you give two time stamps, this function returns the integer number of seconds between them. Instead, use the DATEDIFF function.
- If you give a date and an integer, this function adds the integer number of seconds to the specified timestamp. Instead, use the DATEADD function.

Syntax 1: integer

SECONDS([*datetime-expression*,] *datetime-expression* **)**

Syntax 2: timestamp

SECONDS(*datetime-expression*, *integer-expression* **)**

Parameters

- **datetime-expression** A date and time.
- **integer-expression** The number of seconds to be added to the *datetime-expression*. If *integer-expression* is negative, the appropriate number of minutes is subtracted from the datetime value. If you supply an integer expression, the *datetime-expression* must be explicitly cast as a datetime data type.

Returns

INTEGER

TIMESTAMP

See also

- "CAST function [Data type conversion]" on page 148
- "DATEADD function [Date and time]" on page 173
- "DATEDIFF function [Date and time]" on page 174

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statements return the value 14400, signifying that the second timestamp is 14400 seconds after the first.

```
SELECT SECONDS( '1999-07-13 06:07:12',
   '1999-07-13 10:07:12' );

SELECT DATEDIFF( second,
   '1999-07-13 06:07:12',
   '1999-07-13 10:07:12' );
```

The following statement returns the value 63062431632.

```
SELECT SECONDS( '1998-07-13 06:07:12' );
```

The following statements return the datetime 1999-05-12 21:05:12.000.

```
SELECT SECONDS( CAST( '1999-05-12 21:05:07'
AS TIMESTAMP ), 5);

SELECT DATEADD( second, 5, '1999-05-12 21:05:07' );
```

SET_BIT function [Bit array]

Set the value of a specific bit in a bit array.

Syntax

SET_BIT([*bit-expression*,]*bit-position* [**,** *value*]**)**

Parameters

- **bit-expression** The bit array in which to change the bit.
- **bit-position** The position of the bit to be set. This must be an unsigned integer.
- **value** The value to which the bit is to be set.

Returns

LONG VARBIT

Remarks

The default value of *bit-expression* is a bit array of length *bit-position*, containing all bits set to 0 (FALSE).

The default value of *value* is 1 (TRUE).

The result is NULL if any parameter is NULL.

The positions in the array are counted from the left side, starting at 1.

See also

- "GET_BIT function [Bit array]" on page 208
- "SET_BITS function [Aggregate]" on page 301
- "INTEGER data type" on page 90
- "Bitwise operators" on page 14
- "sa_get_bits system procedure" on page 873

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value 00100011:

```
SELECT SET_BIT( '00110011', 4 , 0);
```

The following statement returns the value 00111011:

```
SELECT SET_BIT( '00110011', 5 , 1);
```

The following statement returns the value 00111011:

```
SELECT SET_BIT( '00110011', 5 );
```

The following statement returns the value 00001:

```
SELECT SET_BIT( 5 );
```

SET_BITS function [Aggregate]

Creates a bit array where specific bits, corresponding to values from a set of rows, are set to 1 (TRUE).

Syntax

SET_BITS(*expression* **)**

Parameters

- **expression** The expression used to determine which bits to set to 1. This is typically a column name.

Returns

LONG VARBIT

Remarks

Rows where the specified values are NULL are ignored.

If there are no rows, NULL is returned.

The length of the result is the largest position that was set to 1.

The SET_BITS function is equivalent to, but faster than, the following statement:

```
SELECT BIT_OR( SET_BIT( expression ) )
FROM table;
```

See also

- “Bitwise operators” on page 14
- “GET_BIT function [Bit array]” on page 208
- “SET_BIT function [Bit array]” on page 300
- “sa_get_bits system procedure” on page 873

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statements return a bit array with the 2nd, 5th, and 10th bits set to 1 (or 0100100001):

```
CREATE TABLE t( r INTEGER );
INSERT INTO t values( 2 );
INSERT INTO t values( 5 );
INSERT INTO t values(10 );
SELECT SET_BITS( r ) FROM t;
```

SIGN function [Numeric]

Returns the sign of a number.

Syntax

SIGN(*numeric-expression* **)**

Parameters

- **numeric-expression** The number for which the sign is to be returned.

Returns

SMALL INT

Remarks

For negative numbers, the SIGN function returns -1.

For zero, the SIGN function returns 0.

For positive numbers, the SIGN function returns 1.

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value -1

```
SELECT SIGN( -550 );
```

SIMILAR function [String]

Returns a number indicating the similarity between two strings.

Syntax

SIMILAR(*string-expression-1*, *string-expression-2* **)**

Parameters

- **string-expression-1** The first string to be compared.
- **string-expression-2** The second string to be compared.

Returns

SMALL INT

Remarks

The function returns an integer between 0 and 100 representing the similarity between the two strings. The result can be interpreted as the percentage of characters matched between the two strings. A value of 100 indicates that the two strings are identical.

This function can be used to correct a list of names (such as customers). Some customers may have been added to the list more than once with slightly different names. Join the table to itself and produce a report of all similarities greater than 90 percent, but less than 100 percent.

The calculation performed for the SIMILAR function is more complex than just the number of characters that match.

See also

- “String functions” on page 131

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value 75, indicating that the two values are 75% similar.

```
SELECT SIMILAR( 'toast', 'coast' );
```

SIN function [Numeric]

Returns the sine of a number.

Syntax

SIN(*numeric-expression* **)**

Parameters

- **numeric-expression** The angle, in radians.

Returns

DOUBLE

Remarks

The SIN function returns the sine of the argument, where the argument is an angle expressed in radians. The SIN and ASIN functions are inverse operations.

This function converts its argument to DOUBLE, performs the computation in double-precision floating point, and returns a DOUBLE as the result.

See also

- "ASIN function [Numeric]" on page 138
- "COS function [Numeric]" on page 163
- "COT function [Numeric]" on page 164
- "TAN function [Numeric]" on page 322

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the SIN value of 0.52.

```
SELECT SIN( 0.52 );
```

SOAP_HEADER function [SOAP]

Returns a SOAP header entry, or an attribute value for a header entry of the SOAP request.

Syntax

SOAP_HEADER(*header-key* [*index*, *header-attribute*] **)**

Parameters

- **header-key** This VARCHAR parameter specifies the XML local name of the top level XML element for a given SOAP header entry.
- **index** This optional INTEGER parameter differentiates between SOAP header fields that have identical names. This can occur when multiple header entries have top level XML elements with the same localname. Usually, such elements have unique namespaces.
- **header-attribute** This optional VARCHAR parameter can be any attribute node within a header entry element, including:
  - **@namespace** A special SQL Anywhere attribute used to access the namespace of the given header entry.

- **mustUnderstand** A SOAP 1.1 header entry attribute indicating whether a header entry is mandatory or optional for the recipient to process.
- **encodingStyle** A SOAP 1.1 header entry attribute indicating the encoding style. This attribute may be accessed, but it is not used internally by SQL Anywhere.
- **actor** A SOAP 1.1 header entry attribute indicating the intended recipient of a header entry by specifying the recipient's URL.

Returns

LONG VARCHAR

Remarks

This function may be used with a single parameter *header-key* to return a header entry. A header entry is an XML string representation of an element, and all its sub-elements, contained within a SOAP header.

This function may also be used to extract a header entry attribute by specifying the optional *index* and *header-attribute* parameters.

This function returns the value of the named SOAP header field, or NULL if not called from a SOAP service. It is used when processing a SOAP request via a web service.

If a header for the given *header-key* does not exist, the return value is NULL.

See also

- "-xs server option" [*SQL Anywhere Server - Database Administration*]
- "Working with SOAP headers" [*SQL Anywhere Server - Programming*]
- "SQL Anywhere web services" [*SQL Anywhere Server - Programming*]
- "NEXT_SOAP_HEADER function [SOAP]" on page 258
- "sa_set_soap_header system procedure" on page 948
- "Web services functions" on page 130
- "Web services system procedures" on page 830

Standards and compatibility

- **SQL/2003** Vendor extension.

SORTKEY function [String]

Generates sort key values. That is, values that can be used to sort character strings based on alternate collation rules.

Syntax

SORTKEY(*string-expression*
[**,** { *collation-id*
| *collation-name*[**(***collation-tailoring-string***)**] }]
)

Parameters

- **string-expression** The string expression must contain characters that are encoded in the database's character set.

  If *string-expression* is an empty string, the SORTKEY function returns a zero-length binary value. If *string-expression* is NULL, the SORTKEY function returns a NULL value. An empty string has a different sort order value than a NULL string from a database column.

  The maximum length of the string that the SORTKEY function can handle is 254 bytes. Any longer part is ignored.

- **collation-name** A string or a character variable that specifies the name of the sort order to use. You can also specify the alias char_collation, or, equivalently, db_collation, to generate sortkeys as used by the CHAR collation in use by the database. Similarly, you can specify the alias nchar_collation to generate sortkeys as used by the NCHAR collation in use by the database.

- **collation-id** A variable, integer constant, or string that specifies the ID number of the sort order to use. If you do not specify *collation-id*, the default is Default Unicode multilingual. For a list of valid collations, see "Supported and alternate collations" [*SQL Anywhere Server - Database Administration*], and "Recommended character sets and collations" [*SQL Anywhere Server - Database Administration*].

- **collation-tailoring-string** Optionally, you can specify collation tailoring options (*collation-tailoring-string*) for additional control over the sorting and comparing of characters. These options take the form of keyword=value pairs assembled in parentheses, following the collation name. For example, `'UCA(locale=es;case=LowerFirst;accent=respect)'`. The syntax for specifying these options is identical to the syntax defined for the COLLATION clause of the CREATE DATABASE statement. See "Collation tailoring options" [*SQL Anywhere Server - Database Administration*].

Note
All the collation tailoring options are supported when specifying the UCA collation. For all other collations, only case sensitivity tailoring is supported.

Returns

BINARY

Remarks

The SORTKEY function generates values that can be used to order results based on predefined sort order behavior. This allows you to work with character sort order behaviors that may not be available from the database collation. The returned value is a binary value that contains coded sort order information for the input string that is retained from the SORTKEY function. For example, you can store the values returned by the SORTKEY function in a column with the source character string. When you want to retrieve the character data in the desired order, the SELECT statement only needs to include an ORDER BY clause on the columns that contain the results of running the SORTKEY function.

The SORTKEY function guarantees that the values it returns for a given set of sort order criteria work for the binary comparisons that are performed on VARBINARY data types.

Generating sortkeys for queries can be expensive. As an alternative for frequently requested sortkeys, consider creating a computed column to hold the sortkey values, and then referencing that column in the ORDER BY clause of the query.

The input of the SORTKEY function can generate up to six bytes of sort order information for each input character. The output of the SORTKEY function is of type VARBINARY and has a maximum length of 1024 bytes.

When specifying UCA for the collation during sort key generation, by default, collation tailorings are accent and case sensitive. For example, when UCA is specified by itself, the default tailoring applied is equivalent to `'UCA(case=UpperFirst;accent=Respect;punct=Primary)'`.

If a different tailoring is provided in the second parameter to SORTKEY, those settings override the default settings. For example, the following two statements are equivalent:

```
SELECT SORTKEY( 'abc', 'UCA(accent=Ignore)' );
SELECT SORTKEY( 'abc', 'UCA(case=UpperFirst;accent=Ignore;punct=Primary)' );
```

When specifying a non-UCA collation, by default, collation tailorings are also accent and case sensitive. However, for non-UCA collations, only the case sensitivity can be overridden using a collation tailoring. For example:

```
SELECT SORTKEY( 'abc', '1252LATIN1(case=Respect)' );
```

If the database was created without specifying tailoring options (for example, `dbinit -c -zn uca mydb.db`), the following two clauses may generate different sort orders, even if the database collation name is specified for the SORTKEY function:

```
ORDER BY string-expression
ORDER BY SORTKEY( string-expression, database-collation-name )
```

This is because the default tailoring settings used for database creation and for the SORTKEY function are different. To get the same behavior from SORTKEY as for the database collation, either provide a tailoring syntax for *collation-tailoring-string* that matches the settings for the database collation, or specify db_collation for *collation-name*. For example:

```
SORTKEY( expression, 'db_collation' )
```

Note
Sort key values are dependent on the version of SQL Anywhere used to create the database file. For example, the sort key values created by versions of SQL Anywhere prior to 10.0.0, SQL Anywhere version 10, and SQL Anywhere version 11 are different. This may be an issue for your applications if sort key values created by one version of SQL Anywhere are used in a database created by a different version of SQL Anywhere.

It is recommended that you regenerate the sort key values that are stored in the database when using unload and reload to upgrade your SQL Anywhere software.

See also

- "sort_collation option [database]" [*SQL Anywhere Server - Database Administration*]
- "COMPARE function [String]" on page 153
- "International languages and character sets" [*SQL Anywhere Server - Database Administration*]
- "String functions" on page 131

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statements queries the Employees table and returns the FirstName and Surname of all employees, sorted by the sortkey values for the Surname column using the dict collation (Latin-1, English, French, German dictionary).

```
SELECT Surname, GivenName FROM Employees ORDER BY SORTKEY( Surname, 'dict' );
```

The following example returns the sortkey value for abc, using the UCA collation and tailoring options.

```
SELECT SORTKEY( 'abc', 'UCA(locale=es;case=LowerFirst;accent=respect)' );
```

SOUNDEX function [String]

Returns a number representing the sound of a string.

Syntax

SOUNDEX(*string-expression* **)**

Parameters

- **string-expression** The string to be evaluated.

Returns

SMALLINT

Remarks

The SOUNDEX function value for a string is based on the first letter and the next three consonants other than H, Y, and W. Vowels in *string-expression* are ignored unless they are the first letter of the string. Doubled letters are counted as one letter. For example, the word apples is based on the letters A, P, L, and S.

Multibyte characters are ignored by the SOUNDEX function.

Although it is not perfect, the SOUNDEX function normally returns the same number for words that sound similar and that start with the same letter.

The SOUNDEX function works best with English words. It is less useful for other languages.

See also

- "String functions" on page 131

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns two identical numbers, 3827, representing the sound of each name.

```
SELECT SOUNDEX( 'Smith' ), SOUNDEX( 'Smythe' );
```

SPACE function [String]

Returns a specified number of spaces.

Syntax

SPACE(*integer-expression* **)**

Parameters

- **integer-expression** The number of spaces to return.

Returns

LONG VARCHAR

Remarks

If *integer-expression* is negative, a null string is returned.

See also

- "String functions" on page 131

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns a string containing 10 spaces.

```
SELECT SPACE( 10 );
```

SQLDIALECT function [Miscellaneous]

Returns either Watcom-SQL or Transact-SQL, to indicate the SQL dialect of a statement.

Syntax

SQLDIALECT(*sql-statement-string* **)**

Parameters

- **sql-statement-string** The SQL statement that the function uses to determine its dialect.

Returns

LONG VARCHAR

See also

- "TRANSACTSQL function [Miscellaneous]" on page 327
- "WATCOMSQL function [Miscellaneous]" on page 339

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the string Transact-SQL.

```
SELECT
   SQLDIALECT( 'SELECT EmployeeName = Surname FROM Employees' )
FROM dummy;
```

SQLFLAGGER function [Miscellaneous]

Returns the conformity of a given SQL statement to a specified standard.

Syntax

SQLFLAGGER(*sql-standard-string*, *sql-statement-string* **)**

Parameters

- **sql-standard-string** The standard level against which to test compliance. Possible values are the same as for the sql_flagger_error_level database option:
  - **SQL:2003/Core** Test for conformance to core SQL/2003 syntax.
  - **SQL:2003/Package** Test for conformance to full SQL/2003 syntax.
  - **SQL:1999/Core** Test for conformance to core SQL/1999 syntax.
  - **SQL:1999/Package** Test for conformance to full SQL/1999 syntax.
  - **SQL:1992/Entry** Test for conformance to entry-level SQL/1992 syntax.
  - **SQL:1992/Intermediate** Test for conformance to intermediate-level SQL/1992 syntax.
  - **SQL:1992/Full** Test for conformance to full-SQL/1992 syntax.
  - **Ultralite** Test for conformance to UltraLite.
- **sql-statement-string** The SQL statement to check for conformance.

Returns

LONG VARCHAR

See also

- "sql_flagger_error_level option [compatibility]" [*SQL Anywhere Server - Database Administration*]
- "SQL preprocessor" [*SQL Anywhere Server - Programming*]
- "sa_ansi_standard_packages system procedure" on page 841
- "Testing SQL compliance using the SQL Flagger" [*SQL Anywhere Server - SQL Usage*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement shows an example of the message that is returned when a disallowed extension is found:

```
SELECT SQLFLAGGER(
   'SQL:2003/Package', 'SELECT top 1 dummy_col FROM sys.dummy ORDER BY
dummy_col' );
```

This statement returns the message `'0AW03 Disallowed language extension detected in syntax near 'top' on line 1'`.

The following statement returns `'00000'` because it does not contain any disallowed extensions:

```
SELECT SQLFLAGGER( 'SQL:2003/Package', 'SELECT dummy_col FROM sys.dummy' );
```

SQRT function [Numeric]

Returns the square root of a number.

Syntax

SQRT(*numeric-expression* **)**

Parameters

- **numeric-expression** The number for which the square root is to be calculated.

Returns

DOUBLE

Remarks

This function converts its argument to DOUBLE, performs the computation in double-precision floating point, and returns a DOUBLE as the result.

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value 3.

```
SELECT SQRT( 9 );
```

STDDEV function [Aggregate]

An alias for STDDEV_SAMP. See "STDDEV_SAMP function [Aggregate]" on page 313.

STDDEV_POP function [Aggregate]

Computes the standard deviation of a population consisting of a numeric-expression, as a DOUBLE.

Syntax 1

STDDEV_POP(*numeric-expression* **)**

Syntax 2

STDDEV_POP(*numeric-expression* **) OVER (** *window-spec* **)**

window-spec : see Syntax 2 instructions in the Remarks section below

Parameters

- **numeric-expression** The expression whose population-based standard deviation is calculated over a set of rows. The expression is commonly a column name.

Returns

DOUBLE

Remarks

This function converts its argument to DOUBLE, and performs the computation in double-precision floating point.

The population-based standard deviation (s) is computed according to the following formula:

$s = [(1/N) * SUM(x_I - mean(x))^2]^{½}$

This standard deviation does not include rows where *numeric-expression* is NULL. It returns NULL for a group containing no rows.

For more information about the statistical computation performed, see "Mathematical formulas for the aggregate functions" [*SQL Anywhere Server - SQL Usage*].

Syntax 2 represents usage as a window function in a SELECT statement. As such, elements of *window-spec* can be specified either in the function syntax (inline), or in conjunction with a WINDOW clause in the SELECT statement. See the *window-spec* definition provided in "WINDOW clause" on page 793.

For more information about using window functions in SELECT statements, including working examples, see "Window functions" [*SQL Anywhere Server - SQL Usage*].

See also

- "Aggregate functions" on page 124

Standards and compatibility

- **SQL/2003** SQL foundation feature (T621) outside core SQL.

Example

The following statement lists the average and variance in the number of items per order in different time periods:

```
SELECT YEAR( ShipDate ) AS Year,
    QUARTER( ShipDate ) AS Quarter,
    AVG( Quantity ) AS Average,
    STDDEV_POP( quantity ) AS Variance
FROM SalesOrderItems
GROUP BY Year, Quarter
ORDER BY Year, Quarter;
```

Year	Quarter	Average	Variance
2000	1	25.775148	14.2794...
2000	2	27.050847	15.0270...
...	...	...	...

STDDEV_SAMP function [Aggregate]

Computes the standard deviation of a sample consisting of a numeric-expression, as a DOUBLE.

Syntax 1

STDDEV_SAMP(*numeric-expression* **)**

Syntax 2

STDDEV_SAMP(*numeric-expression* **) OVER (** *window-spec* **)**

window-spec : see Syntax 2 instructions in the Remarks section below

Parameters

- **numeric-expression** The expression whose sample-based standard deviation is calculated over a set of rows. The expression is commonly a column name.

Returns

DOUBLE

Remarks

This function converts its argument to DOUBLE, and performs the computation in double-precision floating point.

The standard deviation (s) is computed according to the following formula, which assumes a normal distribution:

$s = [\ (1/(\ N - 1\ ))\ *\ SUM(\ x_I - mean(\ x\ )\ )^2\ ]^{1/2}$

This standard deviation does not include rows where *numeric-expression* is NULL. It returns NULL for a group containing either 0 or 1 rows.

For more information about the statistical computation performed, see "Mathematical formulas for the aggregate functions" [*SQL Anywhere Server - SQL Usage*].

Syntax 2 represents usage as a window function in a SELECT statement. As such, elements of *window-spec* can be specified either in the function syntax (inline), or in conjunction with a WINDOW clause in the SELECT statement. See the *window-spec* definition provided in "WINDOW clause" on page 793.

For more information about using window functions in SELECT statements, including working examples, see "Window functions" [*SQL Anywhere Server - SQL Usage*].

See also

- "Aggregate functions" on page 124

Standards and compatibility

- **SQL/2003** SQL foundation feature (T621) outside core SQL.

Example

The following statement lists the average and variance in the number of items per order in different time periods:

```
SELECT YEAR( ShipDate ) AS Year,
    QUARTER( ShipDate ) AS Quarter,
    AVG( Quantity ) AS Average,
    STDDEV_SAMP( quantity ) AS Variance
FROM SalesOrderItems
GROUP BY Year, Quarter
ORDER BY Year, Quarter;
```

Year	Quarter	Average	Variance
2000	1	25.775148	14.3218...
2000	2	27.050847	15.0696...
...	...	...	...

STR function [String]

Returns the string equivalent of a number.

Syntax

STR(*numeric-expression* [, *length* [, *decimal*]] **)**

Parameters

- **numeric-expression** Any approximate numeric (float, real, or double precision) expression between -1E126 and 1E127.
- **length** The number of characters to be returned (including the decimal point, all digits to the right and left of the decimal point, and blanks). The default is 10.
- **decimal** The number of decimal digits to be returned. The default is 0.

Returns

VARCHAR

Remarks

If the integer portion of the number cannot fit in the length specified, then the result is a string of the specified length containing all asterisks. For example, the following statement returns ***.

```
SELECT STR( 1234.56, 3 );
```

Note
The maximum length that is supported is 128. Any length that is not between 1 and 128 yields a result of NULL.

See also

- "String functions" on page 131

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns a string of six spaces followed by 1235, for a total of ten characters.

```
SELECT STR( 1234.56 );
```

The following statement returns the result 1234.6.

```
SELECT STR( 1234.56, 6, 1 );
```

STRING function [String]

Concatenates one or more strings into one large string.

Syntax

STRING(*string-expression* **[, ...])**

Parameters

- **string-expression** The string to be evaluated.

  If only one argument is supplied, it is converted into a single expression. If more than one argument is supplied, they are concatenated into a single string.

Returns

LONG VARCHAR

LONG NVARCHAR

LONG BINARY

Remarks

Numeric or date parameters are converted to strings before concatenation. The STRING function can also be used to convert any single expression to a string by supplying that expression as the only parameter.

If all parameters are NULL, STRING returns NULL. If any parameters are non-NULL, then any NULL parameters are treated as empty strings.

See also

- "String functions" on page 131

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value *testing123*.

```
SELECT STRING( 'testing', NULL, 123 );
```

STRTOUUID function [String]

Converts a string value to a unique identifier (UUID or GUID) value.

Not needed in newer databases
In databases created before version 9.0.2, the UNIQUEIDENTIFIER data type was defined as a user-defined data type and the STRTOUUID and UUIDTOSTR functions were needed to convert between binary and string representations of UUID values.

In databases created using version 9.0.2 or later, the UNIQUEIDENTIFIER data type was changed to a native data type and SQL Anywhere carries out conversions as needed. You do not need to use STRTOUUID and UUIDTOSTR functions with these versions.

For more information, see "UNIQUEIDENTIFIER data type" on page 106.

Syntax

STRTOUUID(*string-expression* **)**

Parameters

- **string-expression** A string in the format *xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx*.

Returns

UNIQUEIDENTIFIER

Remarks

Converts a string in the format *xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx* , where *x* is a hexadecimal digit, to a unique identifier value.

If the string is not a valid UUID string, a conversion error is returned unless the conversion_error option is set to OFF, in which case it returns NULL.

This function is useful for inserting UUID values into a database.

This function supports NCHAR inputs and/or outputs.

See also

- "UUIDTOSTR function [String]" on page 335
- "NEWID function [Miscellaneous]" on page 252
- "String functions" on page 131

Standards and compatibility

- **SQL/2003** Vendor extension.

STUFF function [String]

Deletes multiple characters from one string and replaces them with another string.

Syntax

STUFF(*string-expression-1*, *start*, *length*, *string-expression-2* **)**

Parameters

- **string-expression-1** The string to be modified by the STUFF function.
- **start** The character position at which to begin deleting characters. The first character in the string is position 1.
- **length** The number of characters to delete.
- **string-expression-2** The string to be inserted. To delete a portion of a string using the STUFF function, use a replacement string of NULL.

Returns

LONG NVARCHAR

Remarks

This function supports NCHAR inputs and/or outputs.

See also

- “INSERTSTR function [String]” on page 229
- “String functions” on page 131

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value chocolate pie.

```
SELECT STUFF( 'chocolate cake', 11, 4, 'pie' );
```

SUBSTRING function [String]

Returns a substring of a string.

Syntax

{ **SUBSTRING** | **SUBSTR** } **(** *string-expression*, *start*
[, *length*] **)**

Parameters

- **string-expression** The string from which a substring is to be returned.
- **start** The start position of the substring to return, in characters.
- **length** The length of the substring to return, in characters. If *length* is specified, the substring is restricted to that length.

Returns

LONG BINARY

LONG VARCHAR

LONG NVARCHAR

Remarks

The behavior of this function depends on the setting of the ansi_substring database option. When the ansi_substring option is set to On (the default), the behavior of the SUBSTRING function corresponds to ANSI/ISO SQL/2003 behavior. The behavior is as follows:

ansi_substring option setting	start value	length value
On	The first character in the string is at position 1. A negative or zero start offset is treated as if the string were padded on the left with non-characters.	A positive *length* specifies that the substring ends *length* characters to the right of the starting position. A negative *length* returns an error.
Off	The first character in the string is at position 1. A negative starting position specifies a number of characters from the end of the string instead of the beginning. If *start* is zero and length is non-negative, a start value of 1 is used. If start is zero and *length* is negative, a start value of -1 is used.	A positive *length* specifies that the substring ends *length* characters to the right of the starting position. A negative *length* returns at most *length* characters up to, and including, the starting position, from the left of the starting position.

If *string-expression* is of binary data type, the SUBSTRING function behaves as BYTE_SUBSTR.

To obtain characters at the end of a string, use the RIGHT function.

This function supports NCHAR inputs and/or outputs. Whenever possible, if the input string uses character length semantics the return value is described in terms of character length semantics.

See also

- "BYTE_SUBSTR function [String]" on page 147
- "LEFT function [String]" on page 235
- "RIGHT function [String]" on page 293
- "CHARINDEX function [String]" on page 150
- "String functions" on page 131

Standards and compatibility

- **SQL/2003** Core feature.

Example

The following table shows the values returned by the SUBSTRING function.

Example	Result
SUBSTRING('front yard', 1, 4)	fron
SUBSTRING('back yard', 6, 4)	yard
SUBSTR('abcdefgh', 0, -2)	Returns an error if ansi_substring is On

Example	Result
SUBSTR('abcdefgh', -2, 2)	Returns an empty string if ansi_substring is On

SUM function [Aggregate]

Returns the total of the specified expression for each group of rows.

Syntax 1

SUM(*expression* | **DISTINCT** *expression* **)**

Syntax 2

SUM(*expression* **) OVER (** *window-spec* **)**

window-spec : see Syntax 2 instructions in the Remarks section below

Parameters

- **expression** The object to be summed. This is commonly a column name.
- **DISTINCT expression** Computes the sum of the unique values of *expression* in the input.

Returns

INTEGER

DOUBLE

NUMERIC

Remarks

Rows where the specified expression is NULL are not included.

Returns NULL for a group containing no rows.

Syntax 2 represents usage as a window function in a SELECT statement. As such, elements of *window-spec* can be specified either in the function syntax (inline), or in conjunction with a WINDOW clause in the SELECT statement. See the *window-spec* definition provided in "WINDOW clause" on page 793.

For more information about using window functions in SELECT statements, including working examples, see "Window functions" [*SQL Anywhere Server - SQL Usage*].

See also

- "COUNT function [Aggregate]" on page 164
- "AVG function [Aggregate]" on page 140

Standards and compatibility

- **SQL/2003** Core feature. Syntax 2 is feature T611.

Example

The following statement returns the value 3749146.740.

```
SELECT SUM( Salary )
FROM Employees;
```

SUSER_ID function [System]

Returns the numeric user ID for the specified user name.

Syntax

SUSER_ID([*user-name*] **)**

Parameters

- **user-name** The user name for the user ID you are searching for.

Returns

INT

Remarks

If you do not specify *user-name*, the ID of the current user is returned.

See also

- "SUSER_NAME function [System]" on page 321
- "USER_ID function [System]" on page 333

Standards and compatibility

- **SQL/2003** Transact-SQL extension.

Example

The following statement returns the ID for the GROUPO user.

```
SELECT SUSER_ID( 'GROUPO' );
```

SUSER_NAME function [System]

Returns the user name for the specified user ID.

Syntax

SUSER_NAME([*user-id*] **)**

Parameters

- **user-id** The user ID of the user you are searching for.

Returns

LONG VARCHAR

Remarks

If you do not specify *user-id*, the user name of the current user is returned.

See also

- "SUSER_ID function [System]" on page 321
- "USER_NAME function [System]" on page 334

Standards and compatibility

- **SQL/2003** Transact-SQL extension.

Example

The following statement returns the user name for a user with ID 101.

```
SELECT SUSER_NAME( 101 );
```

TAN function [Numeric]

Returns the tangent of a number.

Syntax

TAN(*numeric-expression* **)**

Parameters

- **numeric-expression** An angle, in radians.

Returns

DOUBLE

Remarks

The ATAN and TAN functions are inverse operations.

This function converts its argument to DOUBLE, performs the computation in double-precision floating point, and returns a DOUBLE as the result.

See also

- "COS function [Numeric]" on page 163
- "SIN function [Numeric]" on page 303

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value of the tan of 0.52.

```
SELECT TAN( 0.52 );
```

TEXTPTR function [Text and image]

Returns the 16-byte binary pointer to the first page of the specified text column.

Syntax

TEXTPTR(*column-name* **)**

Parameters

- **column-name** The name of a text column.

Returns

BINARY

Remarks

This function is included for Transact-SQL compatibility.

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

Use TEXTPTR to locate the text column, copy, associated with au_id 486-29-1786 in the author's blurbs table.

The text pointer is put into a local variable @val and supplied as a parameter to the readtext command, which returns 5 bytes, starting at the second byte (offset of 1).

```
DECLARE @val VARBINARY(16)
SELECT @val = TEXTPTR(copy)
FROM blurbs
WHERE au_id = "486-29-1786"
READTEXT blurbs.copy @val 1 5;
```

TO_CHAR function [String]

Converts character data from any supported character set into the CHAR character set for the database.

Syntax

TO_CHAR(*string-expression* [**,** *source-charset-name*] **)**

Parameters

- **string-expression** The string to be converted.
- **source-charset-name** The character set of the string.

Returns

LONG VARCHAR

Remarks

If *source-charset-name* is specified, then this function is equivalent to:

```
CAST( CSCONVERT( CAST( string-expression AS BINARY ),
 'db_charset', source-charset-name )
   AS CHAR );
```

For more information about db_charset, see "CSCONVERT function [String]" on page 168.

If *source-charset-name* is not specified, then this function is equivalent to:

```
CAST( string-expression AS CHAR );
```

See also

- "Recommended character sets and collations" [*SQL Anywhere Server - Database Administration*]
- "CONNECTION_EXTENDED_PROPERTY function [String]" on page 156
- "CSCONVERT function [String]" on page 168
- "NCHAR function [String]" on page 252
- "TO_NCHAR function [String]" on page 324
- "UNICODE function [String]" on page 331
- "UNISTR function [String]" on page 331

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

If you have a BINARY value containing data in the cp850 character set, the following statement converts the data to the CHAR character set and data type:

```
SELECT TO_CHAR( 'cp850_data', 'cp850' );
```

TO_NCHAR function [String]

Converts character data from any supported character set into the NCHAR character set.

Syntax

TO_NCHAR(*string-expression* [**,** *source-charset-name*] **)**

Parameters

- **string-expression** The string to be converted

- **source-charset-name** The character set of the string.

Returns

LONG NVARCHAR

Remarks

If *source-charset-name* is specified then this function is equivalent to:

```
CAST( CSCONVERT( CAST( string-expression AS BINARY ),
 'nchar_charset', source-charset-name )
 AS NCHAR );
```

For more information about nchar_charset, see "CSCONVERT function [String]" on page 168.

If *source-charset-name* is not provided then this function is equivalent to:

```
CAST( string-expression AS NCHAR );
```

See also

- "Recommended character sets and collations" [*SQL Anywhere Server - Database Administration*]
- "CONNECTION_EXTENDED_PROPERTY function [String]" on page 156
- "CSCONVERT function [String]" on page 168
- "NCHAR function [String]" on page 252
- "TO_CHAR function [String]" on page 323
- "UNICODE function [String]" on page 331
- "UNISTR function [String]" on page 331

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

If you have a BINARY value containing data in the cp850 character set, the following example to converts the data to the NCHAR character set and data type:

```
SELECT TO_NCHAR( 'cp850_data', 'cp850' );
```

TODAY function [Date and time]

Returns the current date.

Syntax

TODAY(*)

Returns

DATE

Remarks

Use this syntax in place of the historical CURRENT DATE function.

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statements return the current day according to the system clock.

```
SELECT TODAY( * );
SELECT CURRENT DATE;
```

TRACEBACK function [Miscellaneous]

Returns a string containing a traceback of the procedures and triggers that were executing when the most recent exception (error) occurred.

Syntax

TRACEBACK(*)

Returns

LONG VARCHAR

Remarks

This is useful for debugging procedures and triggers

Standards and compatibility

- **SQL/2003** Transact-SQL extension.

Example

To use the traceback function, enter the following after an error occurs while executing a procedure:

```
SELECT TRACEBACK( * );
```

TRACED_PLAN function [Miscellaneous]

This function is used by Sybase Central to generate a graphical plan for a query using tracing data.

Syntax

TRACED_PLAN(*logging_session_id*, *query_id* **)**

Parameters

- **logging_session_id** Combined with *query_id*, this INTEGER parameter identifies a row from the sa_diagnostic_query table for which to generate the plan.
- **query_id** Combined with *logging_session_id*, this INTEGER parameter identifies a row from the sa_diagnostic_query table for which to generate the plan.

Returns

LONG VARCHAR

Remarks

This function is for use by Sybase Central.

See also

- "sa_diagnostic_query table" on page 818

Standards and compatibility

- **SQL/2003** Vendor extension.

TRANSACTSQL function [Miscellaneous]

Takes a Watcom-SQL statement and rewrites it in the Transact-SQL dialect.

Syntax

TRANSACTSQL(*sql-statement-string* **)**

Parameters

- **sql-statement-string** The SQL statement that is to be rewritten in Transact-SQL.

Returns

LONG VARCHAR

See also

- "SQLDIALECT function [Miscellaneous]" on page 309
- "WATCOMSQL function [Miscellaneous]" on page 339

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the string `'SELECT EmployeeName=empl_name FROM Employees'`.

```
SELECT TRANSACTSQL( 'SELECT empl_name as EmployeeName FROM Employees' ) FROM
dummy;
```

TRIM function [String]

Removes leading and trailing blanks from a string.

Syntax

TRIM(*string-expression* **)**

Parameters

- **string-expression** The string to be trimmed.

Returns

VARCHAR

NVARCHAR

LONG VARCHAR

LONG NVARCHAR

Remarks

This function supports NCHAR inputs and/or outputs.

See also

- "LTRIM function [String]" on page 244
- "RTRIM function [String]" on page 297
- "String functions" on page 131

Standards and compatibility

- **SQL/2003** The TRIM function is a SQL/2003 core feature.

  SQL Anywhere does not support the additional parameters *trim specification* and *trim character*, as defined in SQL/2003. The SQL Anywhere implementation of TRIM corresponds to a TRIM specification of BOTH.

  For the other TRIM specifications defined by the SQL/2003 standard (LEADING and TRAILING), SQL Anywhere supplies the LTRIM and RTRIM functions respectively.

Example

The following statement returns the value chocolate with no leading or trailing blanks.

```
SELECT TRIM( '   chocolate   ' );
```

TRUNCNUM function [Numeric]

Truncates a number at a specified number of places after the decimal point.

Syntax

{ **TRUNCNUM** | **"TRUNCATE"** }**(** *numeric-expression*, *integer-expression* **)**

Parameters

- **numeric-expression** The number to be truncated.

- **integer-expression** A positive integer specifies the number of significant digits to the right of the decimal point at which to round. A negative expression specifies the number of significant digits to the left of the decimal point at which to round.

Returns

NUMERIC

Remarks

You should use the TRUNCNUM function, not the TRUNCATE function, when truncating numbers.

Use of the TRUNCATE statement is not recommended because the word truncate is a keyword, and therefore requires you to either set the quoted_identifier option to OFF, or put quotes around the word TRUNCATE.

See also

- "ROUND function [Numeric]" on page 294
- "quoted_identifier option [compatibility]" [*SQL Anywhere Server - Database Administration*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value 600.

```
SELECT TRUNCNUM( 655, -2 );
```

The following statement: returns the value 655.340.

```
SELECT TRUNCNUM( 655.348, 2 );
```

TSEQUAL function [System] (deprecated)

Compares two timestamp values and returns whether they are the same.

Syntax

TSEQUAL (*timestamp1*, *timestamp2* **)**

Parameters

- **timestamp1** Timestamp expression.
- **timestamp2** Timestamp expression.

Returns

BIT

Remarks

The TSEQUAL function can only be used in a WHERE clause and is most commonly used as part of an UPDATE statement.

If *timestamp1* is equal to *timestamp2*, a row has changed since it was fetched. If the row changed, its timestamp has been modified and the TSEQUAL function returns FALSE. When the TSEQUAL function returns FALSE, the application determines that no rows were updated and assumes that the row was modified by another user. The updated row is re-fetched.

You can use the TSEQUAL function to determine whether a row has been changed since it was fetched.

See also

- "The data type of a timestamp column" [*SQL Anywhere Server - SQL Usage*]
- "TIMESTAMP special value" on page 63
- "The special Transact-SQL timestamp column and data type" [*SQL Anywhere Server - SQL Usage*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

Suppose you create a TIMESTAMP column Products.LastUpdated to store the timestamp for the last time the row was updated. The following UPDATE statement uses the TSEQUAL function to determine whether to update the row. If the value of LastUpdated is '2010/12/25 11:08:34.173226', the row is updated.

```
UPDATE Products
SET Color = 'Yellow'
WHERE ID = '300'
AND TSEQUAL( LastUpdated, '2010/12/25 11:08:34.173226' );
```

UCASE function [String]

Converts all characters in a string to uppercase. This function is identical to the UPPER function.

Syntax

UCASE(*string-expression* **)**

Parameters

- **string-expression** The string to be converted to uppercase.

Returns

VARCHAR

NVARCHAR

LONG VARCHAR

LONG NVARCHAR

See also

- "UPPER function [String]" on page 333
- "LCASE function [String]" on page 235
- "String functions" on page 131

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value CHOCOLATE.

```
SELECT UCASE( 'ChocoLate' );
```

UNICODE function [String]

Returns an integer containing the Unicode code point of the first character in the string, or NULL if the first character is not a valid encoding.

Syntax

UNICODE(*nchar-string-expression* **)**

Parameters

- **nchar-string-expression** The NCHAR string whose first character is to be converted to an integer.

Returns

INT

See also

- "CONNECTION_EXTENDED_PROPERTY function [String]" on page 156
- "NCHAR function [String]" on page 252
- "TO_CHAR function [String]" on page 323
- "TO_NCHAR function [String]" on page 324
- "UNISTR function [String]" on page 331

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following example returns the integer 65536:

```
SELECT UNICODE(UNISTR( '\u010000data' ));
```

UNISTR function [String]

Converts a string containing characters and Unicode escape sequences to an NCHAR string.

Syntax

UNISTR(*string-expression* **)**

Parameters

- **string-expression** The string to be converted.

Returns

NVARCHAR

LONG NVARCHAR

Remarks

The UNISTR function allows the use of Unicode characters that cannot be represented in the CHAR character set used by the SQL statement. For example, in an English environment, the UNISTR function could be used to include Chinese characters.

The UNISTR function offers similar functionality to the N'' constant, however the UNISTR function allows Unicode characters and characters from the CHAR character set, whereas the N'' constant only allows characters from the CHAR character set.

The *string-expression* contains characters and Unicode escape sequences. The Unicode escape sequences are of the form \uXXXX or \uXXXXXX, where each X is a hexadecimal digit. The UNISTR function converts each character and each Unicode escape sequence to the corresponding Unicode character.

If a 6-digit Unicode escape sequence is used, its value must not exceed 10FFFF, the largest Unicode code point. A sequence such as \u234567 is not a 6-digit Unicode escape sequence. It is the 4-digit sequence \u2345 followed by the characters 6 and 7.

If two adjacent Unicode escape sequences form a UTF-16 surrogate pair, they are combined into one Unicode character in the output.

See also

- "CONNECTION_EXTENDED_PROPERTY function [String]" on page 156
- "NCHAR function [String]" on page 252
- "TO_CHAR function [String]" on page 323
- "TO_NCHAR function [String]" on page 324
- "UNICODE function [String]" on page 331
- "Strings" on page 9

Standards and compatibility

- **SQL/2003** Vendor extension.

Examples

The following example returns the string Hello.

```
SELECT UNISTR( 'Hel\u006c\u006F' );
```

The following example combines the UTF-16 surrogate pair D800-DF02 into the Unicode code point 10302.

```
SELECT UNISTR( '\uD800\uDF02' );
```

The example is equivalent to the previous one.

```
SELECT UNISTR( '\u010302' );
```

UPPER function [String]

Converts all characters in a string to uppercase. This function is identical the UCASE function.

Syntax

UPPER(*string-expression* **)**

Parameters

- **string-expression** The string to be converted to uppercase.

Returns

VARCHAR

NVARCHAR

LONG VARCHAR

LONG NVARCHAR

Remarks

The UCASE function is similar to the UPPER function.

See also

- "UCASE function [String]" on page 330
- "LCASE function [String]" on page 235
- "LOWER function [String]" on page 243
- "String functions" on page 131

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value CHOCOLATE.

```
SELECT UPPER( 'ChocoLate' );
```

USER_ID function [System]

Returns the numeric user ID for the specified user name.

Syntax

USER_ID([*user-name*] **)**

Parameters

- **user-name** The user name for the user ID you are searching for.

Returns

INT

Remarks

If you do not specify *user-name*, the ID of the current user is returned.

See also

- "USER_NAME function [System]" on page 334
- "SUSER_ID function [System]" on page 321

Standards and compatibility

- **SQL/2003** Transact-SQL extension.

Example

The following statement returns the GROUPO user ID.

```
SELECT USER_ID( 'GROUPO' );
```

USER_NAME function [System]

Returns the user name for the specified user ID.

Syntax

USER_NAME([*user-id*] **)**

Parameters

- **user-id** The user ID of the user you are searching for.

Returns

LONG VARCHAR

Remarks

If you do not specify *user-id*, the user name of the current user is returned.

See also

- "USER_ID function [System]" on page 333
- "SUSER_NAME function [System]" on page 321

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the user name for user ID 101.

```
SELECT USER_NAME( 101 );
```

UUIDTOSTR function [String]

Converts a unique identifier value (UUID, also known as GUID) to a string value.

Not needed in newer databases
In databases created before version 9.0.2, the UNIQUEIDENTIFIER data type was defined as a user-defined data type and the STRTOUUID and UUIDTOSTR functions were needed to convert between binary and string representations of UUID values.

In databases created using version 9.0.2 or later, the UNIQUEIDENTIFIER data type was changed to a native data type and SQL Anywhere carries out conversions as needed. You do not need to use STRTOUUID and UUIDTOSTR functions with these versions.

For more information, see "UNIQUEIDENTIFIER data type" on page 106.

Syntax

UUIDTOSTR(*uuid-expression* **)**

Parameters

- **uuid-expression** A unique identifier value.

Returns

VARCHAR

Remarks

Converts a unique identifier to a string value in the format *xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx*, where x is a hexadecimal digit. If the binary value is not a valid uniqueidentifier, NULL is returned.

This function is useful if you want to view a UUID value.

See also

- "NEWID function [Miscellaneous]" on page 252
- "STRTOUUID function [String]" on page 316
- "String functions" on page 131

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement creates a table mytab with two columns. Column pk has a unique identifier data type, and column c1 has an integer data type. It then inserts two rows with the values 1 and 2 respectively into column c1.

```
CREATE TABLE mytab(
    pk UNIQUEIDENTIFIER PRIMARY KEY DEFAULT NEWID(),
    c1 INT );
INSERT INTO mytab( c1 ) values ( 1 );
INSERT INTO mytab( c1 ) values ( 2 );
```

Executing the following SELECT statement returns all the data in the newly created table.

```
SELECT * FROM mytab;
```

You will see a two-column, two-row table. The value displayed for column pk will be binary values.

To convert the unique identifier values into a readable format, execute the following command:

```
SELECT UUIDTOSTR(pk), c1 FROM mytab;
```

The UUIDTOSTR function is not needed for databases created with version 9.0.2 or later.

VAR_POP function [Aggregate]

Computes the statistical variance of a population consisting of a numeric-expression, as a DOUBLE.

Syntax 1

VAR_POP(*numeric-expression* **)**

Syntax 2

VAR_POP(*numeric-expression* **) OVER (** *window-spec* **)**

window-spec : see Syntax 2 instructions in the Remarks section below

Parameters

- **numeric-expression** The expression whose population-based variance is calculated over a set of rows. The expression is commonly a column name.

Returns

DOUBLE

Remarks

This function converts its argument to DOUBLE, performs the computation in double-precision floating point, and returns a DOUBLE as the result.

The population-based variance (s^2) of *numeric-expression* (x) is computed according to the following formula:

$s^2 = (1/N) * \text{SUM}(x_I - \text{mean}(x))^2$

This variance does not include rows where *numeric-expression* is NULL. It returns NULL for a group containing no rows.

Syntax 2 represents usage as a window function in a SELECT statement. As such, elements of *window-spec* can be specified either in the function syntax (inline), or in conjunction with a WINDOW clause in the SELECT statement. See the *window-spec* definition provided in “WINDOW clause” on page 793.

For more information about using window functions in SELECT statements, including working examples, see “Window functions” [*SQL Anywhere Server - SQL Usage*].

See also

- "Aggregate functions" on page 124

Standards and compatibility

- **SQL/2003** SQL foundation feature (T611) outside core SQL.

Example

The following statement lists the average and variance in the number of items per order in different time periods:

```
SELECT YEAR( ShipDate ) AS Year,
    QUARTER( ShipDate ) AS Quarter,
    AVG( Quantity ) AS Average,
    VAR_POP( quantity ) AS Variance
FROM SalesOrderItems
GROUP BY Year, Quarter
ORDER BY Year, Quarter;
```

Year	Quarter	Average	Variance
2000	1	25.775148	203.9021...
2000	2	27.050847	225.8109...
...	...	...	...

VAR_SAMP function [Aggregate]

Computes the statistical variance of a sample consisting of a numeric-expression, as a DOUBLE.

Syntax 1

VAR_SAMP(*numeric-expression* **)**

Syntax 2

VAR_SAMP(*numeric-expression* **) OVER (** *window-spec* **)**

window-spec : see Syntax 2 instructions in the Remarks section below

Parameters

- **numeric-expression** The expression whose sample-based variance is calculated over a set of rows. The expression is commonly a column name.

Returns

DOUBLE

Remarks

This function converts its argument to DOUBLE, performs the computation in double-precision floating point, and returns a DOUBLE as the result.

The variance (s^2) of *numeric-expression* (x) is computed according to the following formula, which assumes a normal distribution:

$s^2 = (1/(N - 1)) * SUM(x_I - mean(x))^2$

This variance does not include rows where *numeric-expression* is NULL. It returns NULL for a group containing either 0 or 1 rows.

Syntax 2 represents usage as a window function in a SELECT statement. As such, elements of *window-spec* can be specified either in the function syntax (inline), or in conjunction with a WINDOW clause in the SELECT statement. See the *window-spec* definition provided in "WINDOW clause" on page 793.

For more information about using window functions in SELECT statements, including working examples, see "Window functions" [*SQL Anywhere Server - SQL Usage*].

See also

- "Aggregate functions" on page 124
- "VARIANCE function [Aggregate]" on page 338

Standards and compatibility

- **SQL/2003** SQL foundation feature outside core SQL. The VARIANCE syntax is a vendor extension.

Example

The following statement lists the average and variance in the number of items per order in different time periods:

```
SELECT YEAR( ShipDate ) AS Year,
    QUARTER( ShipDate ) AS Quarter,
    AVG( Quantity ) AS Average,
    VAR_SAMP( quantity ) AS Variance
FROM SalesOrderItems
GROUP BY Year, Quarter
ORDER BY Year, Quarter;
```

Year	Quarter	Average	Variance
2000	1	25.775148	205.1158...
2000	2	27.050847	227.0939...
...	...	...	...

VARIANCE function [Aggregate]

An alias for VAR_SAMP. See "VAR_SAMP function [Aggregate]" on page 337.

VAREXISTS function [Miscellaneous]

Returns 1 if a user-defined variable has been created or declared with a given name. Returns 0 if no such variable has been created.

Syntax

VAREXISTS(*variable-name-string* **)**

Parameters

- **variable-name-string** The variable name to be tested, as a string.

Returns

INT

See also

- "CREATE VARIABLE statement" on page 545
- "DECLARE statement" on page 551
- "IF statement" on page 641

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following IF statement creates a variable with a name start_time if one is not already created or declared. The variable can then be used safely.

```
IF VAREXISTS( 'start_time' ) = 0 THEN
    CREATE VARIABLE start_time TIMESTAMP;
END IF;
SET start_time = current timestamp;
```

WATCOMSQL function [Miscellaneous]

Takes a Transact-SQL statement and rewrites it in the Watcom-SQL dialect. This can be useful when converting existing Adaptive Server Enterprise stored procedures into Watcom SQL syntax.

Syntax

WATCOMSQL(*sql-statement-string* **)**

Parameters

- **sql-statement-string** The SQL statement that the function uses to determine its dialect.

Returns

LONG VARCHAR

See also

- “SQLDIALECT function [Miscellaneous]” on page 309
- “TRANSACTSQL function [Miscellaneous]” on page 327

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the string `'SELECT empl_name AS EmployeeName FROM Employees'`.

```
SELECT WATCOMSQL( 'SELECT EmployeeName=empl_name FROM Employees' ) FROM dummy;
```

WEEKS function [Date and time]

Returns the number of weeks between two dates.

Syntax 1

WEEKS([*datetime-expression*,] *datetime-expression* **)**

Syntax 2

WEEKS(*datetime-expression*, *integer-expression* **)**

Parameters

- **datetime-expression** A date and time.
- **integer-expression** The number of weeks to be added to the *datetime-expression*. If *integer-expression* is negative, the appropriate number of weeks is subtracted from the datetime value. If you supply an *integer-expression*, the *datetime-expression* must be explicitly cast as DATETIME.

Returns

Syntax 1 returns an INTEGER.

Syntax 2 returns a TIMESTAMP.

Remarks

Given a single date (Syntax 1), the WEEKS function returns the number of weeks since 0000-02-29.

Given two dates (Syntax 1), the WEEKS function returns the number of weeks between them. The WEEKS function is similar to the DATEDIFF function, however the method used to calculate the number of weeks between two dates is not the same and can return a different result. The return value for WEEKS is determined by dividing the number of days between the two dates by seven, and then rounding down; however, DATEDIFF uses number of week boundaries. This can cause the values returned to be different. For example, if the first date is a Friday and the second date is the following Monday, the WEEKS function returns a difference of 0, and the DATEDIFF function returns a difference of 1. While neither method is better than the other, you should consider the difference when choosing between WEEKS and DATEDIFF.

For more information about the DATEDIFF function, see "DATEDIFF function [Date and time]" on page 174.

Given a date and an integer (Syntax 2), the WEEKS function adds the integer number of weeks to the specified date. This function is similar to the DATEADD function.

For more information about the DATEADD function, see "DATEADD function [Date and time]" on page 173.

See also

For information about casting data types, see "CAST function [Data type conversion]" on page 148.

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value 8, signifying that 2008-09-13 10:07:12 is eight weeks after 2008-07-13 06:07:12.

```
SELECT WEEKS( '2008-07-13 06:07:12',
   '2008-09-13 10:07:12' );
```

The following statement returns the value 104792, signifying that the date is 104792 weeks after 0000-02-29.

```
SELECT WEEKS( '2008-07-13 06:07:12' );
```

The following statement returns the timestamp 2008-06-16 21:05:07.0, indicating the date and time five weeks after 2008-05-12 21:05:07.

```
SELECT WEEKS( CAST( '2008-05-12 21:05:07'
AS TIMESTAMP ), 5);
```

WRITE_CLIENT_FILE function [String]

Creates and writes to a file on the client computer.

Syntax

WRITE_CLIENT_FILE(*filename*, *blob-expression* [**, 'A'**] **)**

Parameters

- **filename** The name of the file on the client computer. The name is resolved on the client computer relative to the current working directory of the client application.
- **blob-expression** A binary string to be written to *filename* on the client computer.
- **A** By default, if the file already exists, it is overwritten. If you want the data to be appended to existing data, specify 'A'. If the file does not already exist, and you specify 'A', the file is still created.

Returns

INT

Remarks

The database server converts *filename* from the database character set to the client character set. On the client computer, *filename* is then converted to the operating system character set.

Since the data is a binary string, if you want the data to be in a particular character set, or compressed, or encrypted, you must perform these operations on the data before sending it to the WRITE_CLIENT_FILE function.

Reading of the file is performed by the client software library and the transfer of data is done using the command sequence communication protocol.

Permissions

When writing to a file on a client computer:

- WRITECLIENTFILE authority is required. See "WRITECLIENTFILE authority" [*SQL Anywhere Server - Database Administration*].
- The client application must have write permissions on the computer being written to.
- The allow_write_client_file database option must be enabled. See "allow_write_client_file option [database]" [*SQL Anywhere Server - Database Administration*].
- The write_client_file secured feature must be enabled. See "-sf server option" [*SQL Anywhere Server - Database Administration*].

See also

- "Accessing data on client computers" [*SQL Anywhere Server - SQL Usage*]
- "WRITECLIENTFILE authority" [*SQL Anywhere Server - Database Administration*]
- "UNLOAD statement" on page 772
- "CSCONVERT function [String]" on page 168
- "DECOMPRESS function [String]" on page 186
- "DECRYPT function [String]" on page 187

Standards and compatibility

- **SQL/2003** Vendor extension.

XMLAGG function [Aggregate]

Generates a forest of XML elements from a collection of XML values.

Syntax

XMLAGG(*value-expression* [**ORDER BY** *order-by-expression*] **)**

Parameters

- **value-expression** An XML value. The content is escaped unless the data type is XML. The *order-by-expression* orders the elements returned by the function.

- **order-by-expression** An expression used to order the XML elements according to the value of this expression.

Returns

XML

Remarks

Any values that are NULL are omitted from the result. If all inputs are NULL, or there are no rows, the result is NULL. If you require a well-formed XML document, you must ensure that your query is written so that the generated XML has a single root element.

Data in BINARY, LONG BINARY, IMAGE, and VARBINARY columns is automatically returned in base64-encoded format when you execute a query that contains XMLAGG.

For an example of a query that uses the XMLAGG function with an ORDER BY clause, see "Using the XMLAGG function" [*SQL Anywhere Server - SQL Usage*].

See also

- "Using the XMLAGG function" [*SQL Anywhere Server - SQL Usage*]

Standards and compatibility

- Part of the SQL/XML draft standard.

Example

The following statement generates an XML document that shows the orders placed by each customer.

```
SELECT XMLELEMENT( NAME "order",
                   XMLATTRIBUTES( ID AS order_id ),
                     ( SELECT XMLAGG(
                         XMLELEMENT(
                           NAME "Products",
                           XMLATTRIBUTES( ProductID, Quantity AS
"quantity_shipped" ) ) )
                       FROM SalesOrderItems soi
                       WHERE soi.ID = so.ID
                     )
                 ) AS products_ordered
FROM SalesOrders so
ORDER BY so.ID;
```

XMLCONCAT function [String]

Produces a forest of XML elements.

Syntax

XMLCONCAT(*xml-value* [**,** ...]**)**

Parameters

- **xml-value** The XML values to be concatenated.

Returns

XML

Remarks

Generates a forest of XML elements. In an unparsed XML document, a forest refers to the multiple root nodes within the document. NULL values are omitted from the result. If all the values are NULL, then NULL is returned. The XMLCONCAT function does not check whether the argument has a prolog. If you require a well-formed XML document, you must ensure that your query is written so that a single root element is generated.

Element content is always escaped unless the data type is XML. Data in BINARY, LONG BINARY, IMAGE, and VARBINARY columns is automatically returned in base64-encoded format when you execute a query that contains a XMLCONCAT function.

See also

- "Using the XMLCONCAT function" [*SQL Anywhere Server - SQL Usage*]
- "XMLELEMENT function [String]" on page 344
- "XMLFOREST function [String]" on page 346
- "String functions" on page 131

Standards and Compatibility

- Part of the SQL/XML draft standard.

Example

The following query generates <CustomerID>, <cust_fname>, and <cust_lname> elements for each customer.

```
SELECT XMLCONCAT( XMLELEMENT ( NAME CustomerID, ID ),
                  XMLELEMENT( NAME cust_fname, GivenName ),
                  XMLELEMENT( NAME cust_lname, Surname )
                 ) AS "Customer Information"
FROM Customers
WHERE ID < 120;
```

XMLELEMENT function [String]

Produces an XML element within a query.

Syntax

XMLELEMENT({ **NAME** *element-name-expression* | *string-expression* }
  [, **XMLATTRIBUTES (** *attribute-value-expression* [**AS** *attribute-name*],... **)**]
  [, *element-content-expression*,...]
)

Parameters

- **element-name-expression** An identifier. For each row, an XML element with the same name as the identifier is generated.

- **attribute-value-expression** An attribute of the element. This optional argument allows you to specify an attribute value for the generated element. This argument specifies the attribute name and content. If the *attribute-value-expression* is a column name, then the attribute name defaults to the column name. You can change the attribute name by specifying the *attribute-name argument*.
- **element-content-expression** The content of the element. This can be any string expression. You can specify an unlimited number of *element-content-expression* arguments and they are concatenated together. For example, the following SELECT statement returns the value <x>abcdef</x>:

  ```
  SELECT XMLELEMENT( NAME x, 'abc', 'def' );
  ```

Returns

XML

Remarks

NULL element values and NULL attribute values are omitted from the result. The letter case for both element and attribute names is taken from the query.

Element content is always escaped unless the data type is XML. Invalid element and attribute names are also quoted. For example, consider the following statement:

```
SELECT XMLELEMENT('H1', f_get_page_heading() );
```

If the function f_get_page_heading is defined as RETURNS LONG VARCHAR or RETURNS VARCHAR(1000), then the result is HTML encoded:

```
CREATE FUNCTION f_get_page_heading() RETURNS LONG VARCHAR
   BEGIN
      RETURN ('<B>My Heading</B>');
   END;
```

The above SELECT statement returns the following:

```
<H1>&lt;B&gt;My Heading&lt;/B&gt;</H1>
```

If the function is declared as RETURNS XML, then the above SELECT statement returns the following:

```
<H1><B>My Heading</B></H1>
```

For more information about quoting and the XMLELEMENT function, see "Invalid names and SQL/XML" [*SQL Anywhere Server - SQL Usage*].

XMLELEMENT functions can be nested to create a hierarchy. If you want to return different elements at the same level of the document hierarchy, use the XMLFOREST function.

For more information, see "XMLFOREST function [String]" on page 346.

Data in BINARY, LONG BINARY, IMAGE, and VARBINARY columns is automatically returned in base64-encoded format when you execute a query that contains the XMLELEMENT function.

See also

- “Using the XMLELEMENT function” [*SQL Anywhere Server - SQL Usage*]
- “XMLCONCAT function [String]” on page 343
- “XMLFOREST function [String]” on page 346
- “String functions” on page 131

Standards and compatibility

- Part of the SQL/XML draft standard.
- Omitting the NAME keyword and using a string expression as the first argument is a vendor extension.

Example

The following example produces an <item_name> element for each product in the result set, where the product name is the content of the element.

```
SELECT ID, XMLELEMENT( NAME item_name, p.Name )
FROM Products p
WHERE ID > 400;
```

The following example returns `<A HREF="http://www.ianywhere.com/" TARGET="_top">iAnywhere web site</A>`:

```
SELECT XMLELEMENT(
   'A',
   XMLATTRIBUTES( 'http://www.ianywhere.com/'
      AS "HREF", '_top' AS "TARGET"),
   'iAnywhere web site'
);
```

The following example returns `<table><tbody><tr align="center" valign="top"><td>Cell 1 info</td><td>Cell 2 info</td></tr></tbody></table>`:

```
SELECT XMLELEMENT( name "table",
         XMLELEMENT( name "tbody",
           XMLELEMENT( name "tr",
             XMLATTRIBUTES('center' AS "align", 'top' AS "valign"),
             XMLELEMENT( name "td", 'Cell 1 info' ),
             XMLELEMENT( name "td", 'Cell 2 info' )
           )
         )
       );
```

The following example returns `'<x>abcdef</x>','<custom_element>abcdef</custom_element>'`:

```
CREATE VARIABLE @my_element_name VARCHAR(200);
SET @my_element_name = 'custom_element';
SELECT XMLELEMENT( NAME x, 'abc', 'def' ),
   XMLELEMENT( @my_element_name,'abc', 'def' );
```

XMLFOREST function [String]

Generates a forest of XML elements.

Syntax

XMLFOREST(*element-content-expression* [**AS** *element-name*],... **)**

Parameters

- **element-content-expression** A string. An element is generated for each *element-content-expression* argument that is specified. The *element-content-expression* value becomes the content of the element. For example, if you specify the EmployeeID column from the Employees table for this argument, then an <EmployeeID> element containing an EmployeeID value is generated for each value in the table.

  Specify the *element-name* argument if you want to assign a name other than the *element-content-expression* to the element, otherwise the element name defaults to the *element-content-expression* name.

Returns

XML

Remarks

Produces a forest of XML elements. In the unparsed XML document, a forest refers to the multiple root nodes within the document. When all the arguments to the XMLFOREST function are NULL, a NULL value is returned. If only some values are NULL, the NULL values are omitted from the result. Element content is always quoted unless the data type is XML. You cannot specify attributes using the XMLFOREST function. Use the XMLELEMENT function if you want to specify attributes for generated elements.

For more information about the XMLELEMENT function, see "XMLELEMENT function [String]" on page 344.

Element names are escaped unless the data type is XML.

If you require a well-formed XML document, you must ensure that your query is written so that a single root element is generated.

Data in BINARY, LONG BINARY, IMAGE, and VARBINARY columns is automatically returned in base64-encoded format when you execute a query that contains XMLFOREST.

See also

- "Using the XMLFOREST function" [*SQL Anywhere Server - SQL Usage*]
- "XMLELEMENT function [String]" on page 344
- "XMLCONCAT function [String]" on page 343
- "String functions" on page 131

Standards and compatibility

- Part of the SQL/XML draft standard.

Example

The following statement produces an XML element for the first and last name of each employee.

```
SELECT EmployeeID,
       XMLFOREST( GivenName, Surname )
       AS "Employee Name"
FROM Employees;
```

XMLGEN function [String]

Generates an XML value based on an XQuery constructor.

Syntax

XMLGEN(*xquery-constructor*, *content-expression* [**AS** *variable-name*],... **)**

Parameters

- **xquery-constructor** An XQuery constructor. The XQuery constructor is an item defined in the XQuery language. It gives a syntax for constructing XML elements based on XQuery expressions. The *xquery-constructor* argument must be a well-formed XML document with one or more variable references. A variable reference is enclosed in curly braces and must be prefixed with a $ and have no surrounding white space. For example:

  ```
  SELECT XMLGEN( '<a>{$x}</a>', 1 AS x );
  ```

- **content-expression** A variable. You can specify multiple *content-expression* arguments. The optional *variable-name* argument is used to name the variable. For example,

  ```
  SELECT XMLGEN( '<emp EmployeeID="{$EmployeeID}"><StartDate>{$x}</
  StartDate></emp>',
                 EmployeeID, StartDate
                 AS x )
  FROM Employees;
  ```

Returns

XML

Remarks

Computed constructors as defined in the XQuery specification are not supported by the XMLGEN function.

When you execute a query that contains an XMLGEN function, data in BINARY, LONG BINARY, IMAGE, and VARBINARY columns is automatically returned in base64-encoded format.

Element content is always escaped unless the data type is XML. Illegal XML element and attribute names are also escaped.

For information about escaping and the XMLGEN function, see "Invalid names and SQL/XML" [*SQL Anywhere Server - SQL Usage*].

See also

- "Using the XMLGEN function" [*SQL Anywhere Server - SQL Usage*]
- "String functions" on page 131

Standards and compatibility

- Part of the SQL/XML draft standard.

Example

The following example generates <emp>, <Surname>, <GivenName>, and <StartDate> elements for each employee.

```
SELECT XMLGEN( '<emp EmployeeID="{$EmployeeID}">
                   <Surname>="{$Surname}"</Surname>
                   <GivenName>="{$GivenName}"</GivenName>
                   <StartDate>="{$StartDate}"</StartDate>
                </emp>',
                EmployeeID,
                Surname,
                GivenName,
                StartDate
              ) AS employee_list
FROM Employees;
```

YEAR function [Date and time]

Takes a timestamp value as a parameter and returns the year specified by that timestamp.

Syntax

YEAR(*datetime-expression* **)**

Parameters

- **datetime-expression** A date, time, or timestamp.

Returns

SMALLINT

Remarks

The value is returned as a SMALL INT.

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following example returns the value 2001.

```
SELECT YEAR( '2001-09-12' );
```

YEARS function [Date and time]

Given two dates, this function returns the integer number of years between them. It is recommended that you use the DATEDIFF function instead. See "DATEDIFF function [Date and time]" on page 174.

Given one date, it returns the year. It is recommended that you use the DATEPART function instead. See "DATEPART function [Date and time]" on page 177.

Given one date and an integer, it adds the integer number of years to the specified date. It is recommended that you use the DATEADD function instead. See "DATEADD function [Date and time]" on page 173.

Syntax 1

YEARS([*datetime-expression*,] *datetime-expression* **)**

Syntax 2

YEARS(*datetime-expression*, *integer-expression* **)**

Parameters

- **datetime-expression** A date and time.
- **integer-expression** The number of years to be added to the *datetime-expression*. If *integer-expression* is negative, the appropriate number of years is subtracted from the datetime value. If you supply an *integer-expression*, the *datetime-expression* must be explicitly cast as a datetime data type.

  For information about casting data types, see "CAST function [Data type conversion]" on page 148.

Returns

Syntax 1 returns an INTEGER. Syntax 2 returns a TIMESTAMP.

Remarks

The value of YEARS is calculated from the number of first days of the year between the two dates.

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statements both return -4.

```
SELECT YEARS( '1998-07-13 06:07:12',
              '1994-03-13 08:07:13' );

SELECT DATEDIFF( year,
   '1998-07-13 06:07:12',
   '1994-03-13 08:07:13' );
```

The following statements return 1998.

```
SELECT YEARS( '1998-07-13 06:07:12' )
SELECT DATEPART( year, '1998-07-13 06:07:12' );
```

The following statements return the given date advanced 300 years.

```
SELECT YEARS( CAST( '1998-07-13 06:07:12' AS TIMESTAMP ), 300 )

SELECT DATEADD( year, 300, '1998-07-13 06:07:12' );
```

YMD function [Date and time]

Returns a date value corresponding to the given year, month, and day of the month. Values are small integers from -32768 to 32767.

Syntax

YMD(
integer-expression1,
integer-expression2,
integer-expression3 **)**

Parameters

- **integer-expression1** The year.
- **integer-expression2** The number of the month. The year is adjusted if the month is outside the range 1-12.
- **integer-expression3** The day number. The day can be any integer; the date is adjusted.

Returns

DATE

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement returns the value 1998-06-12.

```
SELECT YMD( 1998, 06, 12 );
```

If the values are outside their normal range, the date is adjusted. For example, the following statement returns the value 2000-03-01.

```
SELECT YMD( 1999, 15, 1 );
```

SQL statements

Contents

Using the SQL statement reference

This section describes some conventions used in documenting the SQL statements.

Common elements in SQL syntax

This section lists language elements that are found in the syntax of many SQL statements.

For more information about the elements described here, see "Identifiers" on page 8, "SQL data types" on page 77, "Search conditions" on page 35, "Expressions" on page 16, or "Strings" on page 9.

- **column-name** An identifier that represents the name of a column. See "Identifiers" on page 8.
- **condition** An expression that evaluates to TRUE, FALSE, or UNKNOWN. See "Truth value search conditions" on page 54.
- **connection-name** A string representing the name of an active connection. See "SQL Anywhere database connections" [*SQL Anywhere Server - Database Administration*].
- **data-type** A storage data type. See "SQL data types" on page 77.
- **expression** An expression. A common example of an expression in syntax is a column name. See "Expressions" on page 16.
- **filename** A string containing a file name.
- **hostvar** A C language variable, declared as a host variable preceded by a colon. See "Using host variables" [*SQL Anywhere Server - Programming*].
- **indicator-variable** A second host variable of type **short int** immediately following a normal host variable. It must also be preceded by a colon. Indicator variables are used to pass NULL values to and from the database. See "Using host variables" [*SQL Anywhere Server - Programming*].
- **materialized-view-name** An identifier that represents the name of a materialized view. See "Working with materialized views" [*SQL Anywhere Server - SQL Usage*].
- **number** Any sequence of digits followed by an optional decimal part and preceded by an optional negative sign. Optionally, the number can be followed by an E and then an exponent. For example,

  ```
  42
  -4.038
  .001
  3.4e10
  1e-10
  ```

- **owner** An identifier representing the user ID who owns a database object. See "Permissions acquired through ownership of an object" [*SQL Anywhere Server - Database Administration*].
- **query-block** A query block is a simple query expression, or a query expression with an ORDER BY clause.
- **query-expression** A query expression can be a SELECT, UNION, INTERSECT, or EXCEPT block (that is, a statement that does not contain an ORDER BY, WITH, FOR, FOR XML, or OPTION clause), or any combination of such blocks.

- **role-name** An identifier representing the role name of a foreign key. In conceptual database modeling, a verb or phrase that describes a relationship from one point of view. You can describe each relationship with two roles. Examples of roles are "contains" and "is a member of."
- **savepoint-name** An identifier that represents the name of a savepoint. See "Savepoints within transactions" [*SQL Anywhere Server - SQL Usage*].
- **search-condition** A condition that evaluates to TRUE, FALSE, or UNKNOWN. See "Search conditions" on page 35.
- **special-value** One of the special values described in "Special values" on page 58.
- **statement-label** An identifier that represents the label of a loop or compound statement. See "Control statements" [*SQL Anywhere Server - SQL Usage*].
- **statement-list** A list of SQL statements, each ending with a semicolon.
- **string-expression** An expression that resolves to a string. See "Expressions" on page 16.
- **table-list** A list of table names, which may include correlation names. See "FROM clause" on page 613 and "Key joins" [*SQL Anywhere Server - SQL Usage*].
- **table-name** An identifier that represents the name of a table. See "Identifiers" on page 8.
- **userid** An identifier representing a user name. See "Identifiers" on page 8.
- **variable-name** An identifier that represents a variable name. See "Variables" on page 66.
- **window-name** An identifier that represents a window name. Used in syntax related to window definition (for example, the WINDOW clause, and window functions such as RANK). See "Identifiers" on page 8.

Syntax conventions

The following conventions are used in the SQL syntax descriptions:

- **Keywords** All SQL keywords appear in uppercase, like the SQL statement ALTER TABLE in the following example:

  ALTER TABLE [*owner*.]*table-name*

- **Placeholders** Items that must be replaced with appropriate identifiers or expressions appear in italics, like the words *owner* and *table-name* in the following example:

  ALTER TABLE [*owner*.]*table-name*

- **Optional portions** Optional portions of a statement are enclosed by square brackets. For example:

  RELEASE SAVEPOINT [*savepoint-name*]

  These square brackets indicate that the *savepoint-name* is optional. The square brackets should not be typed.

  You might also see square brackets around a portions of keywords. For example, the following syntax indicates that you can use either COMMIT TRAN or COMMIT TRANSACTION:

COMMIT TRAN[**SACTION**] ...

Likewise, the following syntax indicates that you can use either COMMIT or COMMIT WORK:

COMMIT [**WORK**]

- **Repeating items** An item that can be repeated is followed by the appropriate list separator and an ellipsis (three dots), like *column-constraint* in the following example:

  ADD *column-definition* [*column-constraint*, ...]

  In this case, you can specify no column constraint, one, or more. If more than one is specified, they must be separated by commas.

- **Options** When none or only one of a list of items can be chosen, vertical bars separate the items and the list is enclosed in square brackets.

  [**ASC** | **DESC**]

  For example, you can choose one of ASC, DESC, or neither. The square brackets should not be typed.

- **Alternatives** When precisely one of the options must be chosen, the alternatives are enclosed in curly braces.

  [**QUOTES** { **ON** | **OFF** }]

  In this case, if the QUOTES option is chosen, one of ON or OFF must be provided. The brackets and braces should not be typed.

Statement applicability indicators

Some statement titles are followed by an indicator in square brackets that indicate where the statement can be used. These indicators are as follows:

- **[ESQL]** The statement is for use in embedded SQL.
- **[Interactive SQL]** The statement can be used only in Interactive SQL.
- **[SP]** The statement is for use in stored procedures, triggers, or batches.
- **[T-SQL]** The statement is implemented for compatibility with Adaptive Server Enterprise. In some cases, the statement cannot be used in stored procedures that are not in Transact-SQL format. In other cases, an alternative statement closer to the SQL/2003 standard is recommended unless Transact-SQL compatibility is an issue.
- **[external procedures]** The statement is for use in calling external functions and procedures.
- **[MobiLink]** The statement is for use only in MobiLink clients.
- **[SQL Remote]** The statement can be used only in SQL Remote.
- **[web services]** The statement is for use in web services clients.

If two sets of brackets are used, the statement can be used in both environments. For example, [ESQL][SP] means a statement can be used in both embedded SQL and stored procedures.

SQL statements (A-D)

The following sections define the syntax information for SQL statements A-D.

See also

- "SQL statements (E-O)" on page 595
- "SQL statements (P-Z)" on page 692
- "Common elements in SQL syntax" on page 354
- "Syntax conventions" on page 355
- "Statement applicability indicators" on page 356

ALLOCATE DESCRIPTOR statement [ESQL]

Use this statement to allocate space for a SQL descriptor area (SQLDA).

Syntax

ALLOCATE DESCRIPTOR *descriptor-name*
[**WITH MAX** { *integer* | *hostvar* }]

descriptor-name : *identifier*

Parameters

- **WITH MAX clause** Allows you to specify the number of variables within the descriptor area. The default size is one. You must still call fill_sqlda to allocate space for the actual data items before doing a fetch or any statement that accesses the data within a descriptor area.

Remarks

Allocates space for a descriptor area (SQLDA). You must declare the following in your C code prior to using this statement:

```
struct sqlda * descriptor_name
```

Permissions

None.

Side effects

None.

See also

- "DEALLOCATE DESCRIPTOR statement [ESQL]" on page 550
- "The SQL descriptor area (SQLDA)" [*SQL Anywhere Server - Programming*]

Standards and compatibility

- **SQL/2003** Core feature.

Example

The following sample program includes an example of ALLOCATE DESCRIPTOR statement usage.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
EXEC SQL INCLUDE SQLCA;
#include "sqldef.h"
EXEC SQL BEGIN DECLARE SECTION;
int         x;
short       type;
int         numcols;
char        string[100];
a_SQL_statement_number  stmt = 0;
EXEC SQL END DECLARE SECTION;
int main(int argc, char * argv[]){
   struct sqlda *      sqlda1;
   if( !db_init( &sqlca ) ) {
      return 1;
   }
   db_string_connect( &sqlca,
   "UID=dba;PWD=sql;DBF=d:\\DB Files\\sample.db");
   EXEC SQL ALLOCATE DESCRIPTOR sqlda1 WITH MAX 25;
   EXEC SQL PREPARE :stmt FROM
      'SELECT * FROM Employees';
   EXEC SQL DECLARE curs CURSOR FOR :stmt;
   EXEC SQL OPEN curs;
   EXEC SQL DESCRIBE :stmt into sqlda1;
   EXEC SQL GET DESCRIPTOR sqlda1 :numcols=COUNT;
   // how many columns?
   if( numcols > 25 ) {
      // reallocate if necessary
      EXEC SQL DEALLOCATE DESCRIPTOR sqlda1;
      EXEC SQL ALLOCATE DESCRIPTOR sqlda1
         WITH MAX :numcols;
      EXEC SQL DESCRIBE :stmt into sqlda1;
   }
   type = DT_STRING; // change the type to string
   EXEC SQL SET DESCRIPTOR sqlda1 VALUE 2 TYPE = :type;
   fill_sqlda( sqlda1 );
   // allocate space for the variables
   EXEC SQL FETCH ABSOLUTE 1 curs
      USING DESCRIPTOR sqlda1;
   EXEC SQL GET DESCRIPTOR sqlda1
      VALUE 2 :string = DATA;
   printf("name = %s", string );
   EXEC SQL DEALLOCATE DESCRIPTOR sqlda1;
   EXEC SQL CLOSE curs;
   EXEC SQL DROP STATEMENT :stmt;
   db_string_disconnect( &sqlca, "" );
   db_fini( &sqlca );
   return 0;
}
```

ALTER DATABASE statement

Use this statement to upgrade the database, turn jConnect support for a database on or off, calibrate the database, change the transaction log and transaction log mirror file names, or force a mirror server to take ownership of a database.

An error message is returned if you execute an ALTER DATABASE UPGRADE statement on a database server that is currently being mirrored.

Syntax 1 - Upgrading components or restoring objects

ALTER DATABASE UPGRADE
[**PROCEDURE ON**]
[**JCONNECT** { **ON** | **OFF** }]

Syntax 2 - Performing calibration

ALTER DATABASE {
 CALIBRATE [**SERVER**]
 | **CALIBRATE DBSPACE** *dbspace-name*
 | **CALIBRATE DBSPACE TEMPORARY**
 | **CALIBRATE GROUP READ**
 | **CALIBRATE PARALLEL READ**
 | **RESTORE DEFAULT CALIBRATION**
}

Syntax 3 - Changing transaction log and transaction log mirror names

ALTER DATABASE *dbfile*
ALTER [**TRANSACTION**] **LOG** {
{ **ON** [*log-name*] [**MIRROR** *mirror-name*] | **OFF** }
[**KEY** *key*]

Syntax 4 - Changing ownership of a database

ALTER DATABASE
{ *dbname* **FORCE START**
| **SET PARTNER FAILOVER** }

Parameters

- **PROCEDURE clause** Drop and re-create all dbo- and sys-owned procedures in the database.
- **JCONNECT clause** To allow the jConnect JDBC driver access to system catalog information, specify JCONNECT ON. This installs the system objects that provide jConnect support. Specify JCONNECT OFF if you want to exclude the jConnect system objects. You can still use JDBC, as long as you do not access system information. JCONNECT is ON by default.

  If you are altering a database for use on Windows Mobile, see "Using jConnect on Windows Mobile" [*SQL Anywhere Server - Database Administration*].
- **CALIBRATE [SERVER] clause** Calibrate all dbspaces except for the temporary dbspace. This clause also performs the work done by CALIBRATE PARALLEL READ.
- **CALIBRATE DBSPACE clause** Calibrate the specified dbspace.
- **CALIBRATE DBSPACE TEMPORARY clause** Calibrate the temporary dbspace.
- **CALIBRATE GROUP READ clause** Perform group read calibration on the temporary dbspace. Writes large work tables to the temporary dbspace and uses different group read sizes to time the reading of the files. If adding space to the temporary table exceeds the limit for the connection, or if the cache is not large enough to allow calibration with the largest memory size, calibration fails and an error message is returned.

- **CALIBRATE PARALLEL READ clause** Calibrate the parallel I/O capabilities of devices for all dbspace files. The CALIBRATE [SERVER] clause also performs this calibration.
- **RESTORE DEFAULT CALIBRATION clause** Restore the Disk Transfer Time (DTT) model to the built-in default values that are based on typical hardware and configuration settings.
- **ALTER [TRANSACTION] LOG clause** Change the file name of the transaction log or transaction log mirror file. If MIRROR *mirror-name* is not specified, the clause sets a file name for a new transaction log. If the database is not currently using a transaction log, it starts using one. If the database is already using a transaction log, it changes to using the new file as its transaction log.

  If MIRROR *mirror-name* is specified, the clause sets a file name for a new transaction log mirror. If the database is not currently using a transaction log mirror, it starts using one. If the database is already using a transaction log mirror, it changes to using the new file as its transaction log mirror.

  You can also use this clause to turn off the transaction or transaction log mirror. For example, ALTER DATABASE LOG OFF.
- **KEY clause** Specifies the encryption key to use for the transaction log or transaction log mirror. When using the ALTER [TRANSACTION] clause on a strongly encrypted database, you must specify the encryption key.
- **dbname FORCE START clause** Forces a database server that is currently acting as the mirror server to take ownership of the database. This clause can be executed from within a procedure or event, and must be executed while connected to the utility database on the mirror server. See "Forcing a database server to become the primary server" [*SQL Anywhere Server - Database Administration*].
- **SET PARTNER FAILOVER clause** Initiate a database mirroring failover from the primary server to the mirror server. This statement must be executed while connected to the database on the primary server, and can be executed from within a procedure or event. When executed, any existing connections to the database are closed, including the connection that executed the statement. If the statement is contained in a procedure or event, other statements that follow it may not be executed. See "Initiating failover on the primary server" [*SQL Anywhere Server - Database Administration*].

Remarks

Syntax 1 You can use the ALTER DATABASE UPGRADE statement as an alternative to the Upgrade utility to upgrade or update a database. This applies to maintenance releases as well. After running this statement, you should restart the database. In general, changes in databases between minor versions are limited to additional database options and minor system table and procedure changes. The ALTER DATABASE UPGRADE statement upgrades the system tables to the current version and adds any new database options. If necessary, it also drops and recreates all system procedures. You can force a rebuild of the system procedures by specifying the PROCEDURE ON clause.

You can also use the ALTER DATABASE UPGRADE statement to restore settings and system objects to their original installed state.

Features that require a physical reorganization of the database file are not made available by executing an ALTER DATABASE UPGRADE statement. Such features include index enhancements and changes in data storage. To obtain the benefits of these enhancements, you must unload and reload your database. See "Rebuilding databases" [*SQL Anywhere Server - SQL Usage*].

Caution
You should always back up your database files before upgrading. If you apply the upgrade to the existing files, then these files become unusable if the upgrade fails. For information about backing up your database, see "Backup and data recovery" [*SQL Anywhere Server - Database Administration*].

To use the jConnect JDBC driver to access system catalog information, specify JCONNECT ON (the default). If you want to exclude the jConnect system objects, specify JCONNECT OFF. Setting JCONNECT OFF does not remove jConnect support from a database. Also, you can still use JDBC, as long as you do not access system catalog information. If you subsequently download a more recent version of jConnect, you can upgrade the version in the database by (re)executing the ALTER DATABASE UPGRADE JCONNECT ON statement. See "Installing jConnect system objects into a database" [*SQL Anywhere Server - Programming*].

Syntax 2 Use Syntax 2 to perform recalibration of the I/O cost model used by the optimizer. This updates the Disk Transfer Time (DTT) model, which is a mathematical model of the disk I/O used by the cost model. When you recalibrate the I/O cost model, the database server is unavailable for other use. In addition, it is essential that all other activities on the computer are idle. Recalibrating the database server is an expensive operation and may take some time to complete. It is recommended that you leave the default in place.

When using the CALIBRATE PARALLEL READ clause, parallel calibration is not performed on dbspace files with fewer than 10000 pages. Even though the database server automatically suspends all of its activity during calibration operations, parallel calibration should be done when there are no processes consuming significant resources on the same computer. After calibration, you can retrieve the maximum estimated number of parallel I/O operations allowed on a dbspace file using the IOParallelism extended database property. See "DB_EXTENDED_PROPERTY function [System]" on page 181.

To eliminate repetitive, time-consuming recalibration activities when there is a large number of similar hardware installations, you can re-use a calibration by unloading it and then applying it (loading it) into another database using the sa_unload_cost_model and sa_load_cost_model system procedures, respectively. See "sa_unload_cost_model system procedure" on page 962, and "sa_load_cost_model system procedure" on page 892.

Syntax 3 You can use the ALTER DATABASE statement to change the transaction log and transaction log mirror names associated with a database file. These changes are the same as those made by the Transaction Log (dblog) utility. You can execute this statement while connected to the utility database or another database, depending on the setting of the -gu option. If you are changing the transaction log or transaction log mirror of an encrypted database, you must specify a key. You cannot stop using the transaction log if the database is using auditing. Once you turn off auditing, you can stop using the transaction log. This syntax is not supported in procedures, triggers, events, or batches.

Syntax 4 ALTER DATABASE ... FORCE START must be run from the mirror server, not the primary server. Attempting to execute an ALTER DATABASE ... FORCE START statement for a database that is not being mirrored or is currently active and owned by this server results in an error. Also, if the primary server is still connected to the mirror server, an error is given. See "Introduction to database mirroring" [*SQL Anywhere Server - Database Administration*].

Permissions

For Syntax 1 and 2, must have DBA authority, and must be the only connection to the database. ALTER DATABASE UPGRADE is not supported on Windows Mobile.

For Syntax 3, you must have file permissions on the directories where the transaction log is located, and the database must not be running.

For Syntax 4, you must have the permissions specified by the -gk server option.

Side effects

Automatic commit

See also

- "CREATE DATABASE statement" on page 432
- "Upgrade utility (dbupgrad)" [*SQL Anywhere Server - Database Administration*]
- "CREATE STATISTICS statement" on page 516
- "Transaction Log utility (dblog)" [*SQL Anywhere Server - Database Administration*]
- "DB_EXTENDED_PROPERTY function [System]" on page 181
- "-gu server option" [*SQL Anywhere Server - Database Administration*]
- "sa_unload_cost_model system procedure" on page 962
- "sa_load_cost_model system procedure" on page 892

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following example disables jConnect support:

```
ALTER DATABASE UPGRADE JCONNECT OFF;
```

The following example sets the transaction log file name associated with *demo.db* to *newdemo.log*:

```
ALTER DATABASE 'demo.db'
   ALTER LOG ON 'newdemo.log';
```

ALTER DBSPACE statement

Use this statement to pre-allocate space for a dbspace or for the transaction log, or to update the catalog when a dbspace file is renamed or moved.

Syntax

ALTER DBSPACE { *dbspace-name* | **TRANSLOG** | **TEMPORARY** }
{ **ADD** *number* [*add-unit*]
| **RENAME** *filename* }

add-unit :
PAGES
| **KB**
| **MB**
| **GB**
| **TB**

Parameters

- **TRANSLOG clause** You supply the special dbspace name TRANSLOG to pre-allocate disk space for the transaction log. Pre-allocation improves performance if the transaction log is expected to grow quickly. You may want to use this feature if, for example, you are handling many binary large objects (BLOBs) such as bitmaps.
- **TEMPORARY clause** You supply the special dbspace name TEMPORARY to add space to temporary dbspaces. When space is added to a temporary dbspace, the additional space materializes in the corresponding temporary file immediately. Pre-allocating space to the temporary dbspace of a database can improve performance during execution complex queries that use large work tables.
- **ADD clause** An ALTER DBSPACE with the ADD clause pre-allocates disk space for a dbspace. It extends the corresponding database file by the specified size, in units of pages, kilobytes (KB), megabytes (MB), gigabytes (GB), or terabytes (TB). If you do not specify a unit, PAGES is the default. The page size of a database is fixed when the database is created.

  If space is not pre-allocated, database files are extended by about 256 KB at a time for page sizes of 2 KB, 4 KB, and 8 KB, and by about 32 pages for other page sizes, when the space is needed. Pre-allocating space can improve performance for loading large amounts of data and also serves to keep the database files more contiguous within the file system.

  You can use this clause to add space to any of the pre-defined dbspaces (system, temporary, temp, translog, and translogmirror). See "Pre-defined dbspaces" [*SQL Anywhere Server - Database Administration*].
- **RENAME clause** If you rename or move a database file other than the main file to a different directory or device, you can use ALTER DBSPACE with the RENAME clause to ensure that SQL Anywhere finds the new file when the database is started. The *filename* parameter can be a string literal, or a variable.

  The name change takes effect as follows:

  - If the dbspace was already open before the statement was executed (that is, you have not yet renamed the actual file), it remains accessible; however, the name stored in the catalog is updated. After the database is stopped, you must rename the file to match what you provided using the RENAME clause, otherwise the file name won't match the dbspace name in the catalog and the database server is unable to open the dbspace the next time the database is started.
  - If the dbspace was not open when the statement was executed, the database server attempts to open it after updating the catalog. If the dbspace can be opened, it becomes accessible. No error is returned if the dbspace cannot be opened.

    To determine if a dbspace is open, execute the statement below. If the result is NULL, the dbspace is not open.

    ```
    SELECT DB_EXTENDED_PROPERTY('FileSize','dbspace-name');
    ```

    Using ALTER DBSPACE with RENAME on the main dbspace, system, has no effect.

Remarks

Each database is held in one or more files. A dbspace is an additional file with a logical name associated with each database file, and used to hold more data than can be held in the main database file alone. ALTER DBSPACE modifies the main dbspace (also called the root file) or an additional dbspace. The dbspace names for a database are held in the ISYSFILE system table. The main database file has a dbspace name of system.

When a multi-file database is started, the start line or ODBC data source description tells SQL Anywhere where to find the main database file. The main database file holds the system tables. SQL Anywhere looks in these system tables to find the location of the other dbspaces, and then opens each of the other dbspaces. You can specify which dbspace new tables are created in by setting the default_dbspace option.

Permissions

Must have DBA authority. Must be the only connection to the database.

Side effects

Automatic commit.

See also

- "CREATE DBSPACE statement" on page 439
- "default_dbspace option [database]" [*SQL Anywhere Server - Database Administration*]
- "Working with database files" [*SQL Anywhere Server - Database Administration*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following example increases the size of the system dbspace by 200 pages:

```
ALTER DBSPACE system
ADD 200;
```

The following example increases the size of the system dbspace by 400 MB:

```
ALTER DBSPACE system
ADD 400 MB;
```

The following example changes the file name associated with the system_2 dbspace:

```
ALTER DBSPACE system_2
RENAME 'e:\db\dbspace2.db';
```

ALTER DOMAIN statement

Use this statement to rename a user-defined domain or data type.

Syntax

ALTER { **DOMAIN** | **DATATYPE** } *user-type*
RENAME *new-name*

Remarks

When you execute this statement, the name of the user-defined domain or data type is updated in the ISYSUSERTYPE system table.

Note
Any procedures, triggers, views, or events that refer to the user-defined domain or data type must be recreated, or else they will continue to refer to the old name.

Permissions

Must have DBA authority or be the database user who created the domain.

Side effects

Automatic commit.

See also

- "ISYSFILE system table" on page 803
- "CREATE DOMAIN statement" on page 443
- "Domains" on page 109
- "Using domains" [*SQL Anywhere Server - SQL Usage*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following example renames the Address domain to MailingAddress:

```
ALTER DOMAIN Address RENAME MailingAddress;
```

ALTER EVENT statement

Use this statement to change the definition of an event or its associated handler for automating predefined actions, or to alter the definition of scheduled actions. You can also use this statement to hide the definition of an event handler.

Syntax 1 - Altering an event

ALTER EVENT [*owner*.]*event-name*
[**AT** { **CONSOLIDATED** | **REMOTE** | **ALL** }]
[{ **DELETE TYPE**
  | **TYPE** *event-type*
  | **WHERE** { *trigger-condition* | **NULL** }
  | { **ADD** | **ALTER** | **DELETE** } **SCHEDULE** *schedule-spec* }]
[**ENABLE** | **DISABLE**]
[[**ALTER**] **HANDLER** *compound-statement* | **DELETE HANDLER**]

event-type :
  BackupEnd
| **Connect**
| **ConnectFailed**
| **DatabaseStart**
| **DBDiskSpace**
| **Deadlock**

| **"Disconnect"**
| **GlobalAutoincrement**
| **GrowDB**
| **GrowLog**
| **GrowTemp**
| **LogDiskSpace**
| **RAISERROR**
| **ServerIdle**
| **TempDiskSpace**

trigger-condition :
event_condition(*condition-name* **)** { **=** | **<** | **>** | **!=** | **<=** | **>=** } *value*

schedule-spec :
[*schedule-name*]
  { **START TIME** *start-time* | **BETWEEN** *start-time* **AND** *end-time* }
  [**EVERY** *period* { **HOURS** | **MINUTES** | **SECONDS** }]
  [**ON** { **(** *day-of-week*, ... **)** | **(** *day-of-month*, ... **)** }]
  [**START DATE** *start-date*]

event-name | *schedule-name* : *identifier*

day-of-week : *string*

value | *period* | *day-of-month* : *integer*

start-time | *end-time* : *time*

start-date : *date*

Syntax 2 - Hiding the definition of an event handler

ALTER EVENT *event-name* **SET HIDDEN**

Parameters

- **AT clause** Use this clause to change the specification regarding the databases at which the event is handled.
- **DELETE TYPE clause** Use this clause to remove an association of the event with an event type. For a description of event types, see "Understanding system events" [*SQL Anywhere Server - Database Administration*].
- **ADD | ALTER | DELETE SCHEDULE clause** Use this clause to change the definition of a schedule. Only one schedule can be altered in any one ALTER EVENT statement.
- **WHERE clause** Use this clause to change the trigger condition under which an event is fired. The WHERE NULL option deletes a condition. For descriptions of most of the parameters, see "CREATE EVENT statement" on page 450.
- **START TIME clause** Use this clause to specify the start time and, optionally, the end time, for the event. The *start-time* and *end-time* parameters are strings (for example, `'12:34:56'`). Variables and expressions are not allowed (for example, `NOW()`).
- **START DATE clause** Use this clause to specify the start date for the event. The *start-date* parameter is a string. Variables and expressions are not allowed (for example, `TODAY()`).

- **SET HIDDEN clause** Use this clause to hide the definition of an event handler. Specifying the SET HIDDEN clause results in the permanent obfuscation of the event handler definition stored in the action column of the ISYSEVENT system table.

Remarks

This statement allows you to alter an event definition created with CREATE EVENT. Possible uses include the following:

- You may want to hide the definition of an event handler.
- You may want to define and test an event handler without a trigger condition or schedule during a development phase, and then add the conditions for execution using ALTER EVENT once the event handler is completed.

If you need to alter an event, you can disable it while it is running by executing an ALTER EVENT ... DISABLE statement. To disable an event in Sybase Central, right-click the event and clear the **Enabled** option. Disabling the event does not interrupt current event handler execution; the event handler continues to execute until completion. When the event handler completes, it is not restarted until you re-enable it. You can alter and then re-enable the definition. To determine what events are running, execute the following statement:

```
SELECT *
FROM dbo.sa_conn_info()
WHERE CONNECTION_PROPERTY('EventName',Number ) = 'event-name'
```

Permissions

Must have DBA authority.

Side effects

Automatic commit.

See also

- "Understanding system events" [*SQL Anywhere Server - Database Administration*]
- "SYSEVENT system view" on page 998
- "BEGIN statement" on page 413
- "CREATE EVENT statement" on page 450

Standards and compatibility

- **SQL/2003** Vendor extension.

ALTER EXTERNAL ENVIRONMENT statement

Use this statement to specify the location of an external environment such as JAVA, PHP, or Perl, or to specify a user with DBA authority to be used by the external environment for connecting to the database.

Syntax

ALTER EXTERNAL ENVIRONMENT *environment-name*
  [**USER** *user-name*]
  [**LOCATION** *location-string*]

environment-name :
JAVA
| **PERL**
| **PHP**
| **CLR**
| **C_ESQL32**
| **C_ESQL64**
| **C_ODBC32**
| **C_ODBC64**

Parameters

- **environment-name** Use *environment-name* to specify the external environment you are altering.
- **USER clause** Use the USER clause to specify the name of a user in the database that has DBA authority. When the external environment is initially launched, it must connect to the database. By default, this connection is made using the DBA user ID, but if the database administrator prefers to have the external environment use a different user ID with DBA authority, use *user-name* to indicate the user.
- **LOCATION clause** Use the LOCATION clause to specify the location on the database server computer where the executable/binary for the external environment can be found. It includes the executable/binary name. This path can either be fully qualified or relative. If the path is relative, then the executable/binary must be in a location where the server can find it.

Remarks

For more information about how to work with external environments, see "Overview of external environments" [*SQL Anywhere Server - Programming*].

Permissions

Must have DBA authority.

Side effects

None

See also

- "Overview of external environments" [*SQL Anywhere Server - Programming*]
- "START EXTERNAL ENVIRONMENT statement" on page 754
- "STOP EXTERNAL ENVIRONMENT statement" on page 761
- "INSTALL EXTERNAL OBJECT statement" on page 654
- "REMOVE EXTERNAL OBJECT statement" on page 710
- "SYSEXTERNENV system view" on page 1000
- "SYSEXTERNENVOBJECT system view" on page 1000

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following example identifies a user with DBA authority that is used by the external environment to connect to the database server and the location of the Perl executable.

```
ALTER EXTERNAL ENVIRONMENT PERL
USER DBADMIN
LOCATION 'c:\\Perl64\\bin\\perl.exe';
```

ALTER FUNCTION statement

Use this statement to modify a function. You must include the entire new function in the ALTER FUNCTION statement.

Syntax 1

ALTER FUNCTION [*owner*.]*function-name function-definition*

function-definition : CREATE FUNCTION syntax

Syntax 2

ALTER FUNCTION [*owner*.]*function-name*
SET HIDDEN

Syntax 3

ALTER FUNCTION [*owner*.]*function-name*
RECOMPILE

Remarks

Syntax 1 The ALTER FUNCTION statement is identical in syntax to the CREATE FUNCTION statement except for the first word. Either version of the CREATE FUNCTION statement can be altered.

Existing permissions on the function are maintained, and do not have to be reassigned. If a DROP FUNCTION and CREATE FUNCTION were carried out, execute permissions would have to be reassigned.

Syntax 2 Use SET HIDDEN to obfuscates the definition of the associated function and cause it to become unreadable. The function can be unloaded and reloaded into other databases.

This setting is irreversible. If you will need the original source again, you must maintain it outside the database.

If SET HIDDEN is used, debugging using the debugger will not show the function definition, nor will it be available through procedure profiling.

Syntax 3 Use the RECOMPILE syntax to recompile a user-defined SQL function. When you recompile a function, the definition stored in the catalog is re-parsed and the syntax is verified. The preserved source for a function is not changed by recompiling. When you recompile a function, the definitions obfuscated by the SET HIDDEN clause remain obfuscated and unreadable.

Permissions

Must be the owner of the function or have DBA authority.

Side effects

Automatic commit.

See also

- "CREATE FUNCTION statement (web services)" on page 467
- "ALTER PROCEDURE statement" on page 376
- "DROP FUNCTION statement" on page 575
- "Hiding the contents of procedures, functions, triggers and views" [*SQL Anywhere Server - SQL Usage*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

In this example, MyFunction is created and altered. The SET HIDDEN clause obfuscates the function definition and makes it unreadable.

```
CREATE FUNCTION MyFunction(
   firstname CHAR(30),
   lastname CHAR(30) )
RETURNS CHAR(61)
BEGIN
   DECLARE name CHAR(61);
   SET name = firstname || ' ' || lastname;
   RETURN (name);
ALTER FUNCTION MyFunction SET HIDDEN;
END;
```

ALTER INDEX statement

Use this statement to rename an index, primary key, or foreign key, or to change the clustered nature of an index.

Syntax

ALTER { **INDEX** *index-name*
| [**INDEX**] **FOREIGN KEY** *role-name*
| [**INDEX**] **PRIMARY KEY** }
ON [*owner*.]*object-name* { **REBUILD** | *rename-clause* | *cluster-clause* } }

object-name : *table-name* | *materialized-view-name*

rename-clause : **RENAME** { **AS** | **TO** } *new-index-name*

cluster-clause : **CLUSTERED** | **NONCLUSTERED**

Parameters

- **rename-clause** Specify the new name for the index, primary key, or foreign key.
- **cluster-clause** Specify whether the index should be changed to CLUSTERED or NONCLUSTERED. Only one index on a table can be clustered.
- **REBUILD clause** Use this clause to rebuild an index, instead of dropping and recreating it.

Remarks

The ALTER INDEX statement carries out two tasks:

- It can be used to rename an index, primary key, or foreign key.
- It can be used to change an index type from nonclustered to clustered, or vice versa.

  The ALTER INDEX statement can be used to change the clustering specification of the index, but does not reorganize the data. As well, only one index per table or materialized view can be clustered.

ALTER INDEX cannot be used to change an index on a local temporary table. An attempt to do so will result in an Index not found error.

This statement cannot be executed when there are cursors opened with the WITH HOLD clause that use either statement or transaction snapshots. See "Snapshot isolation" [*SQL Anywhere Server - SQL Usage*].

Permissions

Must own the table, or have REFERENCES permissions on the table or materialized view, or have DBA authority.

Side effects

Automatic commit. Clears the Results tab in the Results pane in Interactive SQL. Closes all cursors for the current connection.

See also

- "CREATE INDEX statement" on page 470

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement renames the index IX_product_name on the Products table to ixProductName:

```
ALTER INDEX IX_product_name ON Products
RENAME TO ixProductName;
```

The following statement changes IX_product_name to be a clustered index:

```
ALTER INDEX IX_product_name ON Products
CLUSTERED;
```

ALTER LOGIN POLICY statement

Use this statement to alter an existing login policy.

Syntax

ALTER LOGIN POLICY *policy-name policy-options*

policy options :
policy-option [*policy-option* ...]

policy-option :
policy-option-name **=** *policy-option-value*

policy-option-value :
{ **UNLIMITED**
| **root**
| *legal-option-value* }

Parameters

- **policy-name** The name of the login policy. Specify root to modify the root login policy.
- **policy-option-name** The name of the policy option. To view a list of default login policy option names and descriptions, see the Remarks section of "CREATE LOGIN POLICY statement" on page 475.
- **policy-option-value** The value assigned to the login policy option. If you specify UNLIMITED, no limits are used. If you specify default, the default limits are used. To view a list of default login policy option values, see the Remarks section of "CREATE LOGIN POLICY statement" on page 475.

Remarks

When a login policy is altered, changes are immediately applied to all users.

Permissions

Must have DBA authority.

Side effects

None.

See also

- "Altering a login policy" [*SQL Anywhere Server - Database Administration*]
- "ALTER USER statement" on page 402
- "COMMENT statement" on page 425
- "CREATE LOGIN POLICY statement" on page 475
- "CREATE USER statement" on page 546
- "DROP LOGIN POLICY statement" on page 577
- "DROP USER statement" on page 591
- "Managing login policies overview" [*SQL Anywhere Server - Database Administration*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Examples

The following example alters the Test1 login policy. This example changes the locked and max_connections options. The locked value indicates that users with the policy are prohibited from establishing new connections and the max_connections value indicates the number of concurrent connections allowed.

```
ALTER LOGIN POLICY Test1
locked=ON
max_connections=5;
```

ALTER MATERIALIZED VIEW statement

Use this statement to alter a materialized view.

Syntax

ALTER MATERIALIZED VIEW [*owner*.]*materialized-view-name* {
 SET HIDDEN
| { **ENABLE** | **DISABLE** }
| { **ENABLE** | **DISABLE** } **USE IN OPTIMIZATION**
| { **ADD PCTFREE** *percent-free-space* | **DROP PCTFREE** }
| [**NOT**] **ENCRYPTED**
[{ **IMMEDIATE** | **MANUAL** } **REFRESH**]
}

percent-free-space : integer

Parameters

- **SET HIDDEN clause** Use the SET HIDDEN clause to obfuscate the definition of a materialized view. *This setting is irreversible*. For more information, see "Hide materialized views" [*SQL Anywhere Server - SQL Usage*].

- **ENABLE clause** Use the ENABLE clause to enable a disabled materialized view, making it available for the database server to use. This clause has no effect on a view that is already enabled. After using this clause, you must refresh the view to initialize it, and recreate any text indexes that were dropped when the view was disabled.

- **DISABLE clause** Use the DISABLE clause to disable use of the view by the database server. When you disable a materialized view, the database server drops the data and indexes for the view.

- **{ ENABLE | DISABLE } USE IN OPTIMIZATION clause** Use this clause to specify whether you want the materialized view to be available the optimizer to use. If you specify DISABLE USE IN OPTIMIZATION, the materialized view is used only when executing queries that explicitly reference the view. The default is ENABLE USE IN OPTIMIZATION. See "Enable and disable optimizer use of a materialized view" [*SQL Anywhere Server - SQL Usage*].

- **ADD PCTFREE clause** Specify the percentage of free space you want to reserve on each page. The free space is used if rows increase in size when the data is updated. If there is no free space on a page, every increase in the size of a row on that page requires the row to be split across multiple pages, causing row fragmentation and possible performance degradation.

The value of *percent-free-space* is an integer between 0 and 100. The value 0 specifies that no free space is to be left on each page—each page is to be fully packed. A high value causes each row to be inserted into a page by itself. If PCTFREE is not set, or is dropped, the default PCTFREE setting is applied according to the database page size (200 bytes for a 4 KB page size, and 100 bytes for a 2 KB page size).

- **DROP PCTFREE clause** Removes the PCTFREE setting currently in effect for the materialized view, and applies the default PCTFREE according to the database page size.
- **[NOT] ENCRYPTED clause** Specify whether to encrypt the materialized view data. By default, materialized view data is not encrypted at creation time. To encrypt a materialized view, specify ENCRYPTED. To decrypt a materialized view, specify NOT ENCRYPTED.
- **REFRESH clause** Use the REFRESH clause to change the refresh type for the materialized view:
  - **IMMEDIATE REFRESH** Use the IMMEDIATE REFRESH clause to change a manual view to an immediate view. The manual view must be valid and uninitialized to change the refresh type to IMMEDIATE REFRESH. If the view is in an initialized state, execute a TRUNCATE statement to change the state to uninitialized before executing the ALTER MATERIALIZED VIEW...IMMEDIATE REFRESH. See "TRUNCATE statement" on page 768.

    For information about conditions that must be met before you can alter the view to IMMEDIATE REFRESH, see "Additional restrictions for immediate views" [*SQL Anywhere Server - SQL Usage*].
  - **MANUAL REFRESH** Use the MANUAL REFRESH clause to change an immediate view to a manual view.

  For more information about refresh types, see "Manual and immediate materialized views" [*SQL Anywhere Server - SQL Usage*].

  For more information about statuses, see "Materialized view statuses and properties" [*SQL Anywhere Server - SQL Usage*].

Remarks

If you alter a materialized view owned by another user, you must qualify the name by including the owner (for example, GROUPO.EmployeeConfidential). If you don't qualify the name, the database server looks for a materialized view with that name owned by you and alters it. If there isn't one, it returns an error.

When you disable a materialized view (DISABLE clause), it is no longer available for the database server to use for answering queries. As well, the data and indexes are dropped, and the refresh type changes to manual. Any dependent regular views are also disabled.

The DISABLE clause requires exclusive access not only to the view being disabled, but to any dependent views, since they are also disabled. See "Enable and disable materialized views" [*SQL Anywhere Server - SQL Usage*].

Table encryption must already be enabled on the database to encrypt a materialized view (ENCRYPTED clause). The materialized view is then encrypted using the encryption key and algorithm specified at database creation time. See "Encrypt and decrypt materialized views" [*SQL Anywhere Server - SQL Usage*].

Permissions

To execute the ALTER MATERIALIZED VIEW statement you must own the view or have DBA authority.

If you do not have DBA authority but want to alter a materialized view to be immediate (ALTER MATERIALIZED VIEW ... IMMEDIATE REFRESH), you must own the view and all the tables it references.

The only operations a user can perform on a materialized view to change its data are refreshing, truncating, and disabling. However, immediate views are automatically updated by the database server. That is, once an immediate view is enabled and initialized, the database server maintains it automatically, without additional permissions checking.

Side effects

Automatic commit.

See also

- "CREATE MATERIALIZED VIEW statement" on page 477
- "REFRESH MATERIALIZED VIEW statement" on page 703
- "sa_refresh_materialized_views system procedure" on page 926
- "TRUNCATE statement" on page 768
- "DROP MATERIALIZED VIEW statement" on page 578
- "Working with materialized views" [*SQL Anywhere Server - SQL Usage*]
- "View dependencies" [*SQL Anywhere Server - SQL Usage*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statements creates the EmployeeConfid88 materialized view and then disables its use in optimization:

```
CREATE MATERIALIZED VIEW EmployeeConfid88 AS
   SELECT EmployeeID, Employees.DepartmentID, SocialSecurityNumber, Salary,
ManagerID,
      Departments.DepartmentName, Departments.DepartmentHeadID
   FROM Employees, Departments
   WHERE Employees.DepartmentID=Departments.DepartmentID;
REFRESH MATERIALIZED VIEW EmployeeConfid88;
ALTER MATERIALIZED VIEW GROUPO.EmployeeConfid88 DISABLE USE IN OPTIMIZATION;
```

Caution
When you are done with this example, you should drop the materialized view you created. Otherwise, you will not be able to make schema changes to its underlying tables Employees and Departments, when trying out other examples. You cannot alter the schema of a table that has enabled, dependent materialized view. See "Drop materialized views" [*SQL Anywhere Server - SQL Usage*].

ALTER PROCEDURE statement

Use this statement to modify a procedure, or to enable and disable a procedure for replication with Sybase Replication Server. You must include the entire new procedure in the ALTER PROCEDURE statement.

You can use PROC as a synonym for PROCEDURE.

Syntax 1

ALTER PROCEDURE [*owner*.]*procedure-name procedure-definition*

procedure-definition : CREATE PROCEDURE syntax

Syntax 2

ALTER PROCEDURE [*owner*.]*procedure-name*
REPLICATE { **ON** | **OFF** }

Syntax 3

ALTER PROCEDURE [*owner*.]*procedure-name*
SET HIDDEN

Syntax 4

ALTER PROCEDURE [*owner*.]*procedure-name*
RECOMPILE

Remarks

Syntax 1 The ALTER PROCEDURE statement is identical in syntax to the CREATE PROCEDURE statement except for the first word. Either version of the CREATE PROCEDURE statement can be altered.

Existing permissions on the procedure are maintained, and do not have to be reassigned. If a DROP PROCEDURE and CREATE PROCEDURE were carried out, execute permissions would have to be reassigned.

Syntax 2 If a procedure is to be replicated to other sites using Sybase Replication Server, you must set REPLICATE ON for the procedure.

Syntax 3 Use SET HIDDEN to obfuscate the definition of the associated procedure and cause it to become unreadable. The procedure can be unloaded and reloaded into other databases.

This setting is irreversible. If you will need the original source again, you must maintain it outside the database.

If SET HIDDEN is used, debugging using the debugger will not show the procedure definition, nor will it be available through procedure profiling.

You cannot combine Syntax 2 with Syntax 1. You cannot combine Syntax 3 with either Syntax 1 or 2.

Syntax 4 Use the RECOMPILE syntax to recompile a stored procedure. When you recompile a procedure, the definition stored in the catalog is re-parsed and the syntax is verified. For procedures that generate a result set but do not include a RESULT clause, the database server attempts to determine the result set characteristics for the procedure and stores the information in the catalog. This can be useful if a table referenced by the procedure has been altered to add, remove, or rename columns since the procedure was created.

The procedure definition is not changed by recompiling. You can recompile procedures with definitions hidden with the SET HIDDEN clause, but their definitions remain hidden.

Permissions

Must be the owner of the procedure or have DBA authority.

Side effects

Automatic commit.

See also

- "CREATE PROCEDURE statement (web services)" on page 495
- "ALTER FUNCTION statement" on page 369
- "DROP PROCEDURE statement" on page 579
- "Hiding the contents of procedures, functions, triggers and views" [*SQL Anywhere Server - SQL Usage*]

Standards and compatibility

- **SQL/2003** Vendor extension.

ALTER PUBLICATION statement [MobiLink] [SQL Remote]

Use this statement to alter a publication. In MobiLink, a publication identifies synchronized data in a SQL Anywhere remote database. In SQL Remote, a publication identifies replicated data in both consolidated and remote databases.

Syntax

```
ALTER PUBLICATION [ owner.]publication-name alterpub-clause, ...

alterpub-clause:
  ADD article-definition
| ALTER article-definition
| { DELETE | DROP } TABLE [ owner.]table-name
| RENAME publication-name

article-definition :
TABLE table-name [ ( column-name, ... ) ]
[ WHERE search-condition ]
[ SUBSCRIBE BY expression ]
[ USING ( [PROCEDURE ] [ owner.][procedure-name ]
  FOR UPLOAD { INSERT | DELETE | UPDATE }, ... ) ]
```

Remarks

This statement is applicable only to MobiLink and SQL Remote.

The ALTER PUBLICATION statement alters a publication in the database. The contribution to a publication from one table is called an article. Changes can be made to a publication by adding, modifying, or deleting articles, or by renaming the publication. If an article is modified, the entire definition of the modified article must be entered.

It is recommended that you perform a successful synchronization of the publication immediately before you alter it.

You cannot use the WHERE clause for publications that are defined as FOR DOWNLOAD ONLY or WITH SCRIPTED UPLOAD.

The SUBSCRIBE BY clause applies to SQL Remote only.

The USING clause is for scripted upload only.

You set options for a MobiLink publication with the ADD OPTION clause in the ALTER SYNCHRONIZATION SUBSCRIPTION statement or CREATE SYNCHRONIZATION SUBSCRIPTION statement.

Permissions

Must have DBA authority, or be the owner of the publication. Requires exclusive access to all tables referred to in the statement.

Side effects

Automatic commit.

See also

- "CREATE PUBLICATION statement [MobiLink] [SQL Remote]" on page 500
- "DROP PUBLICATION statement [MobiLink] [SQL Remote]" on page 580
- SQL Anywhere MobiLink clients: "Publishing data" [*MobiLink - Client Administration*]
- UltraLite MobiLink clients: "Designing synchronization in UltraLite" [*UltraLite - Database Management and Reference*]
- "ALTER SYNCHRONIZATION SUBSCRIPTION statement [MobiLink]" on page 386
- "CREATE SYNCHRONIZATION SUBSCRIPTION statement [MobiLink]" on page 519
- "ISYSSYNC system table" on page 807

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement adds the Customers table to the pub_contact publication.

```
ALTER PUBLICATION pub_contact
   ADD TABLE Customers;
```

ALTER REMOTE MESSAGE TYPE statement [SQL Remote]

Use this statement to change the publisher's message system, or the publisher's address for a given message system, for a message type that has been created.

Syntax

ALTER REMOTE MESSAGE TYPE *message-system*
ADDRESS *address*

message-system: **FILE** | **FTP** | **SMTP**

address: *string*

Parameters

- **message-system** One of the message systems supported by SQL Remote. It must be one of the following values:
  - **address** A string containing a valid address for the specified message system.

Remarks

The statement changes the publisher's address for a given message type.

The Message Agent sends outgoing messages from a database by one of the supported message links. The extraction utility uses this address when executing the GRANT CONSOLIDATE statement in the remote database.

The address is the publisher's address under the specified message system. If it is an email system, the address string must be a valid email address. If it is a file-sharing system, the address string is a subdirectory of the directory specified by the SQLREMOTE environment variable, or of the current directory if that is not set. You can override this setting on the GRANT CONSOLIDATE statement at the remote database.

Permissions

Must have DBA authority.

Side effects

Automatic commit.

See also

- “CREATE REMOTE MESSAGE TYPE statement [SQL Remote]” on page 503

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement changes the publisher's address for the FILE message link to new_addr.

```
ALTER REMOTE MESSAGE TYPE file
ADDRESS 'new_addr';
```

ALTER SERVER statement

Use this statement to modify the attributes of a remote server.

Syntax

ALTER SERVER *server-name*
[**CLASS** *server-class*]
[**USING** *connection-info*]
[**CAPABILITY** *cap-name* { **ON** | **OFF** }]
[**CONNECTION CLOSE** [**CURRENT** | **ALL** | *connection-id*]]

server-class :
  SAODBC
| **ASEODBC**
| **DB2ODBC**
| **MSSODBC**
| **ORAODBC**
| **MSACCESSODBC**
| **MYSQLODBC**
| **ULODBC**
| **ADSODBC**
| **ODBC**
| **SAJDBC**
| **ASEJDBC**

connection-info :
computer-name:*port-number*[/*dbname*] | *data-source-name*

Parameters

- **CLASS clause** The CLASS clause is specified to change the server class.

  For more information about server classes and how to configure a server, see "Server classes for remote data access" [*SQL Anywhere Server - SQL Usage*].

- **USING clause** The USING clause is specified to change the server connection information. For information about *connection-info*, see "CREATE SERVER statement" on page 506.

- **CAPABILITY clause** The CAPABILITY clause turns a server capability ON or OFF. Server capabilities are stored in the ISYSCAPABILITY system table. The names of these capabilities are accessible via the SYSCAPABILITYNAME system view. The ISYSCAPABILITY system table and SYSCAPABILITYNAME system view is not populated with data until the first connection to a remote server is made. For subsequent connections, the database server's capabilities are obtained from the ISYSCAPABILITY system table.

  In general, you do not need to alter a server's capabilities. It may be necessary to alter capabilities of a generic server of class ODBC.

- **CONNECTION CLOSE clause** When a user creates a connection to a remote server, the remote connection is not closed until the user disconnects from the local database. The CONNECTION CLOSE clause allows you to explicitly close connections to a remote server. You may find this useful when a remote connection becomes inactive or is no longer needed.

  The following SQL statements are equivalent and close the current connection to the remote server:

```
ALTER SERVER server-name CONNECTION CLOSE;
```

```
ALTER SERVER server-name CONNECTION CLOSE CURRENT;
```

You can close both ODBC and JDBC connections to a remote server using this syntax. You do not need DBA authority to execute either of these statements.

You can also disconnect a specific remote ODBC connection by specifying a connection ID, or disconnect all remote ODBC connections by specifying the ALL keyword. If you attempt to close a JDBC connection by specifying the connection ID or the ALL keyword, an error occurs. When the connection identified by *connection-id* is not the current local connection, the user must have DBA authority to be able to close the connection.

Remarks

The ALTER SERVER statement modifies the attributes of a server. These changes do not take effect until the next connection to the remote server.

Permissions

Must have RESOURCE authority.

Side effects

Automatic commit.

See also

- "SYSCAPABILITY system view" on page 991
- "SYSCAPABILITYNAME system view" on page 992
- "CREATE SERVER statement" on page 506
- "DROP SERVER statement" on page 581
- "Server classes for remote data access" [*SQL Anywhere Server - SQL Usage*]
- "Troubleshooting remote data access" [*SQL Anywhere Server - SQL Usage*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following example changes the server class of the Adaptive Server Enterprise server named ase_prod so its connection to SQL Anywhere is ODBC-based. Its Data Source Name is ase_prod.

```
ALTER SERVER ase_prod
CLASS 'ASEODBC'
USING 'ase_prod';
```

The following example changes a capability of server infodc.

```
ALTER SERVER infodc
CAPABILITY 'insert select' OFF;
```

The following example closes all connections to the remote server named rem_test.

```
ALTER SERVER rem_test
CONNECTION CLOSE ALL;
```

The following example closes the connection to the remote server named rem_test that has the connection ID 142536.

```
ALTER SERVER rem_test
CONNECTION CLOSE 142536;
```

ALTER SERVICE statement

Use this statement to alter a web service.

Syntax 1 - DISH service

ALTER SERVICE *service-name*
[**TYPE 'DISH'**]
[**GROUP** { *group-name* | **NULL** }]
[**FORMAT** { **'DNET'** | **'CONCRETE'** [**EXPLICIT** { **ON** | **OFF** }]| **'XML'** | **NULL** }]
[*common-attributes*]

Syntax 2 - SOAP service

ALTER SERVICE *service-name*
[**TYPE 'SOAP'**]
[**DATATYPE** { **ON** | **OFF** | **IN** | **OUT** }]
[**FORMAT** { **'DNET'** | **'CONCRETE'** [**EXPLICIT** { **ON** | **OFF** }] | **'XML'** | **NULL** }]
[*common-attributes*]
[**AS** *statement*]

Syntax 3 - Miscellaneous services

ALTER SERVICE *service-name*
[**TYPE** { **'RAW'** | **'HTML'** | **'XML'** }]
[**URL** [**PATH**] { **ON** | **OFF** | **ELEMENTS** }]
[*common-attributes*]
[**AS** { *statement* | **NULL** }]

common-attributes:
[**AUTHORIZATION** { **ON** | **OFF** }]
[**ENABLE** | **DISABLE**]
[**METHODS** **'***method,...***'**]
[**SECURE** { **ON** | **OFF** }]
[**USER** { *user-name* | **NULL** }]

method:
DEFAULT
| **POST**
| **GET**
| **HEAD**
| **PUT**
| **DELETE**
| **NONE**
| *****

Parameters

The description of the ALTER SERVICE parameters are identical to those of the CREATE SERVICE statement. See "CREATE SERVICE statement" on page 509.

Remarks

The ALTER SERVICE statement makes changes to the ISYSWEBSERVICE system table, and allows the database server to act as a web server.

Permissions

Must have DBA authority.

Side effects

None.

See also

- "Using SOAP services" [*SQL Anywhere Server - Programming*]
- "CREATE SERVICE statement" on page 509
- "DROP SERVICE statement" on page 582
- "SYSWEBSERVICE system view" on page 1043
- "-xs server option" [*SQL Anywhere Server - Database Administration*]
- "SQL Anywhere web services" [*SQL Anywhere Server - Programming*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

To set up a web server quickly, start a database server with the -xs (http or https) option, then execute the following statements:

```
CREATE SERVICE tables TYPE 'HTML';

ALTER SERVICE tables
   AUTHORIZATION OFF
   USER DBA
   AS SELECT *
      FROM SYS.SYSTAB;
```

After executing these statements, use any web browser to open the URL *http://localhost/tables*.

ALTER STATISTICS statement

Use this statement to control whether statistics are automatically updated on a column, or columns, in a table.

Syntax

ALTER STATISTICS
[**ON**] *table* [**(** *column1* [**,** *column2* ...] **)**]
AUTO UPDATE { **ENABLE** | **DISABLE** }

Parameters

- **ON** The word ON is optional. Including it has no impact on the execution of the statement.
- **AUTO UPDATE clause** Specify whether to enable or disable automatic updating of statistics for the column(s).

Remarks

During normal execution of queries, DML statements, and LOAD TABLE statements, the database server automatically maintains column statistics for use by the optimizer. The benefit of maintaining statistics for some columns may not outweigh the overhead necessary to generate them. For example, if a column is not queried often, or if it is subject to periodic mass changes that are eventually rolled back, there is little value in continually updating its statistics. Use the ALTER STATISTICS statement to suppress the automatic updating of statistics for these types of columns.

When automatic updating is disabled, you can still update the statistics for the column using the CREATE STATISTICS and DROP STATISTICS statements. However, you should only update them if it has been determined that it would have a positive impact on performance. Normally, column statistics should not be disabled.

Permissions

Must have DBA authority.

Side effects

If automatic updating has been disabled, the statistics may become out of date. Re-enabling will not immediately bring them up to date. Run the CREATE STATISTICS statement to recreate them, if necessary.

See also

- "CREATE STATISTICS statement" on page 516
- "DROP STATISTICS statement" on page 583

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following example disables the automatic updating of statistics on the Street column in the Customers table:

```
ALTER STATISTICS Customers ( Street ) AUTO UPDATE DISABLE;
```

ALTER SYNCHRONIZATION PROFILE statement [MobiLink]

Use this statement to change a SQL Anywhere synchronization profile. Synchronization profiles define how a SQL Anywhere database synchronizes with the MobiLink server.

Syntax

ALTER SYNCHRONIZATION PROFILE *name*
{ **REPLACE** | **MERGE** } *string*

Parameters

- **name** The name of the synchronization profile to alter.
- **REPLACE clause** Use this clause to drop the options currently defined for the profile, and add the specified options instead.
- **MERGE clause** Use this clause to change existing, or add new, options to a synchronization profile.
- **string** A string of one or more synchronization option value pairs, separated by semicolons. For example, `'option1=value1;option2=value2'`.

Remarks

For a list of the synchronization profile options supported by SQL Anywhere, see "CREATE SYNCHRONIZATION PROFILE statement [MobiLink]" on page 518.

When **REPLACE** is used in the ALTER SYNCHRONIZATION PROFILE statement the entire synchronization profile is replaced with the specified string. This is equivalent to doing a drop then a create. For example:

```
ALTER SYNCHRONIZATION PROFILE myProfile
REPLACE 'publication=p1;verbosity=high'
```

is equivalent to:

```
DROP SYNCHRONIZATION PROFILE myProfile;
CREATE SYNCHRONIZATION PROFILE myProfile 'publication=p1;verbosity=high'
```

When **MERGE** is used in the ALTER SYNCHRONIZATION PROFILE statement, options specified in the string will be added to those already in the synchronization profile. If an option in the string already exists in profile, then the value from the string will replace the value already stored in the profile.

For example, executing the following statements will leave the profile *myProfile* with the value *publication=p2;verbosity=high;uploadonly=on*.

```
CREATE SYNCHRONIZATION PROFILE myProfile 'publication=p1;verbosity=high';
ALTER SYNCHRONIZATION PROFILE myProfile MERGE 'publication=p2;uploadonly=on'
```

Permissions

Must have DBA authority.

Side effects

Automatic commit.

See also

- "CREATE SYNCHRONIZATION PROFILE statement [MobiLink]" on page 518
- "DROP SYNCHRONIZATION PROFILE statement [MobiLink]" on page 586

Standards and compatibility

- **SQL/2003** Vendor extension.

ALTER SYNCHRONIZATION SUBSCRIPTION statement [MobiLink]

Use this statement in a SQL Anywhere remote database to alter the properties of a subscription of a MobiLink user to a publication.

Syntax

ALTER SYNCHRONIZATION SUBSCRIPTION
TO *publication-name*
[**FOR** *ml_username*, ...]
[**TYPE** *network-protocol*]
[**ADDRESS** *protocol-options*]
[**ADD OPTION** *option*=*value*, ...]
[**ALTER OPTION** *option*=*value*, ...]
[**DELETE** { **ALL OPTION** | **OPTION** *option*, ... }]

ml_username: *identifier*

network-protocol: **http** | **https** | **tls** |**tcpip**

protocol-options: *string*

value: *string* | *integer*

Parameters

- **TO clause** Specify the name of a publication.
- **FOR clause** Specify one or more MobiLink user names.

  Omit the FOR clause to set the protocol type, protocol options, and extended options for a publication.

  For information about how dbmlsync processes options that are specified in different locations, see "Priority order" [*MobiLink - Client Administration*].
- **TYPE clause** This clause specifies the network protocol to use for synchronization. The default protocol is tcpip.

  For more information about communication protocols, see "CommunicationType (ctp) extended option" [*MobiLink - Client Administration*].
- **ADDRESS clause** This clause specifies network protocol options, including the location of the MobiLink server.

  For a complete list of protocol options, see "MobiLink client network protocol option summary" [*MobiLink - Client Administration*].

- **ADD OPTION, ALTER OPTION, DELETE OPTION, and DELETE ALL OPTION clauses** These clauses allow you to add, alter, delete, or delete all extended options. You may specify only one option in each clause.

  The values for each option cannot contain the characters "=" or "**,**" or "**;**".

  For a complete list of options, see "MobiLink SQL Anywhere client extended options" [*MobiLink - Client Administration*].

Remarks

The *network-protocol*, *protocol-options*, and *options* can be set in several places.

For information about how dbmlsync processes options that are specified in different locations, see "Priority order" [*MobiLink - Client Administration*].

This statement causes options and other information to be stored in the SQL Anywhere ISYSSYNC system table. Anyone with DBA authority for the database can view the information, which could include passwords and encryption certificates. To avoid this potential security issue, you can specify the information about the dbmlsync command line.

See "dbmlsync syntax" [*MobiLink - Client Administration*].

Permissions

Must have DBA authority. Requires exclusive access to all tables referred to in the publication.

Side effects

Automatic commit.

See also

- "CREATE PUBLICATION statement [MobiLink] [SQL Remote]" on page 500
- "DROP PUBLICATION statement [MobiLink] [SQL Remote]" on page 580
- SQL Anywhere MobiLink clients: "Creating synchronization subscriptions" [*MobiLink - Client Administration*]
- UltraLite MobiLink clients: "Designing synchronization in UltraLite" [*UltraLite - Database Management and Reference*]
- "ISYSSYNC system table" on page 807

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following example changes the address of the MobiLink server:

```
ALTER SYNCHRONIZATION SUBSCRIPTION
TO p1
FOR ml1
TYPE TCPIP
ADDRESS 'host=10.11.12.132;port=2439';
```

ALTER SYNCHRONIZATION USER statement [MobiLink]

Use this statement in a SQL Anywhere remote database to alter the properties of a MobiLink user.

Syntax

ALTER SYNCHRONIZATION USER *ml_username*
[**TYPE** *network-protocol*]
[**ADDRESS** *protocol-options*]
[**ADD OPTION** *option*=*value*, ...]
[**ALTER OPTION** *option*=*value*, ...]
[**DELETE** { **ALL OPTION** | **OPTION** *option* }]

ml_username: *identifier*

network-protocol: **http** | **https** | **tls** | **tcpip**

protocol-options: *string*

value: *string* | *integer*

Parameters

- **TYPE clause** This clause specifies the network protocol to use for synchronization.

  For more information about communication protocols, see "CommunicationType (ctp) extended option" [*MobiLink - Client Administration*].

- **ADDRESS clause** This clause specifies network protocol options, including the location of the MobiLink server.

  For a complete list of protocol options, see "MobiLink client network protocol option summary" [*MobiLink - Client Administration*].

- **ADD OPTION, ALTER OPTION, DELETE OPTION, and DELETE ALL OPTION clauses** These clauses allow you to add, modify, delete, or delete all extended options. You may specify only one option in each clause.

  For a complete list of options, see "MobiLink SQL Anywhere client extended options" [*MobiLink - Client Administration*].

Remarks

The *network-protocol*, *protocol-options*, and *options* can be set in several places.

For information about how dbmlsync processes options that are specified in different locations, see "Priority order" [*MobiLink - Client Administration*].

This statement causes options and other information to be stored in the SQL Anywhere ISYSSYNC system table. Anyone with DBA authority for the database can view the information, which could include passwords and encryption certificates. To avoid this potential security issue, you can specify the information about the dbmlsync command line.

See "dbmlsync syntax" [*MobiLink - Client Administration*].

Permissions

Must have DBA authority. Requires exclusive access to all tables referred to in the publication.

Side effects

Automatic commit.

See also

- "CREATE SYNCHRONIZATION USER statement [MobiLink]" on page 521
- "DROP SYNCHRONIZATION USER statement [MobiLink]" on page 587
- "MobiLink users" [*MobiLink - Client Administration*]
- "ISYSSYNC system table" on page 807

Standards and compatibility

- **SQL/2003** Vendor extension.

ALTER TABLE statement

Use this statement to modify a table definition, disable dependent views, or enable a table to take part in Replication Server replication.

Syntax

ALTER TABLE [*owner.*]*table-name* { *alter-clause*, ... }

alter-clause :
ADD *create-clause*
| **ALTER** *column-name column-alteration*
| **ALTER** [**CONSTRAINT** *constraint-name*] **CHECK (** *condition* **)**
| **DROP** *drop-object*
| **RENAME** *rename-object*
| *table-alteration*

create-clause :
column-name [**AS**] *column-data-type* [*new-column-attribute* ...]
| *table-constraint*
| **PCTFREE** *integer*

column-alteration :
{ *column-data-type* | *alterable-column-attribute* } [*alterable-column-attribute* ...]
| **SET COMPUTE (** *compute-expression* **)**
| **ADD** [*constraint-name*] **CHECK (** *condition* **)**
| **DROP** { **DEFAULT** | **COMPUTE** | **CHECK** | **CONSTRAINT** *constraint-name* }

drop-object :
column-name
| **CHECK**
| **CONSTRAINT** *constraint-name*
| **UNIQUE** [**CLUSTERED**] **(** *index-columns-list* **)**
| **FOREIGN KEY** *fkey-name*
| **PRIMARY KEY**

rename-object :
new-table-name
| *column-name* **TO** *new-column-name*
| **CONSTRAINT** *constraint-name* **TO** *new-constraint-name*

table-alteration :
PCTFREE DEFAULT
| **REPLICATE** { **ON** | **OFF** }
| [**NOT**] **ENCRYPTED**

new-column-attribute :
NULL
| **DEFAULT** *default-value*
| **COMPRESSED**
| **INLINE** { *inline-length* | **USE DEFAULT** }
| **PREFIX** { *prefix-length* | **USE DEFAULT** }
| [**NO**] **INDEX**
| **IDENTITY**
| **COMPUTE (** *expression* **)**
| *column-constraint*

table-constraint :
[**CONSTRAINT** *constraint-name*] {
  CHECK (*condition* **)**
  | **UNIQUE** [**CLUSTERED** | **NONCLUSTERED**] **(** *column-name* [**ASC** | **DESC**], ... **)**
  | **PRIMARY KEY** [**CLUSTERED** | **NONCLUSTERED**] **(** *column-name* [**ASC** | **DESC**], ... **)**
  | *foreign-key*
 }

column-constraint :
[**CONSTRAINT** *constraint-name*] {
  CHECK (*condition* **)**
  | **UNIQUE** [**CLUSTERED** | **NONCLUSTERED**] [**ASC** | **DESC**]
  | **PRIMARY KEY** [**CLUSTERED** | **NONCLUSTERED**] [**ASC** | **DESC**]
  | **REFERENCES** *table-name* [**(** *column-name* **)**]
    [**MATCH** [**UNIQUE**] { **SIMPLE** | **FULL** }]
    [*actions*][**CLUSTERED** | **NONCLUSTERED**]
  | **NOT NULL**
 }

alterable-column-attribute :
 [**NOT**] **NULL**
| **DEFAULT** *default-value*
| [**CONSTRAINT** *constraint-name*] **CHECK** { **NULL** | **(** *condition* **)** }
| [**NOT**] **COMPRESSED**
| **INLINE** { *inline-length* | **USE DEFAULT** }
| **PREFIX** { *prefix-length* | **USE DEFAULT** }
| [**NO**] **INDEX**

default-value :
 special-value
| *string*
| *global variable*
| [**-**] *number*
| **(** *constant-expression* **)**
| *built-in-function* **(** *constant-expression* **)**
| **AUTOINCREMENT**

| **GLOBAL AUTOINCREMENT** [**(** *partition-size* **)**]
| **NULL**
| **TIMESTAMP**
| **UTC TIMESTAMP**
| **LAST USER**
| **USER**

special-value :
CURRENT {
  DATABASE
  | **DATE**
  | **REMOTE USER**
  | **TIME**
  | **TIMESTAMP**
  | **UTC TIMESTAMP**
  | **USER**
  | **PUBLISHER** }

foreign-key :
[**NOT NULL**] **FOREIGN KEY** [*role-name*]
  [**(** *column-name* [**ASC** | **DESC**], ... **)**
  REFERENCES *table-name*
  [**(** *pkey-column-list* **)**]
  [**MATCH** [**UNIQUE**] { **SIMPLE** | **FULL** }]
  [*actions*] [**CHECK ON COMMIT**] [**CLUSTERED**]
  [**FOR OLAP WORKLOAD**]

actions :
[**ON UPDATE** *action*] [**ON DELETE** *action*]

action :
CASCADE | **SET NULL** | **SET DEFAULT** | **RESTRICT**

Syntax 2 - Disabling view dependencies

ALTER TABLE [*owner.*]*table-name* {
 DISABLE VIEW DEPENDENCIES
}

Parameters

- **Adding clauses** The following section explains the clauses used for adding definition for a column or table:
  - **ADD column-name [AS] column-data-type [new-column-attribute ...] clause** Use this syntax to add a new column to the table, specifying the data type and attributes for the column. For more information about what data type to specify, see "SQL data types" on page 77.
  - **NULL and NOT NULL clauses** Use this clause to specify whether to allow NULLs in the column. With the exception of bit type columns, new columns allow NULL values. Bit type columns automatically have the NOT NULL constraint applied when created.
  - **DEFAULT clause** Sets the default value for the column. All rows in the column are populated with this value. For information about possible default values, see "CREATE TABLE statement" on page 523.

- **column-constraint clause** Use this clause to add a constraint to the column. With the exception of CHECK constraints, when a new constraint is added, the database server validates existing values to confirm that they satisfy the constraint. CHECK constraints are enforced only for operations that occur after the table alteration is complete. Possible column constraints include:
  - **CHECK clause** Use this subclause to add a check condition for the column.
  - **UNIQUE clause** Use this subclause to specify that values in the column must be unique, and whether to create a clustered or nonclustered index.
  - **PRIMARY KEY clause** Use this subclause to make the column a primary key, and whether to use a clustered index. For more information about clustered indexes, see "Using clustered indexes" [*SQL Anywhere Server - SQL Usage*].
  - **REFERENCES clause** Use this subclause to add or alter a reference to another table, to specify how matches are handled, and to specify whether to use a clustered index. For more information about clustered indexes, see "Using clustered indexes" [*SQL Anywhere Server - SQL Usage*].
  - **NULL and NOT NULL clauses** Use this clause to specify whether to allow NULL values in the column. By default, NULLs are allowed.
- **COMPRESSED clause** Use this clause to compress the column.
- **INLINE and PREFIX clauses** When storing BLOBs (character and binary data types only), use the INLINE and PREFIX clauses to specify how much of a BLOB, in bytes, to keep within a row. For more information, see the INLINE and PREFIX clauses in "CREATE TABLE statement" on page 523.
- **INDEX and NO INDEX clauses** Use this clause to specify whether to build indexes on large BLOBs in this column. For more information about how to use this clause, see the corresponding section for the [NO] INDEX clause in the "CREATE TABLE statement" on page 523.
- **IDENTITY clause** This clause is equivalent to AUTOINCREMENT, and is provided for compatibility with T-SQL. See the description for AUTOINCREMENT in "CREATE TABLE statement" on page 523.
- **COMPUTE clause** Use this clause to ensure that the value in the column reflects the value of *expression*. For more information about what is allowed for the COMPUTE clause, see "CREATE TABLE statement" on page 523.
- **ADD table-constraint clause** Use this clause to add a table constraint. Table constraints place limits on what columns in the table can hold. When adding or altering table constraints, the optional constraint name allows you to modify or drop individual constraints. Following is a list of the table constraints you can add.
  - **UNIQUE** Use this subclause to specify that values in the columns specified in *column-list* must be unique, and, optionally, whether to use a clustered index. For more information about this constraint, see "CREATE TABLE statement" on page 523.
  - **PRIMARY KEY** Use this subclause to add or alter the primary key for the table, and specify whether to use a clustered index. The table must not already have a primary key that was created by the CREATE TABLE statement or another ALTER TABLE statement. For more information about this constraint, see "CREATE TABLE statement" on page 523.

For more information about clustered indexes, see "Using clustered indexes" [*SQL Anywhere Server - SQL Usage*].

- **foreign-key** Use this subclause to add a foreign key as a constraint. If you use a subclause other than ADD FOREIGN KEY with the ALTER TABLE statement on a table with dependent materialized views, the ALTER TABLE statement fails. For all other clauses, you must disable the dependent materialized views and then re-enable them when your changes are complete.

  For more information about this constraint, see "CREATE TABLE statement" on page 523.

- ○ **ADD PCTFREE clause** Specify the percentage of free space you want to reserve in each table page. The free space is used if rows increase in size when the data is updated. If there is no free space in a table page, every increase in the size of a row on that page requires the row to be split across multiple table pages, causing row fragmentation and possible performance degradation. A free space percentage of 0 specifies that no free space is to be left on each page—each page is to be fully packed. A high free space percentage causes each row to be inserted into a page by itself. If PCTFREE is not set, or is dropped, the default PCTFREE value is applied according to the database page size (200 bytes for a 4 KB, and up, page size). The value for PCTFREE is stored in the ISYSTAB system table. When PCTFREE is set, all subsequent inserts into table pages use the new value, but rows that were already inserted are not affected. The value persists until it is changed. The PCTFREE specification can be used for base, global temporary, or local temporary tables.

- **Altering clauses** The following section explains the clauses used for altering the definition for a column or table:

  - ○ **ALTER column-name column-alteration clause** Use this clause to change attributes for the specified column. If a column is contained in a unique constraint, a foreign key, or a primary key, you can change only the default for the column. However, for any other change, you must delete the key or constraint before the column can be modified. Following is a list of the alterations you can make. For further information about these attributes, see "CREATE TABLE statement" on page 523.

  - ○ **column-data-type clause** Use this clause to alter the length or data type of the column. If necessary, the data in the modified column is converted to the new data type. If a conversion error occurs, the operation will fail and the table is left unchanged. You cannot reduce the size of a column. For example, you cannot change a column from a VARCHAR(100) to a VARCHAR(50).

  - ○ **[NOT] NULL clause** Use this clause to change whether NULLs are allowed in the column. If NOT NULL is specified, and the column value is NULL in any of the existing rows, then the operation fails and the table is left unchanged.

  - ○ **CHECK NULL** Use this clause to delete all check constraints for the column.

  - ○ **DEFAULT clause** Use this clause to change the default value for the column.

  - ○ **DEFAULT NULL clause** Use this clause to remove the default value for the column.

  - ○ **[CONSTRAINT constraint-name] CHECK { NULL | (condition) } clause** Use this clause to add a CHECK constraint on the column.

  - ○ **[NOT] COMPRESSED clause** Use this clause to change whether the column is compressed.

  - ○ **INLINE and PREFIX clauses** Use the INLINE and PREFIX clauses with columns that contain BLOBs to specify how much of a BLOB, in bytes, to keep within a row. For more information about how to set the INLINE and PREFIX values, see the corresponding sections for the INLINE and PREFIX clauses in "CREATE TABLE statement" on page 523.

  - **INDEX and NO INDEX clauses** Use this clause to specify whether to build indexes on large BLOBs in this column. For more information about how to use this clause, see the corresponding section for the [NO] INDEX clause in the "CREATE TABLE statement" on page 523.
  - **SET COMPUTE clause** Use this clause to change the expression associated with the computed column. The values in the column are recalculated when the statement is executed, and the statement fails if the new expression is invalid. For more information about what is allowed for the COMPUTE expression, see "CREATE TABLE statement" on page 523.
  - **ALTER CONSTRAINT constraint-name CHECK clause** Use this clause to alter a named check constraint for the table.

- **Dropping clauses** The following section explains the DROP clauses:
  - **DROP DEFAULT** Drops the default value set for the table or specified column. Existing values do not change.
  - **DROP COMPUTE** Removes the COMPUTE attribute for the specified column. This statement does not change any existing values in the table.
  - **DROP CHECK** Drops all CHECK constraints for the table or specified column. DELETE CHECK is also accepted.
  - **DROP CONSTRAINT constraint-name** Drops the named constraint for the table or specified column. DELETE CONSTRAINT is also accepted.
  - **DROP column-name** Drops the specified column from the table. DELETE *column-name* is also accepted. If the column is contained in any index, unique constraint, foreign key, or primary key, then the index, constraint, or key must be deleted before the column can be deleted. This does not delete CHECK constraints that refer to the column.
  - **DROP UNIQUE (column-name ...)** Drop the unique constraints on the specified column(s). Any foreign keys referencing this unique constraint are also deleted. DELETE UNIQUE (*column-name* ...) is also accepted.
  - **DROP FOREIGN KEY fkey-name** Drop the specified foreign key. DELETE FOREIGN KEY *fkey-name* is also accepted.
  - **DROP PRIMARY KEY** Drop the primary key. All foreign keys referencing the primary key for this table are also deleted. DELETE PRIMARY KEY is also accepted.

- **Renaming clauses** The following section explains the clauses used for renaming parts of a column or table definition:
  - **RENAME new-table-name** Change the name of the table to *new-table-name*. Note that any applications using the old table name must be modified, as necessary. After the renaming operation succeeds, foreign keys with ON UPDATE or ON DELETE actions must be dropped and re-created, as the system-created triggers used to implement these actions will continue to refer to the old name.
  - **RENAME column-name TO new-column-name** Change the name of the column to the *new-column-name*. Note that any applications using the old column name will need to be modified, as necessary. After the renaming operation succeeds, foreign keys with ON UPDATE or ON DELETE

actions must be dropped and re-created, as the system-created triggers used to implement these actions will continue to refer to the old name.

- **RENAME CONSTRAINT constraint-name TO new-constraint-name** Change the name of the constraint to the *new-constraint-name*.

- **table-alteration clauses** Use this clause to alter the following table attributes.
  - **PCTFREE DEFAULT** Use this clause to change the percent free setting for the table to the default (200 bytes for a 4 KB, and up, page size).
  - **REPLICATE { ON | OFF }** Use this clause to change whether the table is included during replication. When a table has REPLICATE ON, all changes to the table are sent to Replication Server for replication. The replication definitions in Replication Server are used to decide which table changes are sent to other sites.
  - **[NOT] ENCRYPTED** Use this clause to change whether the table is encrypted. To encrypt a table, table encryption must already be enabled on the database. The table is encrypted using the encryption key and algorithm specified at database creation time. See "Enabling table encryption in the database" [*SQL Anywhere Server - Database Administration*]. After encrypting a table, any data for that table that was in temporary files or the transaction log prior to encryption still exists in unencrypted form. To address this, restart the database to remove the temporary files. Run the Backup utility (dbbackup) with the -o option, or use the BACKUP statement, to back up the transaction log and start a new one. See "Backup utility (dbbackup)" [*SQL Anywhere Server - Database Administration*] or "BACKUP statement" on page 407.

    When table encryption is enabled, table pages for the encrypted table, associated index pages, temporary file pages, and transaction log pages containing transactions on encrypted tables are encrypted.

- **DISABLE VIEW DEPENDENCIES clause** Use this clause to disable dependent regular views. Dependent materialized views are not disabled; you must disable each dependent materialized view by executing an ALTER MATERIALIZED VIEW ... DISABLE statement. See "ALTER MATERIALIZED VIEW statement" on page 373.

Remarks

The ALTER TABLE statement changes table attributes (column definitions, constraints, and so on) in an existing table.

The database server keeps track of object dependencies in the database. Alterations to the schema of a table may impact dependent views. Also, if there are materialized views that are dependent on the table you are attempting to alter, you must first disable them using the ALTER MATERIALIZED VIEW ... DISABLE statement. For information about view dependencies, see "View dependencies" [*SQL Anywhere Server - SQL Usage*].

You cannot use ALTER TABLE on a local temporary table.

ALTER TABLE is prevented whenever the statement affects a table that is currently being used by another connection. ALTER TABLE can be time-consuming, and the database server does not process other requests referencing the table while the statement is being processed.

For more information about using the CLUSTERED option, see "Using clustered indexes" [*SQL Anywhere Server - SQL Usage*].

If you alter a column that a text index defined as IMMEDIATE REFRESH is built on, the text index is immediately rebuilt. If the text index is defined as AUTO REFRESH or MANUAL REFRESH, the text index is rebuilt the next time it is refreshed.

When you execute an ALTER TABLE statement, the database server attempts to restore column permissions on dependent views that are automatically recompiled. Permissions on columns that no longer exist in the recompiled views are lost.

Permissions

Must be one of the following:

- The owner of the table.
- A user with DBA authority.
- A user who has been granted ALTER permission on the table.

ALTER TABLE requires exclusive access to the table.

Global temporary tables cannot be altered unless all users that have referenced the temporary table have disconnected.

Cannot be used within a snapshot transaction. See "Snapshot isolation" [*SQL Anywhere Server - SQL Usage*].

Side effects

Automatic commit.

A checkpoint is carried out at the beginning of the ALTER TABLE operation, and further checkpoints are suspended until the ALTER operation completes.

Once you alter a column or table, any stored procedures, views, or other items that refer to the altered column may no longer work.

If you change the declared length or type of a column, or drop a column, the statistics for that column are dropped. For information about how to generate new statistics, see "Updating column statistics to improve optimizer performance" [*SQL Anywhere Server - SQL Usage*].

See also

- "CREATE TABLE statement" on page 523
- "DROP TABLE statement" on page 588
- "SQL data types" on page 77
- "Altering tables" [*SQL Anywhere Server - SQL Usage*]
- "Special values" on page 58
- "Using table and column constraints" [*SQL Anywhere Server - SQL Usage*]
- "allow_nulls_by_default option [compatibility]" [*SQL Anywhere Server - Database Administration*]
- "Enabling table encryption in the database" [*SQL Anywhere Server - Database Administration*]

Standards and compatibility

- **SQL/2003** ADD COLUMN is a core feature. Other clauses are vendor extensions or implementation of specific, named extensions to SQL/2003.

Example

The following example adds a new timestamp column, TimeStamp, to the Customers table.

```
ALTER TABLE Customers
   ADD TimeStamp AS TIMESTAMP DEFAULT TIMESTAMP;
```

The following example drops the new timestamp column, TimeStamp that you added in the previous example.

```
ALTER TABLE Customers
DROP TimeStamp;
```

The Street column in the Customers table can currently hold up to 35 characters. To allow it to hold up to 50 characters, execute the following:

```
ALTER TABLE Customers
ALTER Street CHAR(50);
```

The following example adds a column to the Customers table, assigning each customer a sales contact.

```
ALTER TABLE Customers
ADD SalesContact INTEGER
REFERENCES Employees ( EmployeeID )
ON UPDATE CASCADE
ON DELETE SET NULL;
```

This foreign key is constructed with cascading updates and is set to NULL on deletes. If an employee has their employee ID changed, the column is updated to reflect this change. If an employee leaves the company and has their employee ID deleted, the column is set to NULL.

The following example creates a foreign key, FK_SalesRepresentative_EmployeeID2, on the SalesOrders.SalesRepresentative column, linking it to Employees.EmployeeID:

```
ALTER TABLE GROUPO.SalesOrders
   ADD CONSTRAINT FK_SalesRepresentative_EmployeeID2
   FOREIGN KEY ( SalesRepresentative )
   REFERENCES GROUPO.Employees (EmployeeID);
```

ALTER TEXT CONFIGURATION statement

Alters a text configuration object.

Syntax

ALTER TEXT CONFIGURATION [*owner.*]*config-name*
STOPLIST *stoplist*
| **DROP STOPLIST**
| { **MINIMUM** | **MAXIMUM** } **TERM LENGTH** *integer*
| **TERM BREAKER** { **GENERIC** | **NGRAM** }

stoplist : *string-expression*

Parameters

- **STOPLIST clause** Use this clause to create or replace the list of terms to ignore when building a text index. Terms specified in this list are also ignored in a query. Separate stoplist terms with spaces. For example, `STOPLIST 'because about therefore only'`. Stoplist terms cannot contain whitespace.

Samples of stoplists for different languages are located in the *samples-dir\SQLAnywhere\SQL* subdirectory. For the location of *samples-dir*, see "Samples directory" [*SQL Anywhere Server - Database Administration*].

Stoplist terms should not contain non-alphanumeric characters.

Carefully consider whether you want to put terms in your stoplist. For more information, see "Text configuration object settings" [*SQL Anywhere Server - SQL Usage*]

- **DROP STOPLIST clause** Use this clause to drop the stoplist for a text configuration object.
- **MINIMUM TERM LENGTH clause** The value specified in the MINIMUM TERM LENGTH clause is ignored when using NGRAM text indexes.

  The minimum length, in characters, of a term to include in the text index. Terms that are shorter than this setting are ignored when building or refreshing the text index. The value of this option must be greater than 0. If you set this option to be higher than MAXIMUM TERM LENGTH, the value of MAXIMUM TERM LENGTH is automatically adjusted to be the same as the new MINIMUM TERM LENGTH value.
- **MAXIMUM TERM LENGTH clause** With NGRAM text indexes the value specified in the MAXIMUM TERM LENGTH clause is N.

  With GENERIC text indexes, the maximum length, in characters, of a term to include in the text index. Terms that are longer than this setting are ignored when building or refreshing the text index. The value of MAXIMUM TERM LENGTH must be less than or equal to 60. If you set this option to be lower than MINIMUM TERM LENGTH, the value of MINIMUM TERM LENGTH is automatically adjusted to be the same as the new MAXIMUM TERM LENGTH value.
- **TERM BREAKER clause** The name of the algorithm to use for separating column values into terms. The choices are GENERIC (the default) or NGRAM. The GENERIC algorithm treats any string of one or more alphanumerics, separated by non-alphanumerics, as a term. The NGRAM algorithm breaks strings into n-grams. An n-gram is an *n*-character substring of a larger string. The NGRAM term breaker is required for fuzzy (approximate) matching, or for documents that do not use whitespace or non-alphanumeric characters to separate terms. For more information about these algorithms and how to choose between them, see "Text configuration object settings" [*SQL Anywhere Server - SQL Usage*].

Remarks

Before changing the term length settings, read about the impact of various settings on what gets indexed and how query terms are interpreted. See "Text configuration object settings" [*SQL Anywhere Server - SQL Usage*], and "Example text configuration objects" [*SQL Anywhere Server - SQL Usage*].

Text indexes are dependent on a text configuration object. You must truncate dependent text indexes before altering the underlying text configuration object. If the dependent text indexes are of the immediate refresh type, and cannot be truncated, you must drop the indexes before you can alter the text configuration object.

To determine the text indexes that refer to a text configuration object, see "View text indexes in the database" [*SQL Anywhere Server - SQL Usage*].

To view the settings for text configuration objects, query the SYSTEXTCONFIG system view. See "SYSTEXTCONFIG system view" on page 1034.

Permissions

Must be the owner of the text configuration object, or have DBA authority.

Side effects

Automatic commit

See also

- "Text configuration object settings" [*SQL Anywhere Server - SQL Usage*]
- "Alter a text configuration object" [*SQL Anywhere Server - SQL Usage*]
- "Tutorial: Performing a full text search on a GENERIC text index" [*SQL Anywhere Server - SQL Usage*]
- "Tutorial: Performing a fuzzy full text search" [*SQL Anywhere Server - SQL Usage*]
- "CREATE TEXT CONFIGURATION statement" on page 535
- "DROP TEXT CONFIGURATION statement" on page 589
- "sa_char_terms system procedure" on page 843
- "sa_nchar_terms system procedure" on page 914
- "sa_refresh_text_indexes system procedure" on page 926
- "sa_text_index_stats system procedure" on page 957

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statements create a text configuration object, maxTerm16, and then change the maximum term length to 16:

```
CREATE TEXT CONFIGURATION maxTerm16 FROM default_char;
ALTER TEXT CONFIGURATION maxTerm16
   MAXIMUM TERM LENGTH 16;
```

The following statement adds a stoplist to the maxTerm16 configuration object:

```
ALTER TEXT CONFIGURATION maxTerm16
   STOPLIST 'because about therefore only';
```

ALTER TEXT INDEX statement

Alters the definition of a text index.

Syntax

ALTER TEXT INDEX [*owner.*]*text-index-name*
ON [*owner.*]*table-name*
alter-clause

alter-clause :
rename-object
| *refresh-alteration*

rename-object :
RENAME { **AS** | **TO** } *new-name*

refresh-alteration :
{ **MANUAL REFRESH**
| **AUTO REFRESH** [**EVERY** *integer* { **MINUTES** | **HOURS** }] }

Parameters

- **RENAME clause** Use the RENAME clause to rename the text index.
- **REFRESH clause** Specify the REFRESH clause to set the refresh type for the text index. For more information about the options for this clause, see "CREATE TEXT INDEX statement" on page 536.

Remarks

Once a text index is created, you cannot change it to, or from, IMMEDIATE REFRESH. If either of these changes is required, you must drop and recreate the text index.

To view text indexes and the text configuration objects they refer to, see "View text indexes in the database" [*SQL Anywhere Server - SQL Usage*].

Permissions

Must be the owner of the underlying table, or have DBA authority, or have REFERENCES permission.

This statement cannot be executed when there are cursors opened with the WITH HOLD clause that use either statement or transaction snapshots. See "Snapshot isolation" [*SQL Anywhere Server - SQL Usage*].

Side effects

Automatic commit

See also

- "Full text searching" [*SQL Anywhere Server - SQL Usage*]
- "Altering text indexes overview" [*SQL Anywhere Server - SQL Usage*]
- "Tutorial: Performing a full text search on a GENERIC text index" [*SQL Anywhere Server - SQL Usage*]
- "Tutorial: Performing a fuzzy full text search" [*SQL Anywhere Server - SQL Usage*]
- "CREATE TEXT INDEX statement" on page 536
- "ALTER TEXT INDEX statement" on page 399
- "DROP TEXT INDEX statement" on page 590
- "REFRESH TEXT INDEX statement" on page 706
- "TRUNCATE TEXT INDEX statement" on page 770

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The first statement creates a text index, txt_index_manual, defining it as MANUAL REFRESH. The second statement alters the text index to refresh automatically every day. The third statement renames the text index to txt_index_daily.

```
CREATE TEXT INDEX txt_index_manual ON MarketingInformation ( Description )
   MANUAL REFRESH;
ALTER TEXT INDEX txt_index_manual ON MarketingInformation
   AUTO REFRESH EVERY 24 HOURS;
ALTER TEXT INDEX txt_index_manual ON MarketingInformation
   RENAME AS txt_index_daily;
```

ALTER TRIGGER statement

Use this statement to replace a trigger definition with a modified version.

You must include the entire new trigger definition in the ALTER TRIGGER statement.

Syntax 1

ALTER TRIGGER *trigger-name trigger-definition*

trigger-definition : CREATE TRIGGER syntax

Syntax 2

ALTER TRIGGER *trigger-name* **ON** [*owner.*] *table-name* **SET HIDDEN**

Remarks

Syntax 1 The ALTER TRIGGER statement is identical in syntax to the CREATE TRIGGER statement except for the first word. For information about *trigger-definition*, see "CREATE TRIGGER statement" on page 538 and "CREATE TRIGGER statement [T-SQL]" on page 544.

Either the Transact-SQL or Watcom-SQL form of the CREATE TRIGGER syntax can be used.

Syntax 2 You can use SET HIDDEN to obfuscate the definition of the associated trigger and cause it to become unreadable. The trigger can be unloaded and reloaded into other databases. If SET HIDDEN is used, debugging using the debugger will not show the trigger definition, nor will it be available through procedure profiling.

Note
The SET HIDDEN operation is irreversible.

Permissions

Must be the owner of the table on which the trigger is defined, or be user DBA, or have ALTER permissions on the table and have RESOURCE authority.

Side effects

Automatic commit.

See also

- "CREATE TRIGGER statement" on page 538
- "CREATE TRIGGER statement [T-SQL]" on page 544
- "DROP TRIGGER statement" on page 591
- "Hiding the contents of procedures, functions, triggers and views" [*SQL Anywhere Server - SQL Usage*]

Standards and compatibility

- **SQL/2003** Vendor extension.

ALTER USER statement

Use this statement to alter user settings.

Syntax 1

ALTER USER *user-name* [**IDENTIFIED BY** *password*]
 [**LOGIN POLICY** *policy-name*]
 [**FORCE PASSWORD CHANGE** { **ON** | **OFF** }]

Syntax 2

ALTER USER *user-name*
[**RESET LOGIN POLICY**]

Parameters

- **user-name** The name of the user.
- **IDENTIFIED BY clause** The password for the user.
- **policy-name** The name of the login policy to assign the user. No change is made if the LOGIN POLICY clause is not specified.
- **FORCE PASSWORD CHANGE clause** Controls whether the user must specify a new password when they log in. This setting overrides the password_expiry_on_next_login option setting in their policy.
- **RESET LOGIN POLICY clause** Reverts the settings of a user's login policy to the original values. When you reset a login policy, a user can access an account that has been locked for exceeding a login policy option limit such as max_failed_login_attempts or max_days_since_login.

Remarks

User IDs and passwords cannot:

- begin with white space, single quotes, or double quotes
- end with white space
- contain semicolons

A password can be either a valid identifier, or a string (maximum 255 bytes) placed in single quotes. Passwords are case sensitive. It is recommended that the password be composed of 7-bit ASCII characters, as other characters may not work correctly if the database server cannot convert them from the client's character set to UTF-8.

The verify_password_function option can be used to specify a function to implement password rules (for example, passwords must include at least one digit). If a password verification function is used, you cannot specify more than one user ID and password in the GRANT CONNECT statement. See "verify_password_function option [database]" [*SQL Anywhere Server - Database Administration*].

If you set the password_expiry_on_next_login value to ON, the user's password expires immediately when they next login even if they are assigned to the same policy. You can use the ALTER USER and LOGIN POLICY clauses to force a user to change their password when they next login.

Permissions

Any user can change their own password. All other changes require DBA authority.

Side effects

None.

See also

- "ALTER LOGIN POLICY statement" on page 372
- "COMMENT statement" on page 425
- "CREATE LOGIN POLICY statement" on page 475
- "CREATE USER statement" on page 546
- "DROP LOGIN POLICY statement" on page 577
- "DROP USER statement" on page 591
- "Managing login policies overview" [*SQL Anywhere Server - Database Administration*]
- "Assigning a login policy to an existing user" [*SQL Anywhere Server - Database Administration*]
- "GRANT statement" on page 627

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following alters a user named SQLTester. The password is set to "welcome". The SQLTester user is assigned to the Test1 login policy and the password does not expire on the next login.

```
ALTER USER SQLTester IDENTIFIED BY welcome
LOGIN POLICY Test1
FORCE PASSWORD CHANGE off;
```

ALTER VIEW statement

Use this statement to replace a view definition with a modified version.

Syntax 1

ALTER VIEW
[*owner*.]*view-name* [**(** *column-name*, ... **)**] **AS** *select-statement*
[**WITH CHECK OPTION**]

Syntax 2

ALTER VIEW
[*owner*.]*view-name* { **SET HIDDEN** | **RECOMPILE** | **DISABLE** | **ENABLE** }

Parameters

- **AS clause** The purpose and syntax of this clause is identical to that of the CREATE VIEW statement. See "CREATE VIEW statement" on page 547.
- **WITH CHECK OPTION clause** The purpose and syntax of this clause is identical to that of the CREATE VIEW statement. See "CREATE VIEW statement" on page 547.
- **SET HIDDEN clause** Use the SET HIDDEN clause to obfuscate the definition of the view and cause the view to become hidden from view, for example in Sybase Central. Explicit references to the view will still work, however.

Note
The SET HIDDEN operation is irreversible.

- **RECOMPILE clause** Use the RECOMPILE clause to re-create the column definitions for the view. This clause is identical in functionality to the ENABLE clause, except that you may decide to use it on a view that is not disabled. When a view is recompiled, the database server restores the column permissions based on the column names specified in the new view definition. The existing permissions are lost when a column no longer exists after the recompilation.
- **DISABLE clause** Use the DISABLE clause to disable the view from use by the database server.
- **ENABLE clause** Use the ENABLE clause to enable a disabled view. Enabling the view causes the database server to re-create the column definitions for the view. Before you enable a view, you must enable any views upon which it depends.

Remarks

If you alter a view owned by another user, you must qualify the name by including the owner (for example, GROUPO.ViewSalesOrders). If you don't qualify the name, the database server looks for a view with that name owned by you and alters it. If there isn't one, it returns an error.

When you alter a view, existing permissions on the view are maintained, and do not have to be reassigned. Instead of using the ALTER VIEW statement, you could also drop the view and recreate it using the DROP VIEW and CREATE VIEW, respectively. However, if you do so, permissions on the view need to be reassigned.

After completing the view alteration using Syntax 1, the database server recompiles the view. Depending on the type of change you made, if there are dependent views, the database server will attempt to recompile them as well. If you have made a change that impacts a dependent view, you may need to alter the definition for the dependent view as well. For more information about view alterations and how they impact view dependencies, see "View dependencies" [*SQL Anywhere Server - SQL Usage*].

Caution
If the SELECT statement defining the view contained an asterisk (*), the number of the columns in the view may change if columns have been added or deleted from the underlying tables. The names and data types of the view columns may also change.

Syntax 1 This syntax is used to alter the structure of the view. Unlike altering tables where your change may be limited to individual columns, altering the structure of a view requires you to replace the entire view definition with a new definition, much as you would for creating the view. For a description of the parameters used to define the structure of a view, see "CREATE VIEW statement" on page 547.

Syntax 2 This syntax is used to change attributes for the view, such as whether the view definition is hidden.

When you use SET HIDDEN, the view can be unloaded and reloaded into other databases. If SET HIDDEN is used, debugging using the debugger will not show the view definition, nor will it be available through procedure profiling. If you need to change the definition of a hidden view, you must drop the view and create it again using the CREATE VIEW statement.

When you use the DISABLE clause, the view is no longer available for use by the database server for answering queries. Disabling a view is similar to dropping it, except that the view definition remains in the database. Disabling a view also disables any dependent views. Therefore, the DISABLE clause requires exclusive access not only to the view being disabled, but also any dependent views, since they are disabled too.

Permissions

Must be owner of the view or have DBA authority.

Side effects

Automatic commit.

All procedures and triggers are unloaded from memory, so that any procedure or trigger that references the view reflects the new view definition. The unloading and loading of procedures and triggers can have a performance impact if you are regularly altering views.

See also

- "CREATE VIEW statement" on page 547
- "DROP VIEW statement" on page 593
- "Hiding the contents of procedures, functions, triggers and views" [*SQL Anywhere Server - SQL Usage*]
- "View dependencies" [*SQL Anywhere Server - SQL Usage*]
- "CREATE MATERIALIZED VIEW statement" on page 477
- "ALTER MATERIALIZED VIEW statement" on page 373

Standards and compatibility

- **SQL/2003** Vendor extension.

ATTACH TRACING statement

Use this statement to start a diagnostic tracing session. That is, to start sending diagnostic information to the diagnostic tables.

Syntax

ATTACH TRACING TO { **LOCAL DATABASE** | *connect-string* }
[**LIMIT** { *size* | *history* }]

connect-string : the connection string for the database

size : **SIZE** *nnn* { **MB** | **GB** }

history : **HISTORY** *nnn* { **MINUTES** | **HOURS** | **DAYS** }

nnn : *integer*

Parameters

- **connect-string** The connection string required to connect to the database receiving the tracing information. This parameter is only required when the database being profiled is different from the database receiving the data.

  The following parameters are allowed in *connect-string*: DBF, DBKEY, DBN, ENG, LINKS, PWD, UID.

  Specify DBF relative to the database server location. Otherwise, the database server attempts to start a database with that name on the current server computer.

  An error is returned if you specify the DBF parameter with the LINKS or ENG connection parameters.

- **LIMIT clause** The volume limit of data stored in the tracing database, either by size, or by length of time.

Remarks

The ATTACH TRACING statement is primarily used by the tracing wizards in Sybase Central. However, you can also run it manually. You must run if from the database you want to profile.

The ATTACH TRACING statement is used to start a tracing session for the database you want to profile. You can only use it once a tracing level has been set. You can set the tracing level using Sybase Central, or using the sa_set_tracing_level system procedure.

Once a session is started, tracing information is generated according to the tracing levels set in the sa_diagnostic_tracing_level table. You can send the tracing data to tracing tables within the same database that is being profiled, by specifying LOCAL DATABASE. Alternatively, you can send the tracing data to a separate tracing database by specifying a connection string (*connect-string*) to that database. The tracing database must already exist, and you must have permissions to access it.

You can limit the amount of tracing data to store using the LIMIT SIZE or LIMIT HISTORY clauses. Use the LIMIT SIZE clause when you want to limit the volume of tracing data to a certain size, as measured in megabytes or gigabytes. Use the LIMIT HISTORY clause to limit the volume of tracing data to a period of time, as measured in minutes, hours, or days. For example, `HISTORY 8 DAYS` limits the amount of tracing data stored in the tracing database to 8 days' worth.

To start a tracing session, TCP/IP must be running on the database server(s) on which the tracing database and production database are running. See "Using the TCP/IP protocol" [*SQL Anywhere Server - Database Administration*].

Packets that contain potentially sensitive data are visible on the network interface, even when tracing to a local database. For security purposes, you can specify encryption in the connection string.

To see the current tracing levels set for a database, look in the sa_diagnostic_tracing_level table. See "sa_diagnostic_tracing_level table" on page 824.

To see where tracing data is being sent to, examine the SendingTracingTo database property. See "Database properties" [*SQL Anywhere Server - Database Administration*].

Permissions

Must be connected to the database being profiled and must have DBA or PROFILE authority.

Side effects

None.

See also

- "DETACH TRACING statement" on page 568
- "REFRESH TRACING LEVEL statement" on page 708
- "Advanced application profiling using diagnostic tracing" [*SQL Anywhere Server - SQL Usage*]
- "sa_set_tracing_level system procedure" on page 949

Standards and compatibility

- **SQL/2003** Vendor extension.

Examples

The following example sets the tracing level to 1 using the sa_set_tracing_level system procedure. Then it starts a tracing session. Tracing data generated for the local database will be sent to the mytracingdb tracing database on another computer, as shown by the specified connection string. A maximum of two hours of tracing data will be maintained during the tracing session. Note that the ATTACH TRACING statement example is all on one line.

```
CALL sa_set_tracing_level( 1 );
ATTACH TRACING TO
'uid=DBA;pwd=sql;eng=remotedbsrv11;dbn=mytracingdb;links=tcpip'
 LIMIT HISTORY 2 HOURS;
```

BACKUP statement

Use this statement to back up a database and transaction log.

Syntax 1 (image backup)

BACKUP DATABASE
DIRECTORY *backup-directory*
[**WAIT BEFORE START**]
[**WAIT AFTER END**]
[**DBFILE ONLY**]
[**TRANSACTION LOG ONLY**]
[**TRANSACTION LOG RENAME** [**MATCH**]]
[**TRANSACTION LOG TRUNCATE**]
[**ON EXISTING ERROR**]
[**WITH COMMENT** *comment string*]
[**HISTORY** { **ON** | **OFF** }]
[**AUTO TUNE WRITERS** { **ON** | **OFF** }]
[**WITH CHECKPOINT LOG** { **AUTO** | **COPY** | **NO COPY** | **RECOVER** }]

backup-directory : { *string* | *variable* }

Syntax 2 (archive backup)

BACKUP DATABASE TO *archive-root*
[**WAIT BEFORE START**]
[**WAIT AFTER END**]
[**DBFILE ONLY**]
[**TRANSACTION LOG ONLY**]
[**TRANSACTION LOG RENAME** [**MATCH**]]
[**TRANSACTION LOG TRUNCATE**]
[**ATTENDED** { **ON** | **OFF** }]
[**WITH COMMENT** *comment string*]
[**HISTORY** { **ON** | **OFF** }]
[**WITH CHECKPOINT LOG** [**NO**] **COPY**]
[**MAX WRITE** { *number-of-writers* | **AUTO** }]

archive-root : { *string* | *variable* }

comment-string : *string*

number-of-writers : *integer*

Parameters

- **DIRECTORY clause** The target location on disk for the backup files, relative to the database server's current directory at startup. If the directory does not exist, it is created. Specifying an empty string as a directory allows you to rename or truncate the log without first making a copy of it. Do not use this clause if you are using database mirroring. See "Database mirroring and transaction log files" [*SQL Anywhere Server - Database Administration*].

- **WAIT BEFORE START clause** Use this clause to ensure that the backup copy of the database does not contain any information required for recovery. In particular, it ensures that the rollback log for each connection is empty.

  If a backup is carried out using this clause, you can start the backup copy of the database in read-only mode and validate it. By enabling validation of the backup database, you can avoid making an additional copy of the database.

- **WAIT AFTER END clause** Use this clause if the transaction log is being renamed or truncated. It ensures that all transactions are completed before the log is renamed or truncated. If this clause is used, the backup must wait for other connections to commit or rollback any open transactions before finishing.
- **DBFILE ONLY clause** Use this clause to cause backup copies of the main database file and any associated dbspaces to be made. The transaction log is not copied. You cannot use the DBFILE ONLY clause with the TRANSACTION LOG RENAME or TRANSACTION LOG TRUNCATE clauses.
- **TRANSACTION LOG ONLY clause** Use this clause to cause a backup copy of the transaction log to be made. No other database files are copied.
- **TRANSACTION LOG RENAME [MATCH] clause** This clause causes the database server to rename the current transaction log at the completion of the backup. If the MATCH keyword is omitted, the backup copy of the log will have the same name as the current transaction log for the database. If you supply the MATCH keyword, the backup copy of the transaction log is given a name of the form *YYMMDDnn.log*, to match the renamed copy of the current transaction log. Using the MATCH keyword enables the same statement to be executed several times without writing over old data.
- **TRANSACTION LOG TRUNCATE clause** If this clause is used, the current transaction log is truncated and restarted at the completion of the backup. Do not use this clause if you are using database mirroring. See "Database mirroring and transaction log files" [*SQL Anywhere Server - Database Administration*].
- **archive-root clause** The file name or tape drive device name for the archive file.

  To back up to tape, you must specify the device name of the tape drive. For example, on NetWare the first tape drive is *\\.\tape0*. The number automatically appended to the end of the archive file name is incremented each time you execute an archive backup.

  The backslash (\ ) is an escape character in SQL strings, so each backslash must be doubled. For more information about escape characters and strings, see "Strings" on page 9.
- **ON EXISTING ERROR clause** This clause applies only to image backups. By default, existing files are overwritten when you execute a BACKUP DATABASE statement. If this clause is used, an error occurs if any of the files to be created by the backup already exist.
- **ATTENDED clause** The clause applies only when backing up to a tape device. ATTENDED ON (the default) indicates that someone is available to monitor the status of the tape drive and to place a new tape in the drive when needed. A message is sent to the application that issued the BACKUP DATABASE statement if the tape drive requires intervention. The database server then waits for the drive to become ready. This may happen, for example, when a new tape is required.

  If ATTENDED OFF is specified and a new tape is required or the drive is not ready, no message is sent and an error is given.
- **WITH COMMENT clause** This clause records a comment in the backup history file. For archive backups, the comment is also recorded in the archive file.

- **HISTORY clause** By default, each backup operation appends a line to the *backup.syb* file. You can prevent updates to the *backup.syb* file by specifying HISTORY OFF. You may want to prevent the file from being updated if any of the following conditions apply:

  - your backups occur frequently
  - there is no procedure to periodically archive or delete the *backup.syb* file
  - disk space is very limited

- **AUTO TUNE WRITERS clause** When the backup starts, one thread is dedicated to writing the backup files to the backup directory. However, if the backup directory is on a device that can handle an increased writer load (such as a RAID array), then overall backup performance can be improved by increasing the number of threads acting as writers. If this clause is ON (the default), the database server periodically examines the read and write performance from all the devices taking part in the backup. If the overall backup speed can be improved by creating another writer, then the database server creates another writer.

- **WITH CHECKPOINT LOG clause** This clause specifies how the backup processes the database files before writing them to the destination directory. The choice of whether to apply pre-images during a backup, or copy the checkpoint log as part of the backup, has performance implications. The default setting is AUTO for image backups and COPY for archive backups.

- **COPY clause** This option cannot be used with the WAIT BEFORE START clause of the BACKUP statement.

  When you specify COPY, the backup reads the database files without applying any modified pages. The entire checkpoint log and the system dbspace are copied to the backup directory. The next time the database server is started, the database server automatically recovers the database to the state it was in as of the checkpoint at the time the backup started.

  Because pages do not have to be written to the temporary file, using this option can provide better backup performance, and reduce internal server contention for other connections that are operating during a backup. However, since the checkpoint log contains original images of modified pages, it will grow in the presence of database updates. With copy specified, the backed-up copy of the database files may be larger than the database files at the time the backup started. The COPY option should be used if disk space in the destination directory is not an issue.

- **NO COPY clause** When you specify NO COPY, the checkpoint log is not copied as part of the backup. This option causes modified pages to be saved in the temporary file so that they can be applied to the backup as it progresses. The backup copies of the database files will be the same size as the database when the backup operation commenced.

  This option results in smaller backed up database files, but the backup may proceed more slowly, and possibly decrease performance of other operations in the database server. It is useful in situations where space on the destination drive is limited.

- **RECOVER clause** When you specify RECOVER, the database server copies the checkpoint log (as with the COPY option), but applies the checkpoint log to the database when the backup is complete. This restores the backed up database files to the same state (and size) that they were in at the start of the backup operation. This option is useful if space on the backup drive is limited (it requires the same amount of space as the COPY option for backing up the checkpoint log, but the resulting file size is smaller).

- **AUTO clause** When you specify AUTO, the database server checks the amount of available disk space on the volume hosting the backup directory. If there is at least twice as much disk space available as the size of the database at the start of the backup, then this option behaves as if copy were specified. Otherwise, it behaves as NO COPY. AUTO is the default behavior.
- **MAX WRITE clause** For archive backups, by default one thread is dedicated to writing the backup files. If the backup directory is on a device that can handle an increased writer load (such as a RAID array), then overall backup performance can be improved by increasing the number of threads acting as writers.

  If AUTO is specified, one output stream is created for each reader thread. The value *n* specifies the maximum number of output streams that can be created, up to the number of reader threads. The default value for this clause is 1. If you are backing up to tape, only one writer can be used.

  The first stream, stream 0, produces files named *myarchive.X*, where *X* is a number that starts at 1 and continues incrementing to the number of files required. All of the other streams produce files named *myarchive.Y.Z*, where *Y* is the stream number (starting at 1), and *Z* is a number that starts at 1 and continues incrementing to the number of files required.

Remarks

The BACKUP statement performs a server-side backup. To perform a client-side backup, use the dbbackup utility. See "Backup utility (dbbackup)" [*SQL Anywhere Server - Database Administration*].

Each backup operation, whether image or archive, updates a history file called *backup.syb*. This file records the BACKUP and RESTORE operations that have been performed on a database server. For information about how the location of the *backup.syb* file is determined, see "SALOGDIR environment variable" [*SQL Anywhere Server - Database Administration*].

To create a backup that can be started on a read-only server without having to go through recovery, you must use both the WAIT BEFORE START and WITH CHECKPOINT LOG NO COPY clauses. The WAIT BEFORE START clause ensures that the rollback log is empty, and the WITH CHECKPOINT LOG NO COPY clause ensures that the checkpoint log is empty. If either of these files is missing, then recovery is required.

Syntax 1 (image backup) An image backup creates copies of each of the database files, in the same way as the Backup utility (dbbackup). By default, the Backup utility makes the backup on the client computer, but you can specify the -s option to create the backup on the database server when using the Backup utility. In the case of the BACKUP DATABASE statement, however, the backup can only be made on the database server.

Optionally, only the database file(s) or transaction log can be saved. The log may also be renamed or truncated after the backup has completed.

Alternatively, you can specify an empty string as a directory to rename or truncate the log without copying it first. This is useful in a replication environment where space is a concern. You can use this feature with an event handler on transaction log size to rename the log when it reaches a given size, and with the delete_old_logs option to delete the log when it is no longer needed.

To restore from an image backup, copy the saved files back to their original locations and reapply the transaction logs as described in "Recovering a database with multiple transaction logs" [*SQL Anywhere Server - Database Administration*].

Syntax 2 (archive backup) An archive backup creates a single file holding all the required backup information. The destination can be either a file name or a tape drive device name.

There can be only one backup on a given tape. The tape is ejected at the end of the backup.

Only one archive per tape is allowed, but a single archive can span multiple tapes. To restore a database from an archive backup, use the RESTORE DATABASE statement.

If a RESTORE DATABASE statement references an archive file containing only a transaction log, the statement must specify a file name for the location of the restored database file, even if that file does not exist. For example, to restore from an archive containing only a log to the directory *C:\MYNEWDB*, the RESTORE DATABASE statement is:

```
RESTORE DATABASE 'c:\mynewdb\my.db' FROM archive-root
```

Caution
Backup copies of the database and transaction log must not be changed in any way. If there were no transactions in progress during the backup, or if you specified BACKUP DATABASE WITH CHECKPOINT LOG RECOVER or WITH CHECKPOINT LOG NO COPY, you can check the validity of the backup database using read-only mode or by validating a copy of the backup database.

However, if transactions were in progress, or if you specified BACKUP DATABASE WITH CHECKPOINT LOG COPY, the database server must perform recovery on the database when you start it. Recovery modifies the backup copy, which is not desirable.

Permissions

Must have DBA, REMOTE DBA, or BACKUP authority.

Side effects

Causes a checkpoint.

See also

- "Backup utility (dbbackup)" [*SQL Anywhere Server - Database Administration*]
- "Image backups" [*SQL Anywhere Server - Database Administration*]
- "RESTORE DATABASE statement" on page 714
- "Backup and data recovery" [*SQL Anywhere Server - Database Administration*]
- "EXECUTE IMMEDIATE statement [SP]" on page 600
- "Understanding parallel database backups" [*SQL Anywhere Server - Database Administration*]

Standards and compatibility

- **SQL/2003** Vendor extension.
- **Windows Mobile** Only the BACKUP DATABASE DIRECTORY syntax (syntax 1 above) is supported on Windows Mobile.

Example

Back up the current database and the transaction log, each to a different file, and rename the existing transaction log. An image backup is created.

```
BACKUP DATABASE
DIRECTORY 'd:\\temp\\backup'
TRANSACTION LOG RENAME;
```

The option to rename the transaction log is useful especially in replication environments, where the old transaction log is still required.

Back up the current database and transaction log to tape:

```
BACKUP DATABASE
TO '\\\\.\\tape0';
```

Rename the log without making a copy:

```
BACKUP DATABASE DIRECTORY ' '
TRANSACTION LOG ONLY
TRANSACTION LOG RENAME;
```

Execute the BACKUP DATABASE statement with a dynamically-constructed directory name:

```
CREATE EVENT NightlyBackup
SCHEDULE
START TIME '23:00' EVERY 24 HOURS
HANDLER
BEGIN
    DECLARE dest LONG VARCHAR;
    DECLARE day_name CHAR(20);

    SET day_name = DATENAME( WEEKDAY, CURRENT DATE );
    SET dest = 'd:\\backups\\' || day_name;
            BACKUP DATABASE DIRECTORY dest
    TRANSACTION LOG RENAME;
END;
```

BEGIN statement

Use this statement to group SQL statements together.

Syntax

[*statement-label* :]
BEGIN [[**NOT**] **ATOMIC**]
  [*local-declaration;* ...]
  statement-list
  [**EXCEPTION** [*exception-case* ...]]
END [*statement-label*]

local-declaration :
 variable-declaration
| *cursor-declaration*
| *exception-declaration*
| *temporary-table-declaration*

variable-declaration :
DECLARE *variable-name data-type*

exception-declaration :
DECLARE *exception-name* **EXCEPTION**
FOR SQLSTATE [**VALUE**] *string*

exception-case :
 WHEN *exception-name* [, ...] **THEN** *statement-list*
| **WHEN OTHERS THEN** *statement-list*

Parameters

- **local-declaration** Immediately following the BEGIN, a compound statement can have local declarations for objects that only exist within the compound statement. A compound statement can have a local declaration for a variable, a cursor, a temporary table, or an exception. Local declarations can be referenced by any statement in that compound statement, or in any compound statement nested within it. Local declarations are not visible to other procedures that are called from within a compound statement.
- **statement-label** If the ending *statement-label* is specified, it must match the beginning *statement-label*. The LEAVE statement can be used to resume execution at the first statement after the compound statement. The compound statement that is the body of a procedure or trigger has an implicit label that is the same as the name of the procedure or trigger.

  For a complete description of compound statements and exception handling, see "Errors and warnings in procedures and triggers" [*SQL Anywhere Server - SQL Usage*].
- **ATOMIC clause** An atomic statement is a statement executed completely or not at all. For example, an UPDATE statement that updates thousands of rows might encounter an error after updating many rows. If the statement does not complete, all changes revert back to their original state. Similarly, if you specify that the BEGIN statement is atomic, the statement is executed either in its entirety or not at all.

Remarks

The body of a procedure or trigger is a compound statement. Compound statements can also be used in control statements within a procedure or trigger.

A compound statement allows one or more SQL statements to be grouped together and treated as a unit. A compound statement starts with the keyword BEGIN and ends with the keyword END.

Permissions

None.

Side effects

None.

See also

- "DECLARE CURSOR statement [ESQL] [SP]" on page 552
- "DECLARE LOCAL TEMPORARY TABLE statement" on page 557
- "CONTINUE statement [T-SQL]" on page 431
- "SIGNAL statement" on page 750
- "RESIGNAL statement" on page 713
- "Using procedures, triggers, and batches" [*SQL Anywhere Server - SQL Usage*]
- "Atomic compound statements" [*SQL Anywhere Server - SQL Usage*]

Standards and compatibility

- **SQL/2003** Persistent Stored Module feature.

Example

The body of a procedure or trigger is a compound statement.

```
CREATE PROCEDURE TopCustomer (OUT TopCompany CHAR(35), OUT TopValue INT)
BEGIN
   DECLARE err_notfound EXCEPTION FOR
      SQLSTATE '02000';
   DECLARE curThisCust CURSOR FOR
      SELECT CompanyName, CAST(
            sum( SalesOrderItems.Quantity *
            Products.UnitPrice ) AS INTEGER) VALUE
      FROM Customers
            LEFT OUTER JOIN SalesOrders
            LEFT OUTER JOIN SalesOrderItems
            LEFT OUTER JOIN Products
      GROUP BY CompanyName;
   DECLARE ThisValue INT;
   DECLARE ThisCompany CHAR( 35 );
   SET TopValue = 0;
   OPEN curThisCust;
   CustomerLoop:
   LOOP
      FETCH NEXT curThisCust
         INTO ThisCompany, ThisValue;
      IF SQLSTATE = err_notfound THEN
         LEAVE CustomerLoop;
      END IF;
      IF ThisValue > TopValue THEN
         SET TopValue = ThisValue;
         SET TopCompany = ThisCompany;
      END IF;
   END LOOP CustomerLoop;
   CLOSE curThisCust;
END;
```

BEGIN SNAPSHOT statement

Use this statement to start a snapshot at a specified period in time for use with snapshot isolation transactions.

Syntax

BEGIN SNAPSHOT

Remarks

By default, when a transaction begins, the database server defers creating the snapshot until the application causes the first row of a table to be fetched. You can use the BEGIN SNAPSHOT statement to start the snapshot earlier within the transaction. The database server creates a snapshot when the BEGIN SNAPSHOT statement is executed by a snapshot transaction.

The statement fails and returns an error when either of the following conditions is met:

- support for snapshots transactions has not been enabled for the database. See "allow_snapshot_isolation option [database]" [*SQL Anywhere Server - Database Administration*].
- a snapshot has already been started for the current transaction.

This statement is also useful for non-snapshot transactions because it allows them to start a snapshot that can be used later in the transaction for a statement-level snapshot operation.

Permissions

None.

Side effects

None.

See also

- "Snapshot isolation" [*SQL Anywhere Server - SQL Usage*]

Standards and compatibility

- **SQL/2003** Vendor extension.

BEGIN TRANSACTION statement [T-SQL]

Use this statement to begin a user-defined transaction.

Syntax

BEGIN TRAN[**SACTION**] [*transaction-name*]

Remarks

The optional parameter *transaction-name* is the name assigned to this transaction. It must be a valid identifier. Use transaction names only on the outermost pair of nested BEGIN/COMMIT or BEGIN/ROLLBACK statements.

When executed inside a transaction, the BEGIN TRANSACTION statement increases the nesting level of transactions by one. The nesting level is decreased by a COMMIT statement. When transactions are nested, only the outermost COMMIT makes the changes to the database permanent.

Both Adaptive Server Enterprise and SQL Anywhere have two transaction modes.

The default Adaptive Server Enterprise transaction mode, called unchained mode, commits each statement individually, unless an explicit BEGIN TRANSACTION statement is executed to start a transaction. In contrast, the ISO SQL/2003 compatible chained mode only commits a transaction when an explicit COMMIT is executed or when a statement that carries out an autocommit (such as data definition statements) is executed.

You can control the mode by setting the chained database option. The default setting for ODBC and embedded SQL connections in SQL Anywhere is On, in which case SQL Anywhere runs in chained mode.

(ODBC users should also check the AutoCommit ODBC setting). The default for TDS connections is Off. See "chained option [compatibility]" [*SQL Anywhere Server - Database Administration*].

In unchained mode, a transaction is implicitly started before any data retrieval or modification statement. These statements include: DELETE, INSERT, OPEN, FETCH, SELECT, and UPDATE. You must still explicitly end the transaction with a COMMIT or ROLLBACK statement.

You cannot alter the chained option within a transaction.

Caution
When calling a stored procedure, you should ensure that it operates correctly under the required transaction mode.

The current nesting level is held in the global variable @@trancount. The @@trancount variable has a value of zero before the first BEGIN TRANSACTION statement is executed, and only a COMMIT executed when @@trancount is equal to one makes changes to the database permanent.

A ROLLBACK statement without a transaction or savepoint name always rolls back statements to the outermost BEGIN TRANSACTION (explicit or implicit) statement, and cancels the entire transaction.

Permissions

None.

Side effects

None.

See also

- "COMMIT statement" on page 426
- "isolation_level option [database] [compatibility]" [*SQL Anywhere Server - Database Administration*]
- "ROLLBACK statement" on page 723
- "SAVEPOINT statement" on page 727
- "Savepoints within transactions" [*SQL Anywhere Server - SQL Usage*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following batch reports successive values of @@trancount as 0, 1, 2, 1, and 0. The values are printed on the database server messages window.

```
PRINT @@trancount
BEGIN TRANSACTION
PRINT @@trancount
BEGIN TRANSACTION
PRINT @@trancount
COMMIT
PRINT @@trancount
COMMIT
PRINT @@trancount
```

You should not rely on the value of @@trancount for more than keeping track of the number of explicit BEGIN TRANSACTION statements that have been issued.

When Adaptive Server Enterprise starts a transaction implicitly, the @@trancount variable is set to 1. SQL Anywhere does not set the @@trancount value to 1 when a transaction is started implicitly. So, the SQL Anywhere @@trancount variable has a value of zero before any BEGIN TRANSACTION statement (even though there is a current transaction), while in Adaptive Server Enterprise (in chained mode) it has a value of 1.

For transactions starting with a BEGIN TRANSACTION statement, @@trancount has a value of 1 in both SQL Anywhere and Adaptive Server Enterprise after the first BEGIN TRANSACTION statement. If a transaction is implicitly started with a different statement, and a BEGIN TRANSACTION statement is then executed, @@trancount has a value of 2 in both SQL Anywhere, and Adaptive Server Enterprise after the BEGIN TRANSACTION statement.

BREAK statement [T-SQL]

Use this statement to exit a compound statement or loop.

Syntax

BREAK

Remarks

The BREAK statement is a control statement that allows you to leave a loop. Execution resumes at the first statement after the loop.

Permissions

None.

Side effects

None.

See also

- "WHILE statement [T-SQL]" on page 792
- "CONTINUE statement [T-SQL]" on page 431
- "BEGIN statement" on page 413
- "Using procedures, triggers, and batches" [*SQL Anywhere Server - SQL Usage*]

Standards and compatibility

- **SQL/2003** Transact-SQL extension.

Example

In this example, the BREAK statement breaks the WHILE loop if the most expensive product has a price above $50. Otherwise, the loop continues until the average price is greater than or equal to $30:

```
WHILE ( SELECT AVG( UnitPrice ) FROM Products ) < $30
BEGIN
   UPDATE Products
   SET UnitPrice = UnitPrice + 2
   IF ( SELECT MAX(UnitPrice) FROM Products ) > $50
      BREAK
END
```

CALL statement

Use this statement to invoke a procedure.

Syntax 1

[*variable* =] **CALL** *procedure-name* **(** [*expression*, ...] **)**

Syntax 2

[*variable* =] **CALL** *procedure-name* **(** [*parameter-name* = *expression*, ...] **)**

Remarks

The CALL statement invokes a procedure that has been previously created with a CREATE PROCEDURE statement. When the procedure completes, any INOUT or OUT parameter value is copied back.

The argument list can be specified by position or by using keyword format. By position, the arguments will match up with the corresponding parameter in the parameter list for the procedure. By keyword, the arguments are matched up with the named parameters.

Procedure arguments can be assigned default values in the CREATE PROCEDURE statement, and missing parameters are assigned the default value. If no default is set, and an argument is not provided, an error is given.

Inside a procedure, a CALL statement can be used in a DECLARE statement when the procedure returns result sets. See "Returning results from procedures" [*SQL Anywhere Server - SQL Usage*].

Procedures can return an integer value (for example, as a status indicator) using the RETURN statement. You can save this return value in a variable using the equality sign as an assignment operator:

```
CREATE VARIABLE returnval INT;
returnval = CALL proc_integer ( arg1 = val1, ... )
```

Note
If the procedure being called returns an INT and the value is NULL, then the error status value, 0, is returned instead. There is no way to differentiate between this case and the case of an actual value of 0 being returned.

Permissions

Must be the owner of the procedure, have EXECUTE permission for the procedure, or have DBA authority.

Side effects

None.

See also

- “CREATE FUNCTION statement (web services)” on page 467
- “CREATE PROCEDURE statement (web services)” on page 495
- “GRANT statement” on page 627
- “EXECUTE statement [T-SQL]” on page 598
- “Using procedures, triggers, and batches” [*SQL Anywhere Server - SQL Usage*]

Standards and compatibility

- **SQL/2003** Persistent Stored Module feature.

Example

Call the ShowCustomers procedure. This procedure has no parameters, and returns a result set.

```
CALL ShowCustomers();
```

The following Interactive SQL example creates a procedure to return the number of orders placed by the customer whose ID is supplied, creates a variable to hold the result, calls the procedure, and displays the result.

```
CREATE PROCEDURE OrderCount (IN customer_ID INT, OUT Orders INT)
BEGIN
   SELECT COUNT(SalesOrders.ID)
   INTO Orders
   FROM Customers
   KEY LEFT OUTER JOIN SalesOrders
   WHERE Customers.ID = customer_ID;
END
go
 -- Create a variable to hold the result
CREATE VARIABLE Orders INT
go
-- Call the procedure, FOR customer 101
CALL OrderCount ( 101, Orders )
go
--  Display the result
SELECT Orders FROM DUMMY
go
```

CASE statement

Use this statement to select an execution path based on multiple cases.

Syntax 1

CASE *value-expression*
WHEN [*constant* | **NULL**] **THEN** *statement-list* ...
[**WHEN** [*constant* | **NULL**] **THEN** *statement-list*] ...
[**ELSE** *statement-list*]
END [**CASE**]

Syntax 2

CASE
WHEN [*search-condition* | **NULL**] **THEN** *statement-list* ...

[**WHEN** [*search-condition* | **NULL**] **THEN** *statement-list*] ...
[**ELSE** *statement-list*]
END [**CASE**]

Remarks

Syntax 1 The CASE statement is a control statement that allows you to choose a list of SQL statements to execute based on the value of an expression. The *value-expression* is an expression that takes on a single value, which may be a string, a number, a date, or other SQL data type. If a WHEN clause exists for the value of *value-expression*, the *statement-list* in the WHEN clause is executed. If no appropriate WHEN clause exists, and an ELSE clause exists, the *statement-list* in the ELSE clause is executed. Execution resumes at the first statement after the END CASE.

If the *value-expression* can be null, use the ISNULL function to replace the NULL *value-expression* with a different expression.

Syntax 2 With this form, the statements are executed for the first satisfied *search-condition* in the CASE statement. The ELSE clause is executed if none of the *search-conditions* are met.

If the expression can be NULL, use the following syntax for the first *search-condition*:

```
WHEN search-condition IS NULL THEN statement-list
```

CASE statement is different from CASE expression
Do not confuse the syntax of the CASE statement with that of the CASE expression. See "CASE expressions" on page 18.

Permissions

None.

Side effects

None.

See also

- "ISNULL function [Miscellaneous]" on page 231
- "Unknown Values: NULL" [*SQL Anywhere Server - SQL Usage*]
- "BEGIN statement" on page 413
- "Using procedures, triggers, and batches" [*SQL Anywhere Server - SQL Usage*]

Standards and compatibility

- **SQL/2003** Persistent Stored Module feature.

Example

The following procedure using a case statement classifies the products listed in the Products table of the SQL Anywhere sample database into one of shirt, hat, shorts, or unknown.

```
CREATE PROCEDURE ProductType (IN product_ID INT, OUT type CHAR(10))
BEGIN
   DECLARE prod_name CHAR(20);
   SELECT Name INTO prod_name FROM Products
   WHERE ID = product_ID;
```

```
      CASE prod_name
      WHEN 'Tee Shirt' THEN
         SET type = 'Shirt'
      WHEN 'Sweatshirt' THEN
         SET type = 'Shirt'
      WHEN 'Baseball Cap' THEN
         SET type = 'Hat'
      WHEN 'Visor' THEN
         SET type = 'Hat'
      WHEN 'Shorts' THEN
         SET type = 'Shorts'
      ELSE
         SET type = 'UNKNOWN'
      END CASE;
   END;
```

The following example uses Syntax 2 to generate a message about product quantity within the SQL Anywhere sample database.

```
CREATE PROCEDURE StockLevel (IN product_ID INT)
BEGIN
   DECLARE qty INT;
   SELECT Quantity INTO qty FROM Products
   WHERE ID = product_ID;
   CASE
   WHEN qty < 30 THEN
      MESSAGE 'Order Stock' TO CLIENT;
   WHEN qty > 100 THEN
      MESSAGE 'Overstocked' TO CLIENT;
   ELSE
      MESSAGE 'Sufficient stock on hand' TO CLIENT;
   END CASE;
END;
```

CHECKPOINT statement

Use this statement to checkpoint the database.

Syntax

CHECKPOINT

Remarks

The CHECKPOINT statement forces the database server to execute a checkpoint. Checkpoints are also performed automatically by the database server according to an internal algorithm. It is not normally required for applications to issue the CHECKPOINT statement.

Permissions

DBA authority is required to checkpoint the network database server.

No permissions are required to checkpoint the personal database server.

Side effects

None.

See also

- "Backup and data recovery" [*SQL Anywhere Server - Database Administration*]
- "Understanding the checkpoint log" [*SQL Anywhere Server - Database Administration*]
- "checkpoint_time option [database]" [*SQL Anywhere Server - Database Administration*]
- "recovery_time option [database]" [*SQL Anywhere Server - Database Administration*]

Standards and compatibility

- **SQL/2003**   Vendor extension.

CLEAR statement [Interactive SQL]

Use this statement to clear the Interactive SQL panes.

Syntax

CLEAR

Remarks

The CLEAR statement is used to clear the **SQL Statements** and the **Results** panes.

Permissions

None.

Side effects

Closes the cursor associated with the data being cleared.

See also

- "Using Interactive SQL" [*SQL Anywhere Server - Database Administration*]

Standards and compatibility

- **SQL/2003**   Vendor extension.

CLOSE statement [ESQL] [SP]

Use this statement to close a cursor.

Syntax

CLOSE *cursor-name*

cursor-name : *identifier* | *hostvar*

Remarks

This statement closes the named cursor.

Permissions

The cursor must have been previously opened.

Side effects

None.

See also

- "OPEN statement [ESQL] [SP]" on page 683
- "DECLARE CURSOR statement [ESQL] [SP]" on page 552
- "PREPARE statement [ESQL]" on page 694

Standards and compatibility

- **SQL/2003** Core feature.

Example

The following examples close cursors in embedded SQL.

```
EXEC SQL CLOSE employee_cursor;
EXEC SQL CLOSE :cursor_var;
```

The following procedure uses a cursor.

```
CREATE PROCEDURE TopCustomer (OUT TopCompany CHAR(35), OUT TopValue INT)
BEGIN
   DECLARE err_notfound EXCEPTION
      FOR SQLSTATE '02000';
   DECLARE curThisCust CURSOR FOR
   SELECT CompanyName, CAST(    sum(SalesOrderItems.Quantity *
   Products.UnitPrice) AS INTEGER) VALUE
   FROM Customers
   LEFT OUTER JOIN SalesOrders
   LEFT OUTER JOIN SalesOrderItems
   LEFT OUTER JOIN Products
   GROUP BY CompanyName;
DECLARE ThisValue INT;
   DECLARE ThisCompany CHAR(35);
   SET TopValue = 0;
   OPEN curThisCust;
   CustomerLoop:
   LOOP
      FETCH NEXT curThisCust
      INTO ThisCompany, ThisValue;
         IF SQLSTATE = err_notfound THEN
            LEAVE CustomerLoop;
         END IF;
         IF ThisValue > TopValue THEN
            SET TopValue = ThisValue;
            SET TopCompany = ThisCompany;
         END IF;
      END LOOP CustomerLoop;
   CLOSE curThisCust;
END
```

COMMENT statement

Use this statement to store a comment for a database object in the system tables.

Syntax

```
COMMENT ON {
   COLUMN [ owner.]table-name.column-name
  | DBSPACE dbspace-name
  | EVENT [ owner.]event-name
  | EXTERNAL ENVIRONMENT environment-name
  | EXTERNAL OBJECT object-name
  | FOREIGN KEY [ owner.]table-name.role-name
  | INDEX [ [ owner.] table.]index-name
  | INTEGRATED LOGIN integrated-login-id
  | JAVA CLASS java-class-name
  | JAVA JAR java-jar-name
  | KERBEROS LOGIN "client-Kerberos-principal"
  | LOGIN POLICY policy-name
  | MATERIALIZED VIEW [ owner.]materialized-view-name
  | PRIMARY KEY ON [ owner.]table-name
  | PROCEDURE [ owner.]procedure-name
  | SERVICE web-service-name
  | TABLE [ owner.]table-name
  | TEXT CONFIGURATION [ owner.]text-config-name
  | TEXT INDEX text-index-name ON [ owner.]table-name
  | TRIGGER [ [ owner.]tablename.]trigger-name
  | USER userid
  | VIEW [ owner.]view-name
}
IS comment

comment : string | NULL

environment-name :
JAVA
| PERL
| PHP
| CLR
| C_ESQL32
| C_ESQL64
| C_ODBC32
| C_ODBC64
```

Remarks

The COMMENT statement allows you to set a remark (comment) for an object in the database. The COMMENT statement updates remarks listed in the ISYSREMARKS system table. You can remove a comment by setting it to NULL. For a comment on an index or trigger, the owner of the comment is the owner of the table on which the index or trigger is defined.

You cannot add comments for local temporary tables.

The *environment-name* is one of JAVA, PERL, PHP, CLR, C_ESQL32, C_ESQL64, C_ODBC32, or C_ODBC64.

If you use the **Database Documentation Generator** to document your SQL Anywhere database, you have the option to include the comments for procedures, functions, triggers, events, and views in the output. See "Documenting a database" [*SQL Anywhere Server - Database Administration*].

Permissions

Must either be the owner of the database object being commented, or have DBA authority.

Side effects

Automatic commit.

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following examples show how to add and remove a comment.

Add a comment to the Employees table.

```
COMMENT
ON TABLE Employees
IS 'Employee information';
```

Remove the comment from the Employees table.

```
COMMENT
ON TABLE Employees
IS NULL;
```

To view the comment set for an object, use a SELECT statement similar to the following. This example retrieves the comment set for the ViewSalesOrders view in the SQL Anywhere sample database.

```
SELECT remarks
FROM SYSTAB t, SYSREMARK r
WHERE t.object_id = r.object_id
AND t.table_name = 'ViewSalesOrders';
```

COMMIT statement

Use this statement to make changes to the database permanent, or to terminate a user-defined transaction.

Syntax 1

COMMIT [**WORK**]

Syntax 2

COMMIT TRAN[**SACTION**] [*transaction-name*]

Parameters

- **transaction-name** An optional name assigned to this transaction. It must be a valid identifier. You should use transaction names only on the outermost pair of nested BEGIN/COMMIT or BEGIN/ROLLBACK statements.

For more information about transaction nesting in Adaptive Server Enterprise and SQL Anywhere, see "BEGIN TRANSACTION statement [T-SQL]" on page 416. For more information about savepoints, see "SAVEPOINT statement" on page 727.

You can use a set of options to control the detailed behavior of the COMMIT statement. See:

- "cooperative_commit_timeout option [database]" [*SQL Anywhere Server - Database Administration*]
- "cooperative_commits option [database]" [*SQL Anywhere Server - Database Administration*]
- "delayed_commits option [database]" [*SQL Anywhere Server - Database Administration*]
- "delayed_commit_timeout option [database]" [*SQL Anywhere Server - Database Administration*]

You can use the Commit connection property to return the number of Commits on the current connection. See "Connection properties" [*SQL Anywhere Server - Database Administration*].

Remarks

Syntax 1 The COMMIT statement ends a transaction and makes all changes made during this transaction permanent in the database.

Data definition statements all carry out a commit automatically. For information, see the Side effects listing for each SQL statement.

The COMMIT statement fails if the database server detects any invalid foreign keys. This makes it impossible to end a transaction with any invalid foreign keys. Usually, foreign key integrity is checked on each data manipulation operation. However, if the database option wait_for_commit is set On or a particular foreign key was defined with a CHECK ON COMMIT option, the database server delays integrity checking until the COMMIT statement is executed.

Syntax 2 You can use BEGIN TRANSACTION and COMMIT TRANSACTION statements in pairs to construct nested transactions. Nested transactions are similar to savepoints. When executed as the outermost of a set of nested transactions, the statement makes changes to the database permanent. When executed inside a transaction, the COMMIT TRANSACTION statement decreases the nesting level of transactions by one. When transactions are nested, only the outermost COMMIT makes the changes to the database permanent.

Syntax 2 is a Transact-SQL extension.

Permissions

None.

Side effects

Closes all cursors except those opened WITH HOLD.

Deletes all rows of declared temporary tables on this connection, unless they were declared using ON COMMIT PRESERVE ROWS.

See also

- "SAVEPOINT statement" on page 727
- "BEGIN TRANSACTION statement [T-SQL]" on page 416
- "PREPARE TO COMMIT statement" on page 696
- "ROLLBACK statement" on page 723

Standards and compatibility

- **SQL/2003** Syntax 1 is a core feature. Syntax 2 is a Transact-SQL extension.

Example

The following statement commits the current transaction:

```
COMMIT;
```

The following Transact-SQL batch reports successive values of @@trancount as 0, 1, 2, 1, 0.

```
PRINT @@trancount
BEGIN TRANSACTION
PRINT @@trancount
BEGIN TRANSACTION
PRINT @@trancount
COMMIT TRANSACTION
PRINT @@trancount
COMMIT TRANSACTION
PRINT @@trancount
go
```

CONFIGURE statement [Interactive SQL]

Use this statement to open the Interactive SQL Options window.

Syntax

CONFIGURE

Remarks

The CONFIGURE statement opens the Interactive SQL Options window. This window displays the current settings of all Interactive SQL options. It does not display or allow you to modify database options. You can configure Interactive SQL settings in this window.

Permissions

None.

Side effects

None.

See also

- "SET OPTION statement" on page 742
- "Using Interactive SQL" [*SQL Anywhere Server - Database Administration*]

Standards and compatibility

- **SQL/2003** Vendor extension.

CONNECT statement [ESQL] [Interactive SQL]

Use this statement to establish a connection to a database.

Syntax 1

CONNECT
[**TO** *database-server-name*]
[**DATABASE** *database-name*]
[**AS** *connection-name*]
[**USER**] *userid* [**IDENTIFIED BY** *password*]

database-server-name, *database-name*, *connection-name*, *userid*, *password* :
{ *identifier* | *string* | *hostvar* }

Syntax 2

CONNECT USING *connect-string*

connect-string : { *identifier* | *string* | *hostvar* }

Parameters

- **AS clause** A connection can optionally be named by specifying the AS clause. This allows multiple connections to the same database, or multiple connections to the same or different database servers, all simultaneously. Each connection has its own associated transaction. You may even get locking conflicts between your transactions if, for example, you try to modify the same record in the same database from two different connections.

Syntax 2 A *connect-string* is a list of parameter settings of the form keyword=*value*, separated by semicolons, and must be enclosed in single quotes.

For more information about connection strings, see "Connection parameters" [*SQL Anywhere Server - Database Administration*].

Remarks

The CONNECT statement establishes a connection to the database identified by *database-name* running on the database server identified by *database-server-name*. This statement is not supported in procedures, triggers, events, or batches.

Embedded SQL behavior In embedded SQL, if no *database-server-name* is specified, the default local database server is assumed (the first database server started). If no *database-name* is specified, the first database on the given server is assumed.

The WHENEVER statement, SET SQLCA and some DECLARE statements do not generate code and may appear before the CONNECT statement in the source file. Otherwise, no statements are allowed until a successful CONNECT statement has been executed.

The user ID and password are used for permission checks on all dynamic SQL statements.

For a detailed description of the connection algorithm, see "Troubleshooting connections" [*SQL Anywhere Server - Database Administration*].

Note
For SQL Anywhere, only Syntax 1 is valid with embedded SQL. For UltraLite, both Syntax 1 and Syntax 2 can be used with embedded SQL.

Interactive SQL behavior If no database or server is specified in the CONNECT statement, Interactive SQL remains connected to the current database, rather than to the default server and database. If a database name is specified without a server name, Interactive SQL attempts to connect to the specified database on the current server. If a server name is specified without a database name, Interactive SQL connects to the default database on the specified server.

For example, if the following batch is executed while connected to a database, the two tables are created in the same database.

```
CREATE TABLE t1( c1 int );
CONNECT DBA IDENTIFIED BY sql;
CREATE TABLE t2 (c1 int );
```

No other database statements are allowed until a successful CONNECT statement has been executed.

When Interactive SQL is run in windowed mode, you are prompted for any missing connection parameters.

When Interactive SQL is running in command-prompt mode (-nogui is specified when you start Interactive SQL from a command line) or batch mode, or if you execute CONNECT without an AS clause, an unnamed connection is opened. If there is another unnamed connection already opened, the old one is automatically closed. Otherwise, existing connections are not closed when you run CONNECT.

Multiple connections are managed through the concept of a current connection. After a successful connect statement, the new connection becomes the current one. To switch to a different connection, use the SET CONNECTION statement. The DISCONNECT statement is used to drop connections.

When connecting to Interactive SQL, specifying CONNECT [USER] *userid* is the same as executing a SETUSER WITH OPTION *userid* statement. See "SETUSER statement" on page 748.

In Interactive SQL, the connection information (including the database name, your user ID, and the database server) appears in the title bar above the SQL Statements pane. If you are not connected to a database, Not Connected appears in the title bar.

Note
Both Syntax 1 and Syntax 2 are valid with Interactive SQL with the exception that Interactive SQL does not support the *hostvar* argument.

Permissions

None.

Side effects

None.

See also

- “GRANT statement” on page 627
- “DISCONNECT statement [ESQL] [Interactive SQL]” on page 569
- “SET CONNECTION statement [Interactive SQL] [ESQL]” on page 740
- “SETUSER statement” on page 748
- “Connection parameters” [*SQL Anywhere Server - Database Administration*]
- “Using Interactive SQL” [*SQL Anywhere Server - Database Administration*]

Standards and compatibility

- **SQL/2003** Syntax 1 is a SQL/foundation feature outside core SQL. Syntax 2 is a vendor extension.

Examples

The following are examples of CONNECT usage within embedded SQL.

```
EXEC SQL CONNECT AS :conn_name
USER :userid IDENTIFIED BY :password;
EXEC SQL CONNECT USER "DBA" IDENTIFIED BY "sql";
```

The following examples assume that the SQL Anywhere sample database has already been started.

Connect to a database from Interactive SQL. Interactive SQL prompts for a user ID and a password.

```
CONNECT;
```

Connect to the default database as DBA from Interactive SQL. Interactive SQL prompts for a password.

```
CONNECT USER "DBA";
```

Connect to the sample database as user DBA from Interactive SQL.

```
CONNECT
TO demo11
USER DBA
IDENTIFIED BY sql;
```

Connect to the sample database using a connect string, from Interactive SQL.

```
CONNECT
USING 'UID=DBA;PWD=sql;DBN=demo';
```

Once you connect to the sample database, the database name, your user ID, and the database server name appear in the title bar as: **demo (DBA) on demo11**.

CONTINUE statement [T-SQL]

Use this statement to restart a loop.

Syntax

CONTINUE [*statement-label*]

Remarks

The CONTINUE statement is a control statement that allows you to restart a loop. Execution continues at the first statement in the loop. When CONTINUE appears within a set of statements using the Watcom-SQL, *statement-label* must be specified.

When CONTINUE appears within a set of statements using the Transact-SQL, *statement-label* must not be used.

Permissions

None.

Side effects

None.

See also

- "LOOP statement" on page 672
- "WHILE statement [T-SQL]" on page 792
- "FOR statement" on page 608
- "BEGIN statement" on page 413
- "Using procedures, triggers, and batches" [*SQL Anywhere Server - SQL Usage*]

Standards and compatibility

- **SQL/2003** Transact-SQL extension.

Example

The following fragment shows how the CONTINUE statement is used to restart a loop. This example displays the odd numbers between 1 and 10.

```
BEGIN
   DECLARE i INT;
   SET i = 0;
   lbl:
   WHILE i < 10 LOOP
      SET i = i + 1;
      IF mod( i, 2 ) = 0 THEN
         CONTINUE lbl
      END IF;
      MESSAGE 'The value ' || i || ' is odd.' TO CLIENT;
   END LOOP lbl;
END
```

CREATE DATABASE statement

Use this statement to create a database. The database is stored as an operating system file.

Syntax

CREATE DATABASE *db-filename-string* [*create-option* ...]

create-option :
  [**ACCENT** { **RESPECT** | **IGNORE** | **FRENCH** }]
  [**ASE** [**COMPATIBLE**]]
  [**BLANK PADDING** { **ON** | **OFF** }]
  [**CASE** { **RESPECT** | **IGNORE** }]
  [**CHECKSUM** { **ON** | **OFF** }]
  [**COLLATION** *collation-label*[**(** *collation-tailoring-string* **)**]]
  [**DATABASE SIZE** *size* { **KB** | **MB** | **GB** | **PAGES** | **BYTES** }]
  [**DBA USER** *userid*]
  [**DBA PASSWORD** *password*]
  [**ENCODING** *encoding-label*]
  [**ENCRYPTED** [**TABLE**] { *algorithm-key-spec* | **OFF** }]
  [**JCONNECT** { **ON** | **OFF** }]
  [**PAGE SIZE** *page-size*]
  [**NCHAR COLLATION** *nchar-collation-label*[**(** *collation-tailoring-string* **)**]]
  [[**TRANSACTION**] { **LOG OFF** | **LOG ON** [*log-filename-string*]
    [**MIRROR** *mirror-filename-string*] }]

page-size :
2048 | **4096** | **8192** | **16384** | **32768**

algorithm-key-spec:
ON
| [**ON**] **KEY** *key* [**ALGORITHM** *AES-algorithm*]
| [**ON**] **ALGORITHM** *AES-algorithm* **KEY** *key*
| [**ON**] **ALGORITHM 'SIMPLE'**

AES-algorithm :
'AES' | **'AES256'** | **'AES_FIPS'** | **'AES256_FIPS'**

key : quoted string

Parameters

The file names (*db-filename-string*, *log-filename-string*, and *mirror-filename-string*) are strings containing operating system file names. As literal strings, they must be enclosed in single quotes.

- If you specify a path, any backslash characters (\) must be doubled if they are followed by an n or an x. Escaping them prevents them from being interpreted as new line characters (\n) or as hexadecimal numbers (\x), according to the rules for strings in SQL.

  It is always safer to escape the backslash character. For example:

  ```
  CREATE DATABASE 'c:\\databases\\my_db.db'
  LOG ON 'e:\\logdrive\\my_db.log';
  ```

- If you do not specify a path, or a relative path, the database file is created relative to the working directory of the database server. If you specify no path for a transaction log file, the file is created in the same directory as the database file. It is recommended that you store the database files and the transaction log on separate disks on the computer.

- If you provide no file extension, a file is created with extension *.db* for databases, *.log* for the transaction log, and *.mlg* for the transaction log mirror.

You cannot specify utility_db for *db-filename-string*. This name is reserved for the utility database. See "Using the utility database" [*SQL Anywhere Server - Database Administration*].

- **ACCENT clause** This clause is used to specify accent sensitivity for the database. Support for this clause is deprecated. Use the collation tailoring options provided for the COLLATION and NCHAR COLLATION clauses to specify accent sensitivity.

  The ACCENT clause applies only when using the UCA (Unicode Collation Algorithm) for the collation specified in the COLLATION or NCHAR COLLATION clause. ACCENT RESPECT causes the UCA string comparison to respect accent differences between letters. For example, e is less than é. ACCENT FRENCH is similar to ACCENT RESPECT, except that accents are compared from right to left, consistent with the rules of the French language. ACCENT IGNORE causes string comparisons to ignore accents. For example, e is equal to é.

  If accent sensitivity is not specified when the database is created, default accent sensitivity for comparisons and sorting is *insensitive*, with one exception; for Japanese databases created with a UCA collation, default accent sensitivity is *sensitive*.

  For more information about character sets, see "International languages and character sets" [*SQL Anywhere Server - Database Administration*].

- **ASE COMPATIBLE clause** Do not create the SYS.SYSCOLUMNS and SYS.SYSINDEXES views. By default, these views are created for compatibility with system tables available in Watcom SQL (version 4 and earlier of this software). These views conflict with the Sybase Adaptive Server Enterprise compatibility views dbo.syscolumns and dbo.sysindexes.

- **BLANK PADDING clause** SQL Anywhere compares all strings as if they are varying length and stored using the VARCHAR domain. This includes string comparisons involving fixed length CHAR or NCHAR columns. In addition, SQL Anywhere never trims or pads values with trailing blanks when the values are stored in the database.

  By default, SQL Anywhere treats blanks as significant characters. For example, the value 'a ' (the character 'a' followed by a blank) is not equivalent to the single-character string 'a'. Inequality comparisons also treat a blank as any other character in the collation.

  If blank padding is enabled (specifying BLANK PADDING ON), the semantics of string comparisons more closely follow the ANSI/ISO SQL standard. With blank-padding enabled, SQL Anywhere ignores trailing blanks in any comparison.

  In the example above, an equality comparison of 'a ' to 'a' in a blank-padded database returns TRUE. With a blank-padded database, fixed-length string values are padded with blanks when they are fetched by an application. Whether the application receives a string truncation warning on such an assignment is controlled by the ansi_blanks connection option. See "ansi_blanks option [compatibility]" [*SQL Anywhere Server - Database Administration*].

- **CASE clause** This clause is used to specify case sensitivity for the database. Support for this clause is deprecated. Use the collation tailoring options provided for the COLLATION and NCHAR COLLATION clauses to specify case sensitivity.

  CASE RESPECT causes case-sensitive string comparisons for all CHAR and NCHAR data types. Comparisons using UCA consider the case of a letter only if the base letters and accents are all equal. For all other collations, uppercase and lowercase letters are distinct; for example, a is less than A, which is less than b, and so on. CASE IGNORE causes case-insensitive string comparisons. Uppercase and lowercase letters are considered to be exactly equal.

If case sensitivity is not specified when the database is created, default case sensitivity for comparisons and sorting is *insensitive*, with one exception; for Japanese databases created with a UCA collation, default case sensitivity is *sensitive*.

CASE RESPECT is provided for compatibility with the ISO/ANSI SQL standard. Identifiers in the database are always case insensitive, even in case-sensitive databases.

For more information about character sets, see "International languages and character sets" [*SQL Anywhere Server - Database Administration*].

- **CHECKSUM clause** Checksums are used to determine whether a database page has been modified on disk. When you create a database with checksums enabled, a checksum is calculated for each page just before it is written to disk. The next time the page is read from disk, the page's checksum is recalculated and compared to the checksum stored on the page. If the checksums are different, then the page has been modified on disk and an error occurs. Databases created with checksums enabled can also be validated using checksums. You can check whether a database was created with checksums enabled by executing the following statement:

  ```
  SELECT DB_PROPERTY ( 'Checksum' );
  ```

  This query returns ON if checksums are turned on, otherwise, it returns OFF. Checksums are turned off by default, so if the CHECKSUM clause is omitted, OFF is applied.

  Regardless of the setting of this clause, the database server always enables checksums for databases running on storage devices such as removable drives, and databases running on Windows Mobile to help provide early detection if the database file becomes corrupt. It also calculates checksums for critical pages during validation activities. See "Using checksums to detect corruption" [*SQL Anywhere Server - Database Administration*], "Validation utility (dbvalid)" [*SQL Anywhere Server - Database Administration*], "sa_validate system procedure" on page 963, or "VALIDATE statement" on page 787.

- **COLLATION clause** The collation specified by the COLLATION clause is used for sorting and comparison of character data types (CHAR, VARCHAR, and LONG VARCHAR). The collation provides character comparison and ordering information for the encoding (character set) being used. If the COLLATION clause is not specified, SQL Anywhere chooses a collation based on the operating system language and encoding.

  The collation can be chosen from the list of collations that use the SQL Anywhere Collation Algorithm, or it can be the Unicode Collation Algorithm (UCA). If UCA is specified, you should also specify the ENCODING clause.

  It is important to choose your collation carefully. It cannot be changed after the database has been created. See "Choosing collations" [*SQL Anywhere Server - Database Administration*].

  For a list of supported collations, see "Recommended character sets and collations" [*SQL Anywhere Server - Database Administration*], and "Supported and alternate collations" [*SQL Anywhere Server - Database Administration*].

  Optionally, you can specify collation tailoring options (*collation-tailoring-string*) for additional control over the sorting and comparing of characters. These options take the form of keyword=value pairs, assembled in parentheses, following the collation name. For example, `... CHAR COLLATION 'UCA(locale=es;case=respect;accent=respect)'`.

- **DATABASE SIZE clause** Pre-allocating space for the database helps reduce the risk of running out of space on the drive the database is located on. As well, it can help improve performance by increasing the amount of data that can be stored in the database before the database server needs to grow the database, which can be a time-consuming operation. You can use KB, MB, GB, or PAGES to specify units of kilobytes, megabytes, gigabytes, or pages respectively.

- **DBA USER clause** Use this clause to specify a DBA user for the database. When you use this clause, you can no longer connect to the database as the default DBA user. If you do not specify this clause, the default DBA user ID is created.

- **DBA PASSWORD clause** You can specify a different password for the DBA database user. If you do not specify this clause, the default password (**sql**) is used for the DBA user.

- **ENCODING clause** Most collations specified in the COLLATION clause dictate both the encoding (character set) and ordering. For those collations, the ENCODING clause should not be specified. However, if the value specified in the COLLATION clause is UCA (Unicode Collation Algorithm), use the ENCODING clause to specify a locale-specific encoding and get the benefits of the UCA for comparison and ordering. The ENCODING clause may specify UTF-8 or any single-byte encoding for CHAR data types. ENCODING may not specify a multibyte encoding other than UTF-8.

  If you choose the UCA collation, you can optionally specify collation tailoring options. See "Collation tailoring options" [*SQL Anywhere Server - Database Administration*].

  If COLLATION is set to UCA and ENCODING is not specified, then SQL Anywhere uses UTF-8. For more information about the recommended encodings and collations, see "Recommended character sets and collations" [*SQL Anywhere Server - Database Administration*].

  For more information about how to obtain the list of SQL Anywhere supported encodings, see "Supported character sets" [*SQL Anywhere Server - Database Administration*].

- **ENCRYPTED or ENCRYPTED TABLE clause** Encryption makes stored data unreadable. Use the ENCRYPTED keyword (without TABLE) when you want to encrypt the entire database. Use the ENCRYPTED TABLE clause when you only want to enable table encryption. Enabling table encryption means that the tables that are subsequently created or altered using the ENCRYPTED clause are encrypted using the settings you specified at database creation. See "Table encryption" [*SQL Anywhere Server - Database Administration*].

  There are two levels of database and table encryption: simple and strong. Simple encryption is equivalent to obfuscation. The data is unreadable, but someone with cryptographic expertise could decipher the data. Strong encryption makes the data is unreadable and virtually undecipherable.

  For simple encryption, specify ENCRYPTED ON ALGORITHM SIMPLE, or ENCRYPTED ALGORITHM SIMPLE, or specify the ENCRYPTED ON clause without specifying an algorithm or key.

  For strong encryption, specify ENCRYPTED ON ALGORITHM with a 128-bit or 256-bit AES algorithm, and the KEY clause to specify an encryption key. It is recommended that you choose a value for your key that is at least 16 characters long, contains a mix of uppercase and lowercase, and includes numbers, letters, and special characters.

  On Windows Mobile, the AES_FIPS and AES256_FIPS algorithms are only supported with ARM processors.

Caution
For strongly encrypted databases, be sure to store a copy of the key in a safe location. If you lose the encryption key there is no way to access the data, even with the assistance of technical support. The database must be discarded and you must create a new database.

For more information about strong database encryption, see "Strong encryption" [*SQL Anywhere Server - Database Administration*].

You can also create an encrypted copy of an existing database using the CREATE ENCRYPTED DATABASE statement. See "CREATE ENCRYPTED DATABASE statement" on page 445.

- **JCONNECT clause** To allow the jConnect JDBC driver access to system catalog information, specify JCONNECT ON. This will install the system objects that provide jConnect support. Specify JCONNECT OFF if you want to exclude the jConnect system objects. You can still use JDBC, as long as you do not access system information. JCONNECT is ON by default.

  If you are creating a database for use on Windows Mobile, see "Using jConnect on Windows Mobile" [*SQL Anywhere Server - Database Administration*].

- **NCHAR COLLATION clause** The collation specified by the NCHAR COLLATION clause is used for sorting and comparison of national character data types (NCHAR, NVARCHAR, and LONG NVARCHAR). The collation provides character ordering information for the UTF-8 encoding (character set) used for national characters. If the NCHAR COLLATION clause is not specified, SQL Anywhere uses the Unicode Collation Algorithm (UCA). The only other allowed collation is UTF8BIN, which provides a binary ordering of all characters whose encoding is greater than 0x7E. See "Choosing collations" [*SQL Anywhere Server - Database Administration*].

  Optionally, you can specify collation tailoring options (*collation-tailoring-string*) for additional control over the sorting and comparing of characters. These options take the form of keyword=value pairs, assembled in a quoted string, following the collation name. For example, `... NCHAR COLLATION 'UCA(locale=es;case=respect;accent=respect)'`. If you specify the ACCENT or CASE clause and a collation tailoring string that contains settings for case and accent, the values of the ACCENT and CASE clauses are used as defaults only.

  For a list of the supported collation tailoring options, see "Collation tailoring options" [*SQL Anywhere Server - Database Administration*].

  Note
  When you specify the UCA collation, all collation tailoring options are supported. For all other collations, only the case sensitivity tailoring option is supported.

  Note
  Databases created with collation tailoring options cannot be started using a pre-10.0.1 database server.

- **PAGE SIZE clause** The page size for a database can be 2048, 4096, 8192, 16384, or 32768 bytes. The default page size is 4096 bytes. Large databases generally obtain performance benefits from a larger page size, but there can be additional overhead associated with large page sizes.

  For example,

```
CREATE DATABASE 'c:\\databases\\my_db.db'
PAGE SIZE 4096;
```

Page size limit
The page size cannot be larger than the page size used by the current server. The server page size is taken from the first set of databases started or is set on the server command line using the -gp option. See "-gp server option" [*SQL Anywhere Server - Database Administration*].

- **TRANSACTION LOG clause** The transaction log is a file where the database server logs all changes made to the database. The transaction log plays a key role in backup and recovery, and in data replication.

  The MIRROR clause of the TRANSACTION clause allows you to provide a file name if you want to use a transaction log mirror. A transaction log mirror is an identical copy of a transaction log, usually maintained on a separate device, for greater protection of your data. By default, SQL Anywhere does not use a transaction log mirror.

Remarks

Creates a database file with the supplied name and attributes. This statement is not supported in procedures, triggers, events, or batches.

Permissions

The permissions required to execute this statement are set on the server command line, using the -gu option. The default setting is to require DBA authority.

The account under which the database server is running must have write permissions on the directories where files are created.

Side effects

An operating system file is created.

See also

- "ALTER DATABASE statement" on page 358
- "DROP DATABASE statement" on page 571
- "Initialization utility (dbinit)" [*SQL Anywhere Server - Database Administration*]
- "DatabaseKey connection parameter [DBKEY]" [*SQL Anywhere Server - Database Administration*]
- "The transaction log" [*SQL Anywhere Server - Database Administration*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Examples

The following statement creates a database file named *mydb.db* in the *C:\* directory.

```
CREATE DATABASE 'C:\\mydb.db'
TRANSACTION LOG ON
CASE IGNORE
PAGE SIZE 4096
ENCRYPTED OFF
BLANK PADDING OFF;
```

The following statement creates a database using code page 1252 and uses the UCA for both CHAR and NCHAR data types. Accents and case are respected during comparison and sorting.

```
CREATE DATABASE 'c:\\uca.db'
COLLATION 'UCA'
ENCODING 'CP1252'
NCHAR COLLATION 'UCA'
ACCENT RESPECT
CASE RESPECT;
```

The following statement creates a database, *myencrypteddb.db*, that is encrypted using simple encryption:

```
CREATE DATABASE 'myencrypteddb.db'
ENCRYPTED ON;
```

The following statement creates a database, *mystrongencryptdb.db*, that is encrypted using the key gh67AB2 (strong encryption):

```
CREATE DATABASE 'mystrongencryptdb.db'
ENCRYPTED ON KEY 'gh67AB2';
```

The following statement creates a database, *mytableencryptdb.db*, with table encryption enabled using simple encryption. Notice the keyword TABLE inserted after ENCRYPTED to indicate table encryption instead of database encryption:

```
CREATE DATABASE 'mytableencryptdb.db'
ENCRYPTED TABLE ON;
```

The following statement creates a database, *mystrongencrypttabledb.db*, with table encryption enabled using the key gh67AB2 (strong encryption), and the AES_FIPS encryption algorithm:

```
CREATE DATABASE 'mystrongencrypttabledb.db'
ENCRYPTED TABLE ON KEY 'gh67AB2'
ALGORITHM 'AES_FIPS';
```

The following statement creates a database file named *mydb.db* that uses collation 1252LATIN1. The NCHAR collation is set to UCA, with the locale set to es, and has case sensitivity and accent sensitivity enabled:

```
CREATE DATABASE 'my2.db'
      COLLATION '1252LATIN1(case=respect)'
      NCHAR COLLATION 'UCA(locale=es;case=respect;accent=respect)';
```

CREATE DBSPACE statement

Use this statement to define a new database space and create the associated database file.

Syntax

CREATE DBSPACE *dbspace-name* **AS** *filename*

Parameters

- **dbspace-name** Specify a name for the dbspace. This is not the actual database file name, which you specify using *filename*. *dbspace-name* is an internal name you can refer to, for example in statements and procedures. You cannot use the following names for a dbspace because they are reserved for pre-defined dbspaces: system, temporary, temp, translog, and translogmirror. See "Pre-defined dbspaces" [*SQL Anywhere Server - Database Administration*].

An error is returned if you specify a value that contains a period (.).

- **filename** Specify a name for the database file, including, optionally, the path to the file. If no path is specified, the database file is created in the same location (directory) as the main database file. If you specify a different location, the path is relative to the database server. The backslash (\ ) is an escape character in SQL strings, so each backslash must be doubled. For more information about escape characters and strings, see "Strings" on page 9.

  The *filename* parameter must be either a string literal or a variable.

Remarks

The CREATE DBSPACE statement creates a new database file. When a database is created, it is composed of one file. All tables and indexes created are placed in that file. CREATE DBSPACE adds a new file to the database. This file can be on a different disk drive than the main file, which means that the database can be larger than one physical device.

For each database, there is a limit of twelve dbspaces in addition to the main file.

Each object, such as a table or index, is contained entirely within one dbspace. The IN clause of the CREATE statement specifies the dbspace into which an object is placed. Objects are put into the system database file by default. You can also specify which dbspace tables are created in by setting the default_dbspace option before you create the tables.

For information about how the default dbspace for a database object is determined, see "Using additional dbspaces" [*SQL Anywhere Server - Database Administration*].

Permissions

Must have DBA authority.

Side effects

Automatic commit. Automatic checkpoint.

See also

- "default_dbspace option [database]" [*SQL Anywhere Server - Database Administration*]
- "DROP DBSPACE statement" on page 572
- "Using additional dbspaces" [*SQL Anywhere Server - Database Administration*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following CREATE DBSPACE statement creates a dbspace called libbooks. The database file name for the dbspace is *library.db*, located in the *c:\* directory. A subsequent CREATE TABLE statement creates a table called LibraryBooks in the libbooks dbspace.

```
CREATE DBSPACE libbooks
AS 'c:\\library.db';
```

```
CREATE TABLE LibraryBooks (
  title char(100),
  author char(50),
  isbn char(30),
) IN libbooks;
```

CREATE DECRYPTED DATABASE statement

This statement creates a decrypted copy of an existing database, including all transaction logs and dbspaces.

Syntax

CREATE DECRYPTED DATABASE *newfile*
FROM *oldfile*
[**KEY** *key*]

Parameters

- **FROM clause** Use this clause to specify the name of the database to copy (*oldfile*).
- **KEY clause** Use this clause to specify the encryption key needed to decrypt the database. You do not specify the KEY clause if the existing database was encrypted with SIMPLE encryption, which does not require a key.

Remarks

The CREATE DECRYPTED DATABASE statement produces a new database file (*newfile*), and does not replace or remove the original database file (*oldfile*).

All encrypted tables in *oldfile* are not encrypted in *newfile*, and table encryption is not enabled.

Note
For databases created with SQL Anywhere version 11.0.0 and later, the ISYSCOLSTAT, ISYSUSER, and ISYSEXTERNLOGIN system tables always remain encrypted to protect the data from unauthorized access to the database file.

If *oldfile* uses transaction log or transaction log mirror files, they are renamed *newfile.log* and *newfile.mlg*, respectively.

If *oldfile* contains dbspace files, a D (decrypted) is added to the file name. For example, when you execute the CREATE DECRYPTED DATABASE statement, if *oldfile* is *mydbspace.dbs*, *newfile* becomes *mydbspace.dbsD*.

You cannot execute this statement on a database that requires recovery. Also, this statement is not supported in procedures, triggers, events, or batches.

Permissions

Must be a user with DBA authority.

Side effects

None.

See also

- "Encrypting and decrypting a database" [*SQL Anywhere Server - Database Administration*]
- "CREATE ENCRYPTED DATABASE statement" on page 445
- "CREATE DECRYPTED FILE statement" on page 442

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The first statement below creates an AES256-encrypted copy of the *demo.db* called *demoEncrypted.db*,. The second statement creates a decrypted copy of *demoEncrypted.db* called *demoDecrypted.db*.

```
CREATE ENCRYPTED DATABASE 'demoEncrypted.db'
   FROM 'demo.db'
   KEY 'Sd8f6654*Mnn'
   ALGORITHM 'AES256';
CREATE DECRYPTED DATABASE 'demoDecrypted.db'
   FROM 'demoEncrypted.db'
   KEY 'Sd8f6654*Mnn';
```

CREATE DECRYPTED FILE statement

This statement creates a decrypted copy of a strongly encrypted database. You can also use this statement to create decrypted copies of other database related files such as transaction logs, transaction log mirrors, and dbspaces.

Syntax

CREATE DECRYPTED FILE *newfile*
FROM *oldfile* **KEY** *key*

Parameters

- **FROM clause** Lists the file name of the encrypted file.

- **KEY clause** Lists the key required to access the encrypted file.

Remarks

Use this statement when your database requires recovery and you need to create a decrypted copy of the database for support reasons. You must also use this statement to decrypt any database-related files such as the transaction log, transaction log mirror, or dbspace files.

The original database file must be strongly encrypted using an encryption key. The resulting file is an exact copy of the encrypted file, without encryption and therefore requiring no encryption key.

If a database is decrypted using this statement, the corresponding transaction log file (and any dbspaces) must also be decrypted to use the database.

If a database requiring recovery is decrypted, its transaction log file must also be decrypted and recovery on the new database will still be necessary. The name of the transaction log file remains the same in this process, so if the database and transaction log file are renamed, then you need to run dblog -t on the resulting database.

You cannot use this statement on a database on which table encryption is enabled. If you have tables you want to decrypt, use the NOT ENCRYPTED clause of the ALTER TABLE statements to decrypt them. See "ALTER TABLE statement" on page 389.

Note
For databases created with SQL Anywhere version 11.0.0 and later, the ISYSCOLSTAT, ISYSUSER, and ISYSEXTERNLOGIN system tables always remain encrypted to protect the data from unauthorized access to the database file.

This statement is not supported in procedures, triggers, events, or batches.

Permissions

Must be a user with DBA authority.

Side effects

None.

See also

- "CREATE ENCRYPTED FILE statement" on page 447
- "CREATE DECRYPTED DATABASE statement" on page 441

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following example decrypts the contacts database and creates a new unencrypted database called contacts2.

```
CREATE DECRYPTED FILE 'contacts2.db'
FROM 'contacts.db'
KEY 'Sd8f6654*Mnn';
```

CREATE DOMAIN statement

Use this statement to create a domain in a database.

Syntax

CREATE { **DOMAIN** | **DATATYPE** } [**AS**] *domain-name data-type*
[[**NOT**] **NULL**]
[**DEFAULT** *default-value*]
[**CHECK (** *condition* **)**]

domain-name : *identifier*

data-type : *built-in data type*, *with precision and scale*

Parameters

- **DOMAIN | DATATYPE clause** It is recommended that you use CREATE DOMAIN, rather than CREATE DATATYPE, because CREATE DOMAIN is the ANSI/ISO SQL3 term.
- **NULL clause** This clause allows you to specify the nullability of a domain. When a domain is used to define a column, nullability is determined as follows:
  - Nullability specified in the column definition
  - Nullability specified in the domain definition
  - If the nullability was not explicitly specified in either the column definition or the domain definition, then the setting of the allow_nulls_by_default option is used.
- **CHECK clause** When creating a CHECK condition, you can use a variable name prefixed with the @ sign in the condition. When the data type is used in the definition of a column, such a variable is replaced by the column name. This allows CHECK conditions to be defined on data types and used by columns of any name.

Remarks

Domains are aliases for built-in data types, including precision and scale values where applicable. They improve convenience and encourage consistency in the database.

Domains are objects within the database. Their names must conform to the rules for identifiers. Domain names are always case insensitive, as are built-in data type names.

The user who creates a data type is automatically made the owner of that data type. No owner can be specified in the CREATE DATATYPE statement. The domain name must be unique, and all users can access the data type without using the owner as prefix.

Domains can have CHECK conditions and DEFAULT values, and you can indicate whether the data type permits NULL values or not. These conditions and values are inherited by any column defined on the data type. Any conditions or values explicitly specified on the column override those specified for the data type.

To drop the data type from the database, use the DROP statement. You must be either the owner of the data type or have DBA authority to drop a domain.

Permissions

Must have RESOURCE authority.

Side effects

Automatic commit.

See also

- "DROP DOMAIN statement" on page 573
- "SQL data types" on page 77

Standards and compatibility

- **SQL/2003** SQL/foundation feature outside core SQL.

Example

The following statement creates a data type named address, which holds a 35-character string, and which may be NULL.

```
CREATE DOMAIN address CHAR( 35 ) NULL;
```

The following statement creates a data type named ID, which does not allow NULLS, and which is autoincremented by default.

```
CREATE DOMAIN ID INT
NOT NULL
DEFAULT AUTOINCREMENT;
```

CREATE ENCRYPTED DATABASE statement

Creates an encrypted copy of an existing database, including all transaction logs and dbspaces. You can also use this statement to create a copy of a database and enable table encryption in the copy.

Syntax 1 - Create an encrypted copy of a database

CREATE ENCRYPTED DATABASE *newfile*
FROM *oldfile*
[**KEY** *newkey*]
[**OLD KEY** *oldkey*]
[**ALGORITHM** *algorithm*

algorithm :
  | **'SIMPLE'**
  | **'AES'**
  | **'AES256'**
  | **'AES_FIPS'**
  | **'AES256_FIPS'**

Syntax 2 - Create a copy of a database with table encryption enabled

CREATE ENCRYPTED TABLE DATABASE *newfile*
FROM *oldfile*
[**KEY** *newkey*]
[**OLD KEY** *oldkey*]
[**ALGORITHM** *algorithm*]

Parameters

- **ENCRYPTED DATABASE clause** Use this clause to specify a name for the new encrypted database.
- **ENCRYPTED TABLE DATABASE clause** Use this clause to specify a name for the new database. The new database is not encrypted, but has table encryption enabled.
- **FROM clause** Use this clause to specify the name of the original database (*oldfile*).
- **KEY clause** If *algorithm-key* is anything other than SIMPLE, use this clause to specify the encryption key for *newfile*.
- **OLD KEY clause** Use this clause to specify the encryption key for *oldfile*. This clause is only required if *oldfile* is encrypted with anything other than SIMPLE encryption.

- **ALGORITHM clause** Use this clause to specify the encryption algorithm to use for *newfile*. If you specify a KEY clause but do not specify the ALGORITHM clause, AES (128-bit encryption) is used by default. If you specify SIMPLE for *algorithm*, you do not specify a KEY clause.

Remarks

oldfile can be an unencrypted database, an encrypted database, or a database with table encryption enabled.

Syntax 1 takes an existing database, *oldfile*, and creates an encrypted copy of it, *newfile*.

Syntax 2 takes an existing database, *oldfile*, and creates a copy of it, *newfile*, with table encryption enabled. When you use this syntax, any tables encrypted in *oldfile* are encrypted in *newfile* as well. If no tables were encrypted in *oldfile*, but you want to encrypt them, you can execute an ALTER TABLE ... ENCRYPTED statement on each table you want to encrypt. See "ALTER TABLE statement" on page 389.

Neither syntax replaces or removes *oldfile*.

If *oldfile* uses transaction log or transaction log mirror files, they are renamed *newfile.log* and *newfile.mlg* respectively.

If *oldfile* contains dbspace files, an E (for encrypted) is added to the file name. For example, when you execute the CREATE ENCRYPTED DATABASE statement, the file *mydbspace.dbs* is changed to *mydbspace.dbsE*.

You can use this statement to change the encryption algorithm and key for a database. However, the CREATE ENCRYPTED DATABASE statement produces a new file (*newfile*), and does not replace or remove the previous version of the file (*oldfile*).

CREATE ENCRYPTED DATABASE and CREATE ENCRYPTED TABLE DATABASE cannot be run against a database that requires recovery. Also, these statements are not supported in procedures, triggers, events, or batches.

For more information about simple and strong encryption, see "Simple encryption" [*SQL Anywhere Server - Database Administration*], and "Strong encryption" [*SQL Anywhere Server - Database Administration*].

You can also encrypt an existing database or change an existing encryption key by unloading and reloading the database using the dbunload -an option with either -ek or -ep. See "Using the dbunload utility to rebuild databases" [*SQL Anywhere Server - SQL Usage*].

You can also create an encrypted database, or a database with table encryption enabled, using the CREATE DATABASE statement. See "CREATE DATABASE statement" on page 432.

Note
FIPS is not available on all platforms. For a list of supported platforms, see http://www.sybase.com/detail?id=1002288.

Permissions

Must be a user with DBA authority.

Side effects

None.

See also

- "Encrypting and decrypting a database" [*SQL Anywhere Server - Database Administration*]
- "Table encryption" [*SQL Anywhere Server - Database Administration*]
- "Simple encryption" [*SQL Anywhere Server - Database Administration*]
- "Strong encryption" [*SQL Anywhere Server - Database Administration*]
- "CREATE DECRYPTED DATABASE statement" on page 441
- "CREATE DATABASE statement" on page 432
- "ALTER TABLE statement" on page 389

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following example creates an encrypted copy of the demo database called *demoEnc.db*. The new database is encrypted with AES256 encryption.

```
CREATE ENCRYPTED DATABASE 'demoEnc.db'
   FROM 'demo.db'
   KEY 'Sd8f6654*Mnn'
   ALGORITHM 'AES256';
```

The following example creates a copy of the demo database called *demoTableEnc.db*. Table encryption is enabled on the new database. Since a key was specified with no algorithm, AES encryption is used.

```
CREATE ENCRYPTED TABLE DATABASE 'demoTableEnc.db'
   FROM 'demo.db'
   KEY 'Sd8f6654';
```

CREATE ENCRYPTED FILE statement

Creates a strongly encrypted copy of a database file, transaction log, transaction log mirror, or dbspace.

Syntax

CREATE ENCRYPTED FILE *newfile*
FROM *oldfile*
{ **KEY** *key* | **KEY** *key* **OLD KEY** *oldkey* }
[**ALGORITHM** {
  'AES'
  | **'AES256'**
  | **'AES_FIPS'**
  | **'AES256_FIPS'** }]

Parameters

- **FROM clause** Specifies the name of the existing file (*oldfile*) on which to execute the CREATE ENCRYPTED FILE statement.

- **KEY clause** Specifies the encryption key to use.

- **OLD KEY clause** Specifies the current key with which the file is encrypted.

- **ALGORITHM clause** Specifies the algorithm used to encrypt the file. If you do not specify an algorithm, AES (128-bit encryption) is used by default.

Remarks

Use this statement when your database requires recovery and you need to create an encrypted copy of the database for support reasons. You must also use this statement to encrypt any database-related files such as the transaction log, transaction log mirror, or dbspace files.

When encrypting the database-related files, you must specify the same algorithm and key for all files related to the same database.

If oldfile has dbspaces or logs associated with it and you encrypt those too, you must ensure that the new name and location of those files is stored with the new database. To do so:

- run dblog -t on the new database to change the name and location of the transaction log
- run dblog -m on the new database to change the name and location of the transaction log mirror
- execute an ALTER DBSPACE statement against the new database to change the location and name of the dbspace files

You can use this statement to change the encryption algorithm and key for a database. However, the CREATE ENCRYPTED FILE statement produces a new file (*newfile*), and does not replace or remove the previous version of the file (*oldfile*).

The name of the transaction log file remains the same in this process, so if the database and transaction log file are renamed, then you need to run dblog -t on the resulting database.

You can also encrypt an existing database or change an existing encryption key by unloading and reloading the database using the dbunload -an option with either -ek or -ep.

If you have a database on which table encryption is enabled, you cannot encrypt the database using this statement. However, you can use this statement to change the key used for table encryption. To encrypt a database that has table encryption enabled, use the CREATE ENCRYPTED DATABASE statement. See "CREATE ENCRYPTED DATABASE statement" on page 445.

This statement is not supported in procedures, triggers, events, or batches.

Note
FIPS is not available on all platforms. For a list of supported platforms, see http://www.sybase.com/detail?id=1002288.

Permissions

Must be a user with DBA authority.

On Windows Mobile, the AES_FIPS and AES256_FIPS algorithms are only supported with ARM processors.

Side effects

None.

See also

- "Encrypting and decrypting a database" [*SQL Anywhere Server - Database Administration*]
- "CREATE ENCRYPTED DATABASE statement" on page 445
- "CREATE DECRYPTED FILE statement" on page 442
- "Unload utility (dbunload)" [*SQL Anywhere Server - Database Administration*]
- "Transaction Log utility (dblog)" [*SQL Anywhere Server - Database Administration*]

Standards and compatibility

- **SQL/2003**   Vendor extension.

Example

The following example encrypts the contacts database and creates a new database called contacts2 that is encrypted with AES_FIPS encryption.

```
CREATE ENCRYPTED FILE 'contacts2.db'
FROM 'contacts.db'
   KEY 'Sd8f6654*Mnn'
   ALGORITHM AES_FIPS;
```

The following example encrypts the contacts database and the contacts log file, renaming the both files. You will need to run `dblog -ek Sd8f6654*Mnn -t contacts2.log contacts.db`, since the log has been renamed and the database file still points to the old log.

```
CREATE ENCRYPTED FILE 'contacts2.db'
   FROM 'contacts.db'
   KEY 'Sd8f6654*Mnn';
CREATE ENCRYPTED FILE 'contacts2.log'
   FROM 'contacts.db'
   KEY 'Sd8f6654*Mnn';
```

The following example encrypts the contacts database and the contacts log file, leaving the original log file name untouched. In this case, you do not need to run dblog, since the name of the file remains the same.

```
CREATE ENCRYPTED FILE 'newpath\contacts.db'
   FROM 'contacts.db'
   KEY 'Sd8f6654*Mnn';
CREATE ENCRYPTED FILE 'newpath\contacts.log'
   FROM 'contacts.log'
   KEY 'Sd8f6654*Mnn';
```

To change the encryption key for a database, first create a copy of the database file using the new key, as shown in this statement:

```
CREATE ENCRYPTED FILE 'newcontacts.db'
 FROM 'contacts.db'
 KEY 'newkey' OLD KEY 'oldkey';
```

Once you have created the encrypted file, delete *contacts.db* and then rename *newcontacts.db* to be *contacts.db*.

CREATE EVENT statement

Use this statement to define an event and its associated handler for automating predefined actions, and to define scheduled actions.

Syntax

```
CREATE EVENT [ owner.]event-name
[ TYPE event-type
        [ WHERE trigger-condition [ AND trigger-condition ] ... ]
   | SCHEDULE schedule-spec, ... ]
[ ENABLE | DISABLE ]
[ AT { CONSOLIDATED | REMOTE | ALL } ]
[ HANDLER
     BEGIN
...
    END ]

event-type :
  BackupEnd
| Connect
| ConnectFailed
| DatabaseStart
| DBDiskSpace
| Deadlock
| "Disconnect"
| GlobalAutoincrement
| GrowDB
| GrowLog
| GrowTemp
| LogDiskSpace
| MirrorFailover
| MirrorServerDisconnect
| RAISERROR
| ServerIdle
| TempDiskSpace

trigger-condition :
event_condition( condition-name ) {
=
| <
| >
| !=
| <=
| >=
} value

schedule-spec :
[ schedule-name ]
  { START TIME start-time | BETWEEN start-time AND end-time }
  [ EVERY period { HOURS | MINUTES | SECONDS } ]
  [ ON { ( day-of-week, ... ) | ( day-of-month, ... ) } ]
  [ START DATE start-date ]

event-name : identifier
```

schedule-name : *identifier*

day-of-week : *string*

day-of-month : *integer*

value : *integer*

period : *integer*

start-time : *time*

end-time : *time*

start-date : *date*

Parameters

- **CREATE EVENT clause** The event name is an identifier. An event has a creator, which is the user creating the event, and the event handler executes with the permissions of that creator. This is the same as stored procedure execution. You cannot create events owned by other users.
- **TYPE clause** You can specify the TYPE clause with an optional WHERE clause; or specify the SCHEDULE.

  The *event-type* is one of the listed set of system-defined event types. The event types are case insensitive. To specify the conditions under which this *event-type* triggers the event, use the WHERE clause. For a description of event-types not listed below, see "Understanding system events" [*SQL Anywhere Server - Database Administration*].

  - **DiskSpace event types** If the database contains an event handler for one of the DiskSpace types, the database server checks the available space on each device associated with the relevant file every 30 seconds.

    In the event the database has more than one dbspace, on separate drives, DBDiskSpace checks each drive and acts depending on the lowest available space.

    The LogDiskSpace event type checks the location of the transaction log and any transaction log mirror, and reports based on the least available space.

    Disk space event types are not supported on Windows Mobile.

    The TempDiskSpace event type checks the amount of temporary disk space.

    If the appropriate event handlers have been defined (DBDiskSpace, LogDiskSpace, or TempDiskSpace), the database server checks the available space on each device associated with a database file every 30 seconds. Similarly, if an event has been defined to handle the system event type ServerIdle, the database server notifies the handler when no requests have been process during the previous 30 seconds.

    You can specify the -fc option when starting the database server to implement a callback function when the database server encounters a file system full condition.

    See "-fc server option" [*SQL Anywhere Server - Database Administration*].

  - **GlobalAutoIncrement event type** The event fires on *each* insert when the number of remaining values for a GLOBAL AUTOINCREMENT is less than 1% of the end of its range. A typical action for the handler could be to request a new value for the global_database_id option, based on the table and number of remaining values which are supplied as parameters to this event.

    You can use the event_condition function with RemainingValues as an argument for this event type.

  - **ServerIdle event type** If the database contains an event handler for the ServerIdle type, the database server checks for server activity every 30 seconds.

  - **Database mirroring event types** The MirrorServerDisconnect event fires when a connection from the primary database server to the mirror server or arbiter server is lost, and the MirrorFailover event fires whenever a server takes ownership of the database. See "Database mirroring system events" [*SQL Anywhere Server - Database Administration*].

- **WHERE clause** The trigger condition determines the condition under which an event is fired. For example, to take an action when the disk containing the transaction log becomes more than 80% full, use the following triggering condition:

  ```
  ...
  WHERE event_condition( 'LogDiskSpacePercentFree' ) < 20
  ...
  ```

  The argument to the event_condition function must be valid for the event type.

  You can use multiple AND conditions to make up the WHERE clause, but you cannot use OR conditions or other conditions.

  For information about valid arguments, see "EVENT_CONDITION function [System]" on page 198.

- **SCHEDULE clause** This clause specifies when scheduled actions are to take place. The sequence of times acts as a set of triggering conditions for the associated actions defined in the event handler.

  You can create more than one schedule for a given event and its associated handler. This permits complex schedules to be implemented. While it is compulsory to provide a schedule-name when there is more than one schedule, it is optional if you provide only a single schedule.

  A scheduled event is recurring if its definition includes EVERY or ON; if neither of these reserved words is used, the event will execute at most once. An attempt to create a non-recurring scheduled event for which the start time has passed will generate an error. When a non-recurring scheduled event has passed, its schedule is deleted, but the event handler is not deleted.

  Scheduled event times are calculated when the schedules are created, and again when the event handler completes execution. The next event time is computed by inspecting the schedule or schedules for the event, and finding the next schedule time that is in the future. If an event handler is instructed to run every hour between 9:00 and 5:00, and it takes 65 minutes to execute, it runs at 9:00, 11:00, 1:00, 3:00, and 5:00. If you want execution to overlap, you must create more than one event.

The subclauses of a schedule definition are as follows:

- **START TIME clause** The first scheduled time for each day on which the event is scheduled. The *start-time* parameter is a string, and cannot be a variable or an expression such as `NOW()`. If a START DATE is specified, the START TIME refers to that date. If no START DATE is specified, the START

TIME is on the current day (unless the time has passed) and each subsequent day (if the schedule includes EVERY or ON).

- **BETWEEN ... AND clause** A range of times during the day outside which no scheduled times occur. The *start-time* and *end-time* parameters are strings, and cannot be variables or expressions such as `NOW()`. If a START DATE is specified, the scheduled times do not occur until that date.
- **EVERY clause** An interval between successive scheduled events. Scheduled events occur only after the START TIME for the day, or in the range specified by BETWEEN ... AND.
- **ON clause** A list of days on which the scheduled events occur. The default is every day if EVERY is specified. Days can be specified as days of the week or days of the month.

  Days of the week are Mon, Tues, and so on. You may also use the full forms of the day, such as Monday. You must use the full forms of the day names if the language you are using is not English, is not the language requested by the client in the connection string, and is not the language which appears in the database server messages window.

  Days of the month are integers from 0 to 31. A value of 0 represents the last day of any month.
- **START DATE clause** The date on which scheduled events are to start occurring. This value is a string, and cannot be a variable or an expression such as `TODAY()`. The default is the current date.

  Each time a scheduled event handler is completed, the next scheduled time and date is calculated.

  1. If the EVERY clause is used, find whether the next scheduled time falls on the current day, and is before the end of the BETWEEN ... AND range. If so, that is the next scheduled time.
  2. If the next scheduled time does not fall on the current day, find the next date on which the event is to be executed.
  3. Find the START TIME for that date, or the beginning of the BETWEEN ... AND range.
- **ENABLE | DISABLE clause** By default, event handlers are enabled. When DISABLE is specified, the event handler does not execute even when the scheduled time or triggering condition occurs. A TRIGGER EVENT statement does *not* cause a disabled event handler to be executed.
- **AT clause** This clause should be used only in the following circumstance: in a SQL Remote setup, use the AT clause against your remote or consolidated databases to restrict the databases at which the event is handled.

  If you do not use the AT clause when creating events for SQL Remote, all databases execute the event. When executed on a consolidated database, this statement will not affect remote databases that have already been extracted.
- **HANDLER clause** Each event has one handler.

Remarks

Events can be used in two main ways:

- **Scheduling actions** The database server executes actions on a timed schedule. You can use this capability to complete scheduled tasks such as backups, validity checks, and queries used to add data to reporting tables.

- **Event handling actions** The database server executes actions when a predefined event occurs. You can use this capability to complete scheduled tasks such as restrict disk space when a disk fills beyond a specified percentage. Event handler actions are committed if errors are not detected during execution, and rolled back if errors are detected.

An event definition includes two distinct pieces. The trigger condition can be an occurrence, such as a disk filling up beyond a defined threshold. A schedule is a set of times, each of which acts as a trigger condition. When a trigger condition is satisfied, the event handler executes. The event handler includes one or more actions specified inside a compound statement (BEGIN... END).

If no trigger condition or schedule specification is supplied, only an explicit TRIGGER EVENT statement can trigger the event. During development, you may want to test event handlers using TRIGGER EVENT, and add the schedule or WHERE clause once testing is complete.

Event errors are logged to the database server message log. See "Logging database server actions" [*SQL Anywhere Server - Database Administration*].

After each execution of an event handler, a COMMIT occurs if no errors occurred. A ROLLBACK occurs if there was an error.

When event handlers are triggered, the database server makes context information, such as the connection ID that caused the event to be triggered, available to the event handler using the event_parameter function. For more information about event_parameter, see "EVENT_PARAMETER function [System]" on page 200.

Permissions

Must have DBA authority.

Event handlers execute on a separate connection, with the permissions of the event owner. To execute with authority other than DBA, you can call a procedure from within the event handler: the procedure executes with the permissions of its owner. The separate connection does not count towards the ten-connection limit of the personal database server.

Side effects

Automatic commit.

See also

- "BEGIN statement" on page 413
- "ALTER EVENT statement" on page 365
- "COMMENT statement" on page 425
- "DROP EVENT statement" on page 574
- "TRIGGER EVENT statement" on page 767
- "EVENT_PARAMETER function [System]" on page 200
- "Understanding system events" [*SQL Anywhere Server - Database Administration*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

Instruct the database server to carry out an automatic backup to tape using the first tape drive, every day at 1 A.M.

```
CREATE EVENT DailyBackup
SCHEDULE daily_backup
START TIME '1:00AM' EVERY 24 HOURS
HANDLER
   BEGIN
      BACKUP DATABASE TO '\\\\.\\tape0'
      ATTENDED OFF
   END;
```

Instruct the database server to carry out an automatic backup of the transaction log only, every hour, Monday to Friday between 8 A.M. and 6 P.M.

```
CREATE EVENT HourlyLogBackup
SCHEDULE hourly_log_backup
BETWEEN '8:00AM' AND '6:00PM'
EVERY 1 HOURS ON
   ('Monday','Tuesday','Wednesday','Thursday','Friday')
HANDLER
   BEGIN
      BACKUP DATABASE DIRECTORY 'c:\\database\\backup'
      TRANSACTION LOG ONLY
      TRANSACTION LOG RENAME
   END;
```

See “Defining trigger conditions for events” [*SQL Anywhere Server - Database Administration*].

CREATE EXISTING TABLE statement

Use this statement to create a new proxy table, which represents an existing object on a remote server.

Syntax

CREATE EXISTING TABLE [*owner*.]*table-name*
[**(***column-definition*, ...**)**]
AT *location-string*

column-definition :
column-name data-type [**NOT NULL**]

location-string :
  remote-server-name.[*db-name*].[*owner*].*object-name*
| *remote-server-name*;[*db-name*];[*owner*];*object-name*

Parameters

- **AT clause** The AT clause specifies the location of the remote object. The AT clause supports the semicolon (;) as a delimiter. If a semicolon is present anywhere in the *location-string* string, the semicolon is the field delimiter. If no semicolon is present, a period is the field delimiter. This allows file names and extensions to be used in the database and owner fields. For example, the following statement maps the table a1 to the MS Access file *mydbfile.mdb*:

```
CREATE EXISTING TABLE a1
AT 'access;d:\mydbfile.mdb;;a1';
```

Remarks

The CREATE EXISTING TABLE statement creates a new local, proxy table that maps to a table at an external location. The CREATE EXISTING TABLE statement is a variant of the CREATE TABLE statement. The EXISTING keyword is used with CREATE TABLE to specify that a table already exists remotely and that its metadata is to be imported into SQL Anywhere. This establishes the remote table as a visible entity to SQL Anywhere users. SQL Anywhere verifies that the table exists at the external location before it creates the table.

If the object does not exist (either host data file or remote server object), the statement is rejected with an error message.

Index information from the host data file or remote server table is extracted and used to create rows for the ISYSIDX system table. This defines indexes and keys in server terms and enables the query optimizer to consider any indexes that may exist on this table.

Referential constraints are passed to the remote location when appropriate.

If column-definitions are not specified, SQL Anywhere derives the column list from the metadata it obtains from the remote table. If column-definitions are specified, SQL Anywhere verifies the column-definitions. Column names, data types, lengths, identity property, and null properties are checked for the following:

- Column names must match identically (although case is ignored).
- Data types in the CREATE EXISTING TABLE statement must match or be convertible to the data types of the column on the remote location. For example, a local column data type is defined as money, while the remote column data type is numeric.
- Each column's NULL property is checked. If the local column's NULL property is not identical to the remote column's NULL property, a warning message is issued, but the statement is not aborted.
- Each column's length is checked. If the length of char, varchar, binary, varbinary, decimal and numeric columns do not match, a warning message is issued, but the command is not aborted.

  You may choose to include only a subset of the actual remote column list in your CREATE EXISTING statement.

Permissions

Must have RESOURCE authority. To create a table for another user, you must have DBA authority.

Not supported on Windows Mobile.

Side effects

Automatic commit.

See also

- "CREATE TABLE statement" on page 523
- "Specify proxy table locations" [*SQL Anywhere Server - SQL Usage*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Examples

Create a proxy table named blurbs for the blurbs table at the remote server server_a.

```
CREATE EXISTING TABLE blurbs
( author_id ID not null,
copy text not null)
AT 'server_a.db1.joe.blurbs';
```

Create a proxy table named blurbs for the blurbs table at the remote server server_a. SQL Anywhere derives the column list from the metadata it obtains from the remote table.

```
CREATE EXISTING TABLE blurbs
AT 'server_a.db1.joe.blurbs';
```

Create a proxy table named rda_employees for the Employees table at the SQL Anywhere remote server, demo11.

```
CREATE EXISTING TABLE rda_employees
AT 'demo11...Employees';
```

CREATE EXTERNLOGIN statement

Use this statement to assign an alternate login name and password to be used when communicating with a remote server.

Syntax

CREATE EXTERNLOGIN *login-name*
TO *remote-server*
[**REMOTE LOGIN** *remote-user* [**IDENTIFIED BY** *remote-password*]]

Parameters

- **login-name** specifies the local user login name. When using integrated logins, the *login-name* is the database user to which the Windows user or group is mapped.
- **TO clause** The TO clause specifies the name of the remote server.
- **REMOTE LOGIN clause** The REMOTE LOGIN clause specifies the user account on *remote-server* for the local user *login-name*.
- **IDENTIFIED BY clause** The IDENTIFIED BY clause specifies the *remote-password* for *remote-user*. The *remote-user* and *remote-password* combination must be valid on the remote-server.

  If you omit the IDENTIFIED BY clause, the password is sent to the remote server as NULL. However, if you specify IDENTIFIED BY "" (an empty string), then the password sent is the empty string.

Remarks

By default, SQL Anywhere uses the names and passwords of its clients whenever it connects to a remote server on behalf of those clients. CREATE EXTERNLOGIN assigns an alternate login name and password to be used when communicating with a remote server.

The REMOTE LOGIN clause is required only when the remote server requires a user ID and password for the connection. Having an external login without a remote login allows the DBA to control who can access the remote server and tells the remote access layer that logging in to the remote server does not require a user ID and password. For example, the directory access server class requires an external login for restricting access to the directory server, but remote login is not needed because the directory server does not perform user ID and password validation.

The password is stored internally in encrypted form. The *remote-server* must be known to the local server by an entry in the ISYSSERVER table. See "CREATE SERVER statement" on page 506.

Sites with automatic password expiration should plan for periodic updates of passwords for external logins.

CREATE EXTERNLOGIN cannot be used from within a transaction.

Permissions

Only users with DBA authority can add or modify an external login for *login-name*.

Not supported on Windows Mobile.

Side effects

Automatic commit.

See also

- "DROP EXTERNLOGIN statement" on page 574

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

Map the local user named DBA to the user sa with password Plankton when connecting to the server sybase1.

```
CREATE EXTERNLOGIN DBA
TO sybase1
REMOTE LOGIN sa
IDENTIFIED BY Plankton;
```

CREATE FUNCTION statement

Use this statement to create a user-defined SQL function in the database. To create external function interfaces, see "CREATE FUNCTION statement (external procedures)" on page 462. To create web services functions, see "CREATE FUNCTION statement (web services)" on page 467.

CREATE [**OR REPLACE** | **TEMPORARY**] **FUNCTION** [*owner*.]*function-name*
([*parameter*, ...] **)**

RETURNS *data-type*
[**SQL SECURITY** { **INVOKER** | **DEFINER** }]
[**ON EXCEPTION RESUME**]
[[**NOT**] **DETERMINISTIC**]
compound-statement | **AS** *tsql-compound-statement*

parameter :
  [**IN**] *parameter-name data-type* [**DEFAULT** *expression*]

tsql-compound-statement:
sql-statement
sql-statement
 ...

Parameters

- **OR REPLACE clause** Specifying CREATE OR REPLACE FUNCTION creates a new function, or replaces an existing function with the same name. In the case of replacing, this clause changes the definition of the function, but preserves existing permissions.

  You cannot use the OR REPLACE clause with temporary functions.

- **TEMPORARY keyword** Specifying CREATE TEMPORARY FUNCTION means that the function is visible only by the connection that created it, and that it is automatically dropped when the connection is dropped. Temporary functions can also be explicitly dropped. You cannot perform ALTER, GRANT, or REVOKE on them, and, unlike other functions, temporary functions are not recorded in the catalog or transaction log.

  Temporary functions execute with the permissions of their creator (current user) or specified owner. You can specify an owner for a temporary function when:

  ○ the temporary function is created within a permanent stored procedure

  ○ the owner of the temporary function and permanent stored procedure is the same

  To drop the owner of a temporary function, you must drop the temporary function first.

  Temporary functions can be created and dropped when connected to a read-only database.

  You cannot use the OR REPLACE clause with temporary functions.

- **SQL SECURITY clause** The SQL SECURITY clause defines whether the function is executed as the INVOKER (the user who is calling the function), or as the DEFINER (the user who owns the function). The default is DEFINER.

- **compound-statement** A set of SQL statements bracketed by BEGIN and END, and separated by semicolons. See “BEGIN statement” on page 413.

- **tsql-compound-statement** A batch of Transact-SQL statements. See “Transact-SQL batch overview” [*SQL Anywhere Server - SQL Usage*], and “CREATE PROCEDURE statement [T-SQL]” on page 486.

- **ON EXCEPTION RESUME clause** Use Transact-SQL-like error handling. See “CREATE PROCEDURE statement” on page 481.

- **[NOT] DETERMINISTIC clause** Use this clause to indicate whether functions are deterministic or non-deterministic. If this clause is omitted, then the deterministic behavior of the function is unspecified (the default).

  If a function is declared as DETERMINISTIC, it should return the same value every time it is invoked with the same set of parameters.

  If a function is declared as NOT DETERMINISTIC, then it is not guaranteed to return the same value for the same set of parameters. A function declared as NOT DETERMINISTIC is re-evaluated each time it is called in a query. This clause must be used when it is known that the function result for a given set of parameters can vary.

  Also, functions that have side effects such as modifying the underlying data should be declared as NOT DETERMINISTIC. For example, a function that generates primary key values and is used in an INSERT ... SELECT statement should be declared NOT DETERMINISTIC:

  ```
  CREATE FUNCTION keygen( increment INTEGER )
  RETURNS INTEGER
  NOT DETERMINISTIC
  BEGIN
    DECLARE keyval INTEGER;
    UPDATE counter SET x = x + increment;
    SELECT counter.x INTO keyval FROM counter;
    RETURN keyval
  END
  INSERT INTO new_table
  SELECT keygen(1), ...
  FROM old_table;
  ```

  Functions can be declared as DETERMINISTIC if they always return the same value for given input parameters.

Remarks

The CREATE FUNCTION statement creates a function in the database. A function can be created for another user by specifying an owner name. Subject to permissions, a function can be used in exactly the same way as other non-aggregate functions.

Parameter names must conform to the rules for database identifiers. They must have a valid SQL data type, and must be prefixed by the keyword IN, signifying that the argument is an expression that provides a value to the function.

When functions are executed, not all parameters need to be specified. If a default value is provided in the CREATE FUNCTION statement, missing parameters are assigned the default values. If an argument is not provided by the caller and no default is set, an error is given.

When SQL SECURITY INVOKER is specified, more memory is used because annotation must be done for each user that calls the procedure. Also, when SQL SECURITY INVOKER is specified, name resolution is done as the invoker as well. Therefore, care should be taken to qualify all object names (tables, procedures, and so on) with their appropriate owner.

All functions are treated as deterministic unless they are declared NOT DETERMINISTIC. Deterministic functions return a consistent result for the same parameters, and are free of side effects. That is, the database server assumes that two successive calls to the same function with the same parameters returns the same result, and does not have any unwanted side-effects on the query's semantics.

If a function returns a result set, it cannot also set output parameters or return a return value.

Permissions

Must have RESOURCE authority, unless creating a temporary function.

External functions, including Java functions, must have DBA authority.

Side effects

Automatic commit.

See also

- "ALTER FUNCTION statement" on page 369
- "CREATE FUNCTION statement (external procedures)" on page 462
- "CREATE FUNCTION statement (web services)" on page 467
- "BEGIN statement" on page 413
- "CREATE PROCEDURE statement" on page 481
- "DROP FUNCTION statement" on page 575
- "RETURN statement" on page 717
- "Using procedures, triggers, and batches" [*SQL Anywhere Server - SQL Usage*]

Standards and compatibility

- **SQL/2003** Persistent Stored Module feature.

Examples

The following function concatenates a firstname string and a lastname string.

```
CREATE FUNCTION fullname(
   firstname CHAR(30),
   lastname CHAR(30) )
RETURNS CHAR(61)
BEGIN
   DECLARE name CHAR(61);
   SET name = firstname || ' ' || lastname;
   RETURN (name);
END;
```

The following example replaces the fullname function created in the first example. After replacing the function, the local variable `name` is removed:

```
CREATE OR REPLACE FUNCTION fullname(
   firstname CHAR(30),
   lastname CHAR(30) )
RETURNS CHAR(61)
BEGIN
   RETURN = firstname || ' ' || lastname;
END;
```

The following examples illustrate the use of the fullname function.

Return a full name from two supplied strings:

```
SELECT fullname ( 'joe', 'smith' );
```

fullname('joe', 'smith')
joe smith

List the names of all employees:

```
SELECT fullname ( GivenName, Surname )
FROM Employees;
```

fullname (GivenName, Surname)
Fran Whitney
Matthew Cobb
Philip Chin
Julie Jordan
...

The following function uses Transact-SQL syntax:

```
CREATE FUNCTION DoubleIt( @Input INT )
RETURNS INT
AS
BEGIN
  DECLARE @Result INT
  SELECT @Result = @Input * 2
  RETURN @Result
END;
```

The statement `SELECT DoubleIt( 5 )` returns a value of `10`.

CREATE FUNCTION statement (external procedures)

Use this statement to create an interface to a native or external function. To create a user-defined SQL function, see "CREATE FUNCTION statement" on page 458.

Syntax

CREATE [**OR REPLACE**] **FUNCTION** [*owner*.]*function-name*
([*parameter*, ...] **)**
RETURNS *data-type*
[**SQL SECURITY** { **INVOKER** | **DEFINER** }]
[[**NOT**] **DETERMINISTIC**]
EXTERNAL NAME *external-call* [**LANGUAGE** *environment-name*]

parameter :
  [**IN**] *parameter-name data-type* [**DEFAULT** *expression*]

environment-name :
  C_ESQL32
| **C_ESQL64**
| **C_ODBC32**
| **C_ODBC64**
| **CLR**
| **JAVA**
| **PERL**
| **PHP**

Parameters

- **CREATE FUNCTION** You can create permanent stored functions that call external or native functions written in a variety of programming languages.

  Parameter names must conform to the rules for other database identifiers such as column names. They must be a valid SQL data type. For a list of valid data types, see "SQL data types" on page 77.

  Parameters can be prefixed with the keyword IN. However, function parameters are IN by default.

  - **IN** The parameter is an expression that provides a value to the function.

  When functions are executed, not all parameters need to be specified. If a default value is provided in the CREATE FUNCTION statement, missing parameters are assigned the default values. If an argument is not provided when the function is executed, and no default is set, an error is given.

  Specifying OR REPLACE (CREATE OR REPLACE FUNCTION) creates a new function, or replaces an existing function with the same name. This clause changes the definition of the function, but preserves existing permissions.

  The EXTERNAL NAME clause is not supported for TEMPORARY functions.

- **[NOT] DETERMINISTIC clause** Use this clause to indicate whether functions are deterministic or non-deterministic. If this clause is omitted, then the deterministic behavior of the function is unspecified (the default).

  If a function is declared as DETERMINISTIC, it should return the same value every time it is invoked with the same set of parameters.

  If a function is declared as NOT DETERMINISTIC, then it is not guaranteed to return the same value for the same set of parameters. A function declared as NOT DETERMINISTIC is re-evaluated each time it is called in a query. This clause must be used when it is known that the function result for a given set of parameters can vary.

  Also, functions that have side effects such as modifying the underlying data should be declared as NOT DETERMINISTIC. For example, a function that generates primary key values and is used in an INSERT ... SELECT statement should be declared NOT DETERMINISTIC:

  ```
  CREATE FUNCTION keygen( increment INTEGER )
  RETURNS INTEGER
  NOT DETERMINISTIC
  BEGIN
    DECLARE keyval INTEGER;
    UPDATE counter SET x = x + increment;
    SELECT counter.x INTO keyval FROM counter;
    RETURN keyval
  END
  INSERT INTO new_table
  ```

```
SELECT keygen(1), ...
FROM old_table;
```

Functions can be declared as DETERMINISTIC if they always return the same value for given input parameters.

- **SQL SECURITY clause** The SQL SECURITY clause defines whether the function is executed as the INVOKER (the user who is calling the function), or as the DEFINER (the user who owns the function). The default is DEFINER. For external calls, this clause establishes the ownership context for unqualified object references in the external environment.

  When SQL SECURITY INVOKER is specified, more memory is used because annotation must be done for each user that calls the function. Also, when SQL SECURITY INVOKER is specified, name resolution is done as the invoker as well. Therefore, care should be taken to qualify all object names (tables, procedures, and so on) with their appropriate owner. For example, suppose user1 creates the following function:

  ```
  CREATE FUNCTION user1.myFunc()
     RETURNS INT
     SQL SECURITY INVOKER
     BEGIN
       DECLARE res INT;
       SELECT COUNT(*) INTO res FROM table1;
       RETURN res;
     END;
  ```

  If user2 attempts to run this function and a table user2.table1 *does not* exist, a table lookup error results. Additionally, if a user2.table1 *does* exist, that table is used instead of the intended user1.table1. To prevent this situation, qualify the table reference in the statement (user1.table1, instead of just table1).

- **EXTERNAL NAME *native-call* clause**

  native-call :
  [*operating-system*:]*function-name*@*library*; ...

  operating-system : **Unix**
  A function using the EXTERNAL NAME clause with no LANGUAGE attribute defines an interface to a native function written in a programming language such as C. The native function is loaded by the database server into its address space.

  The *library* name can include the file extension, which is typically *.dll* on Windows and *.so* on Unix. In the absence of the extension, the software appends the platform-specific default file extension for libraries. The following is a formal example.

  ```
  CREATE FUNCTION mystring( IN instr LONG VARCHAR )
  RETURNS LONG VARCHAR
  EXTERNAL NAME 'mystring@mylib.dll;Unix:mystring@mylib.so';
  ```

  A simpler way to write the above EXTERNAL NAME clause, using platform-specific defaults, is as follows:

  ```
  CREATE FUNCTION mystring( IN instr LONG VARCHAR )
  RETURNS LONG VARCHAR
  EXTERNAL NAME 'mystring@mylib';
  ```

  When called, the library containing the function is loaded into the address space of the database server. The native function will execute as part of the server. In this case, if the function causes a fault, then the database server will be terminated. Because of this, loading and executing functions in an external environment using the LANGUAGE attribute is recommended. If a function causes a fault in an external environment, the database server will continue to run.

For information about native library calls, see "Calling external libraries from procedures" [*SQL Anywhere Server - Programming*].

- **EXTERNAL NAME *c-call* LANGUAGE {C_ESQL32 | C_ESQL64 | C_ODBC32 | C_ODBC64 } clause** To call a compiled native C function in an external environment instead of within the database server, the stored procedure or function is defined with the EXTERNAL NAME clause followed by the LANGUAGE attribute specifying one of C_ESQL32, C_ESQL64, C_ODBC32, or C_ODBC64.

  When the LANGUAGE attribute is specified, then the library containing the function is loaded by an external process and the external function will execute as part of that external process. In this case, if the function causes a fault, then the database server will continue to run.

  The following is a sample function definition.

  ```
  CREATE FUNCTION ODBCinsert(
    IN ProductName CHAR(30),
    IN ProductDescription CHAR(50)
  )
  RETURNS INT
  EXTERNAL NAME 'ODBCexternalInsert@extodbc.dll'
  LANGUAGE C_ODBC32;
  ```

  For more information, see "The ESQL and ODBC external environments" [*SQL Anywhere Server - Programming*].

- **EXTERNAL NAME *clr-call* LANGUAGE CLR clause** To call a .NET function in an external environment, the function interface is defined with an EXTERNAL NAME clause followed by the LANGUAGE CLR attribute.

  A CLR stored procedure or function behaves the same as a SQL stored procedure or function with the exception that the code for the procedure or function is written in a .NET language such as C# or Visual Basic, and the execution of the procedure or function takes place outside the database server (that is, within a separate .NET executable).

  The following is a sample function definition.

  ```
  CREATE FUNCTION clr_interface(
      IN p1 INT,
      IN p2 UNSIGNED SMALLINT,
      IN p3 LONG VARCHAR)
  RETURNS INT
  EXTERNAL NAME 'CLRlib.dll::CLRproc.Run( int, ushort, string )'
  LANGUAGE CLR;
  ```

  For more information, see "The CLR external environment" [*SQL Anywhere Server - Programming*].

- **EXTERNAL NAME *perl-call* LANGUAGE PERL clause** To call a Perl function in an external environment, the function interface is defined with an EXTERNAL NAME clause followed by the LANGUAGE PERL attribute.

  A Perl stored procedure or function behaves the same as a SQL stored procedure or function with the exception that the code for the procedure or function is written in Perl and the execution of the procedure or function takes place outside the database server (that is, within a Perl executable instance).

The following is a sample function definition.

```
CREATE FUNCTION PerlWriteToConsole( IN str LONG VARCHAR)
RETURNS INT
EXTERNAL NAME '<file=PerlConsoleExample>
    WriteToServerConsole( $sa_perl_arg0 )'
LANGUAGE PERL;
```

For more information, see "The PERL external environment" [*SQL Anywhere Server - Programming*].

- **EXTERNAL NAME *php-call* LANGUAGE PHP clause**   To call a PHP function in an external environment, the function interface is defined with an EXTERNAL NAME clause followed by the LANGUAGE PHP attribute.

  A PHP stored procedure or function behaves the same as a SQL stored procedure or function with the exception that the code for the procedure or function is written in PHP and the execution of the procedure or function takes place outside the database server (that is, within a PHP executable instance).

  The following is a sample function definition.

  ```
  CREATE FUNCTION PHPPopulateTable()
  RETURNS INT
  EXTERNAL NAME '<file=ServerSidePHPExample> ServerSidePHPSub()'
  LANGUAGE PHP;
  ```

  For more information, see "The PHP external environment" [*SQL Anywhere Server - Programming*].

- **EXTERNAL NAME *java-call* LANGUAGE JAVA clause**   To call a Java method in an external environment, the function interface is defined with an EXTERNAL NAME clause followed by the LANGUAGE JAVA attribute.

  A Java-interfacing stored procedure or function behaves the same as a SQL stored procedure or function with the exception that the code for the procedure or function is written in Java and the execution of the procedure or function takes place outside the database server (that is, within a Java Virtual Machine).

  The following is a sample function definition.

  ```
  CREATE FUNCTION HelloDemo( IN name LONG VARCHAR )
  RETURNS INT
  EXTERNAL NAME 'Hello.main([Ljava/lang/String;)V'
  LANGUAGE JAVA;
  ```

  For more information, see "The Java external environment" [*SQL Anywhere Server - Programming*].

Remarks

The CREATE FUNCTION statement creates a function in the database. Users with DBA authority can create functions for other users by specifying an owner. A function is invoked as part of a SQL expression.

When referencing a temporary table from multiple functions, a potential issue can arise if the temporary table definitions are inconsistent and statements referencing the table are cached. See "Referencing temporary tables within procedures" [*SQL Anywhere Server - SQL Usage*].

Permissions

Must have RESOURCE authority, unless creating a temporary function.

Must have DBA authority for external functions or to create a function for another user.

Side effects

Automatic commit.

See also

- "ALTER FUNCTION statement" on page 369
- "CALL statement" on page 419
- "CREATE FUNCTION statement" on page 458
- "CREATE FUNCTION statement (web services)" on page 467
- "CREATE PROCEDURE statement (external procedures)" on page 488
- "DROP FUNCTION statement" on page 575
- "GRANT statement" on page 627
- "Overview of external environments" [*SQL Anywhere Server - Programming*]

Standards and compatibility

- **SQL/2003** Persistent Stored Module feature. The syntax extensions for Java result sets are as specified in the optional J621 feature.

CREATE FUNCTION statement (web services)

Use this statement to create a new web services function in the database. To create a user-defined SQL function, see "CREATE FUNCTION statement" on page 458.

```
CREATE [ OR REPLACE ] FUNCTION [ owner.]function-name ( [ parameter, ... ] )
RETURNS data-type
URL url-string
[ HEADER header-string ]
[ SOAPHEADER soap-header-string ]
[ TYPE {
  'HTTP[ :{ GET | POST[:MIME-type ] | PUT[:MIME-type ] | DELETE | HEAD } ]' |
  'SOAP[:{ RPC | DOC } ]' } ]
[ NAMESPACE namespace-string ]
[ CERTIFICATE certificate-string ]
[ CLIENTPORT clientport-string ]
[ PROXY proxy-string ]
[ SET protocol-option-string ]

url-string :
' { HTTP | HTTPS | HTTPS_FIPS }://[user:password@]hostname[:port][/path]'

parameter :
  [ IN ] parameter-name data-type [ DEFAULT expression ]
```

Parameters

- **CREATE FUNCTION** Parameter names must conform to the rules for database identifiers. They must have a valid SQL data type, and must be prefixed by the keyword IN, signifying that the argument is an expression that provides a value to the function.

When functions are executed, not all parameters need to be specified. If a default value is provided in the CREATE FUNCTION statement, missing parameters are assigned the default values. If an argument is not provided by the caller and no default is set, an error is given.

Specifying OR REPLACE (CREATE OR REPLACE FUNCTION) creates a new function, or replaces an existing function with the same name. This clause changes the definition of the function, but preserves existing permissions. You cannot use the OR REPLACE clause with temporary functions.

- **RETURNS clause** Return data types can be VARCHAR, BINARY, VARBINARY, or LONG BINARY. The data type does not affect how the HTTP response is processed.
- **URL clause** For use only when defining an HTTP or SOAP web services client function. Specifies the URL of the web service. The optional user name and password parameters provide a means of supplying the credentials needed for HTTP basic authentication. HTTP basic authentication base-64 encodes the user and password information and passes it in the Authentication header of the HTTP request.

  Specifying HTTPS_FIPS forces the system to use the FIPS libraries. If HTTPS_FIPS is specified, but no FIPS libraries are present, non-FIPS libraries are used instead.
- **HEADER clause** When creating HTTP web service client functions, use this clause to add or modify HTTP request header entries. Only printable ASCII characters can be specified for HTTP headers, and they are case-insensitive. For more information about how to use this clause, see the HEADER clause of the "CREATE PROCEDURE statement (web services)" on page 495.

  For more information about using HTTP headers, see "Working with HTTP headers" [*SQL Anywhere Server - Programming*].
- **SOAPHEADER clause** When declaring a SOAP web service as a function, use this clause to specify one or more SOAP request header entries. A SOAP header can be declared as a static constant, or can be dynamically set using the parameter substitution mechanism (declaring IN, OUT, or INOUT parameters for hd1, hd2, and so on). A web service function can define one or more IN mode substitution parameters, but can not define an INOUT or OUT substitution parameter. For more information about how to use this clause, see the SOAPHEADER clause of the "CREATE PROCEDURE statement (web services)" on page 495.

  For more information about using SOAP headers, see "Working with SOAP headers" [*SQL Anywhere Server - Programming*].
- **TYPE clause** Used to specify the format used when making the web service request. If SOAP is specified or no type clause is included, the default type SOAP:RPC is used. HTTP implies HTTP:POST. Since SOAP requests are always sent as XML documents, HTTP:POST is always used to send SOAP requests.
- **NAMESPACE clause** Applies to SOAP client functions only. This clause identifies the method namespace usually required for both SOAP:RPC and SOAP:DOC requests. The SOAP server handling the request uses this namespace to interpret the names of the entities in the SOAP request message body. The namespace can be obtained from the WSDL (Web Services Description Language) of the SOAP service available from the web service server. The default value is the function's URL, up to but not including, the optional path component.
- **CERTIFICATE clause** To make a secure (HTTPS) request, a client must have access to the certificate used by the HTTPS server. The necessary information is specified in a string of semicolon-separated

key/value pairs. The certificate can be placed in a file and the name of the file provided using the file key, or the whole certificate can be placed in a string, but not both. The following keys are available:

Key	Abbreviation	Description
file		The file name of the certificate.
certificate	cert	The certificate itself.
company	co	The company specified in the certificate.
unit		The company unit specified in the certificate.
name		The common name specified in the certificate.

Certificates are required only for requests that are directed to an HTTPS server, or for requests that can be redirected from a non-secure to a secure server.

- **CLIENTPORT clause** Identifies the port number on which the HTTP client function communicates using TCP/IP. It is provided for and recommended only for connections across firewalls, as firewalls filter according to the TCP/UDP port. You can specify a single port number, ranges of port numbers, or a combination of both; for example, CLIENTPORT '85,90-97'. See "ClientPort protocol option [CPORT]" [*SQL Anywhere Server - Database Administration*].
- **PROXY clause** Specifies the URI of a proxy server. For use when the client must access the network through a proxy. this clause indicates that the function is to connect to the proxy server and send the request to the web service through it.

Remarks

The CREATE FUNCTION statement creates a web services function in the database. A function can be created for another user by specifying an owner name.

The return type of SOAP and HTTP functions must be one of the character data types, such as VARCHAR. The value returned is the body of the HTTP response. No HTTP header information is included. If more information is required, such as status information, use a procedure instead of a function.

Parameter values are passed as part of the request. The syntax used depends on the type of request. For HTTP:GET, the parameters are passed as part of the URL; for HTTP:POST requests, the values are placed in the body of the request. Parameters to SOAP requests are always bundled in the request body.

Permissions

RESOURCE authority.

DBA authority for external functions, including Java functions.

Side effects

Automatic commit.

See also

- "ALTER FUNCTION statement" on page 369
- "CREATE FUNCTION statement" on page 458
- "CREATE FUNCTION statement (external procedures)" on page 462
- "CREATE PROCEDURE statement" on page 481
- "CREATE PROCEDURE statement (web services)" on page 495
- "DROP FUNCTION statement" on page 575
- "RETURN statement" on page 717
- "SQL Anywhere web services" [*SQL Anywhere Server - Programming*]
- "Creating web service client functions and procedures" [*SQL Anywhere Server - Programming*]

Standards and compatibility

- **SQL/2003** Persistent Stored Module feature.

Examples

The following statement creates a function named **cli_test1** that returns images from the **get_picture** service running on localhost:

```
CREATE FUNCTION cli_test1( image LONG VARCHAR )
RETURNS LONG BINARY
URL 'http://localhost/get_picture'
TYPE 'HTTP:GET';
```

The following statement issues an HTTP request with the URL *http://localhost/get_picture?image=widget*:

```
SELECT cli_test1( 'widget' );
```

The following statement uses a substitution parameter to allow the request URL to be passed as an input parameter. The SET clause is used to turn off CHUNK mode transfer-encoding.

```
CREATE FUNCTION cli_test2( image LONG VARCHAR, myurl LONG VARCHAR )
RETURNS LONG BINARY
URL '!myurl'
TYPE 'HTTP:GET'
SET 'HTTP(CH=OFF)'
HEADER 'ASA-ID';
```

The following statement issues an HTTP request with the URL *http://localhost/get_picture?image=widget*:

```
CREATE VARIABLE a_binary LONG BINARY
a_binary = cli_test2('widget', 'http://localhost/get_picture');
SELECT a_binary;
```

CREATE INDEX statement

Use this statement to create an index on a specified table or materialized view. Indexes can improve database performance.

Syntax 1 - Creating an index on a table

CREATE [**VIRTUAL**] [**UNIQUE**] [**CLUSTERED**] **INDEX** *index-name*
 ON [*owner*.]*table-name*
  (*column-name* [**ASC** | **DESC**], ...
   | *function-name* **(** *argument*, ...] **)** **AS** *column-name* **)**
 [{ **IN** | **ON** } *dbspace-name*]
 [**FOR OLAP WORKLOAD**]

Syntax 2 - Creating an index on a materialized view

CREATE [**VIRTUAL**] [**UNIQUE**] [**CLUSTERED**] **INDEX** *index-name*
 ON [*owner*.]*materialized-view-name*
  (*column-name* [**ASC** | **DESC**], ...**)**
 [{ **IN** | **ON** } *dbspace-name*]
 [**FOR OLAP WORKLOAD**]

Parameters

- **VIRTUAL clause** The VIRTUAL keyword is primarily for use by the Index Consultant. A virtual index mimics the properties of a real physical index during the evaluation of execution plans by the Index Consultant and when the PLAN function is used. You can use virtual indexes together with the PLAN function to explore the performance impact of an index, without the often time-consuming and resource-consuming effects of creating a real index.

  Virtual indexes are not visible to other connections, and are dropped when the connection is closed. Virtual indexes are not used when evaluating plans for the actual execution of queries, and so do not interfere with performance.

  Virtual indexes have a limit of four columns.

  See "Obtain Index Consultant recommendations for a query" [*SQL Anywhere Server - SQL Usage*], and "Index Consultant" [*SQL Anywhere Server - SQL Usage*].

- **CLUSTERED clause** The CLUSTERED attribute causes rows to be stored in an approximate key order corresponding to the index. While the database server makes an attempt to preserve key order, total clustering is not guaranteed.

  If a clustered index exists, the LOAD TABLE statement inserts rows in the order of the index key, and the INSERT statement attempts to put new rows on the same page as the one containing adjacent rows, as defined by the key order.

  See "Using clustered indexes" [*SQL Anywhere Server - SQL Usage*].

- **UNIQUE clause** The UNIQUE attribute ensures that there will not be two rows in the table or materialized view with identical values in all the columns in the index. Each index key must be unique or contain a NULL in at least one column.

  There is a difference between a unique constraint and a unique index. Columns of a unique index are allowed to be NULL, while columns in a unique constraint are not. A foreign key can reference either a primary key or a unique constraint, but not a unique index, because it can include multiple instances of NULL.

  It is recommended that you do not use approximate data types such as FLOAT and DOUBLE for primary keys or for columns in unique constraints. Approximate numeric data types are subject to rounding errors after arithmetic operations.

- **ASC | DESC clause** Columns are sorted in ascending (increasing) order unless descending (DESC) is explicitly specified. An index is used for both an ascending and a descending ORDER BY, whether the index was ascending or descending. However, if an ORDER BY is performed with mixed ascending and descending attributes, an index is used only if the index was created with the same ascending and descending attributes.
- **function-name** The function-name clause creates an index on a function. This clause cannot be used on declared temporary tables or materialized views.

  This form of the CREATE INDEX statement is a convenience method that carries out the following operations:

  1. Adds a computed column named *column-name* to the table. The column is defined with a COMPUTE clause that is the specified function, along with any specified arguments. See the COMPUTE clause of the CREATE TABLE statement for restrictions on the type of function that can be specified. The data type of the column is based on the result type of the function.
  2. Populates the computed column for the existing rows in the table.
  3. Creates an index on the column.

     Dropping the index does not cause the associated computed column to be dropped.

     For more information about computed columns, see "Working with computed columns" [*SQL Anywhere Server - SQL Usage*].
- **IN | ON clause** By default, the index is placed in the same database file as its table or materialized view. You can place the index in a separate database file by specifying a dbspace name in which to put the index. This feature is useful mainly for large databases to circumvent file size limitations, or for performance improvements that might be achieved by using multiple disk devices.

  If the new index can share the physical index with an existing logical index, the IN clause is ignored.

  For more information about limitations, see "SQL Anywhere size and number limitations" [*SQL Anywhere Server - Database Administration*].
- **FOR OLAP WORKLOAD clause** When you specify FOR OLAP WORKLOAD, the database server performs certain optimizations and gathers statistics on the key to help improve performance for OLAP workloads. Performance improvements are most noticeable when the optimization_workload is set to OLAP. See "optimization_workload option [database]" [*SQL Anywhere Server - Database Administration*].

  For more information about OLAP, see "OLAP support" [*SQL Anywhere Server - SQL Usage*].

Remarks

Syntax 1 is for use with tables; Syntax 2 is for use with materialized views.

SQL Anywhere uses physical and logical indexes. A physical index is the actual indexing structure as it is stored on disk. A logical index is a reference to a physical index. If you create an index that is identical in its physical attributes to an existing index, the database server creates a logical index that shares the existing physical index. In general, indexes created by you are considered logical indexes. The database server creates physical indexes as required to implement logical indexes, and can share the same physical index among several logical indexes. See "Index sharing using logical indexes" [*SQL Anywhere Server - SQL Usage*].

The CREATE INDEX statement creates a sorted index on the specified columns of the named table or materialized view. Indexes are automatically used to improve the performance of queries issued to the database, and to sort queries with an ORDER BY clause. Once an index is created, it is never referenced in a SQL statement again except to validate it (VALIDATE INDEX), alter it (ALTER INDEX), delete it (DROP INDEX), or in a hint to the optimizer.

- **Index ownership** There is no way of specifying the index owner in the CREATE INDEX statement. Indexes are always owned by the owner of the table or materialized view.
- **Indexes on views** You can create indexes on materialized views, but not on regular views.
- **Index name space** The name of each index must be unique for a given table or materialized view.
- **Exclusive use** CREATE INDEX is prevented whenever the statement affects a table or materialized view currently being used by another connection. CREATE INDEX can be time consuming and the database server will not process requests referencing the same table while the statement is being processed.
- **Automatically created indexes** SQL Anywhere automatically creates indexes for primary key, foreign key, and unique constraints. These automatically created indexes are held in the same database file as the table.

This statement cannot be executed when there are cursors opened with the WITH HOLD clause that use either statement or transaction snapshots. See "Snapshot isolation" [*SQL Anywhere Server - SQL Usage*].

Permissions

Must be the owner of the table or materialized view, or have either DBA authority or REFERENCES permission.

Side effects

Automatic commit. Creating an index on a built-in function also causes a checkpoint.

Column statistics are updated (or created if they do not exist).

See also

- "DROP INDEX statement" on page 576
- "Indexes" [*SQL Anywhere Server - SQL Usage*]
- "CREATE STATISTICS statement" on page 516
- "Index sharing using logical indexes" [*SQL Anywhere Server - SQL Usage*]
- "Indexes" [*SQL Anywhere Server - SQL Usage*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

Create a two-column index on the Employees table.

```
CREATE INDEX employee_name_index
ON Employees
( Surname, GivenName );
```

Create an index on the SalesOrderItems table for the ProductID column.

```
CREATE INDEX item_prod
ON SalesOrderItems
( ProductID );
```

Use the SORTKEY function to create an index on the Description column of the Products table, sorted according to a Russian collation. As a side effect, the statement adds a computed column desc_ru to the table.

```
CREATE INDEX ix_desc_ru
ON Products (
 SORTKEY( Description, 'rusdict' )
 AS desc_ru );
```

CREATE LOCAL TEMPORARY TABLE statement

Use this statement within a procedure to create a local temporary table that persists after the procedure completes and until it is either explicitly dropped, or until the connection terminates.

Syntax

CREATE LOCAL TEMPORARY TABLE *table-name*
({ *column-definition* [*column-constraint* ...] | *table-constraint* | *pctfree* }, ... **)**
[**ON COMMIT** { **DELETE** | **PRESERVE** } **ROWS** | **NOT TRANSACTIONAL**]

pctfree : **PCTFREE** *percent-free-space*

percent-free-space : *integer*

Parameters

For definitions of *column-definition*, *column-constraint*, *table-constraint*, and *pctfree*, see "CREATE TABLE statement" on page 523.

- **ON COMMIT clause** By default, the rows of a temporary table are deleted on a COMMIT. You can use the ON COMMIT clause to preserve rows on a COMMIT.
- **NOT TRANSACTIONAL clause** The NOT TRANSACTIONAL clause provides performance improvements in some circumstances because operations on non-transactional temporary tables do not cause entries to be made in the rollback log. For example, NOT TRANSACTIONAL may be useful if procedures that use the temporary table are called repeatedly with no intervening COMMITs or ROLLBACKs.

Remarks

In a procedure, use the CREATE LOCAL TEMPORARY TABLE statement, instead of the DECLARE LOCAL TEMPORARY TABLE statement, when you want to create a table that persists after the procedure completes. Local temporary tables created using the CREATE LOCAL TEMPORARY TABLE statement remain until they are either explicitly dropped, or until the connection closes.

Local temporary tables created in IF statements using CREATE LOCAL TEMPORARY TABLE also persist after the IF statement completes.

Permissions

None.

Side effects

None.

See also

- "CREATE TABLE statement" on page 523
- "DECLARE LOCAL TEMPORARY TABLE statement" on page 557
- "Using compound statements" [*SQL Anywhere Server - SQL Usage*]

Standards and compatibility

- **SQL/2003** SQL/foundation feature outside core SQL.

Example

The following example creates a local temporary table called TempTab:

```
CREATE LOCAL TEMPORARY TABLE TempTab ( number INT )
ON COMMIT PRESERVE ROWS;
```

CREATE LOGIN POLICY statement

Use this statement to create a login policy.

Syntax

CREATE LOGIN POLICY *policy-name policy-options*

policy options :
policy-option [*policy-option* ...]

policy-option :
policy-option-name **=** *policy-option-value*

policy-option-value :
{ **UNLIMITED** | *legal-option-value* }

Parameters

- **policy_name** The name of the login policy.
- **policy_option_name** The name of the login policy option. If you do not specify an option, the value from the root login policy is applied.
- **policy_option_value** The value assigned to the login policy option. If you specify UNLIMITED, no limits are imposed. To view a list of default login policy values, see Remarks.

Remarks

If you do not specify a policy option, values for the login policy are taken from the root login policy.

All new databases include a root login policy. You can modify the root login policy values, but you cannot delete the policy. An overview of the default options for the root login policy is provided in the table below.

Policy_option_name	Description	Default value	Applies to
password_life_time	The maximum number of days before a password must be changed.	Unlimited	All users including those with DBA authority
password_grace_time	The number of days before the password expires during which login is allowed, but the default post_login procedure issues warnings.	0	All users including those with DBA authority
password_expiry_on_next_login	If the value for this option is ON, the user's password will expire after the next login.	OFF	All users including those with DBA authority
locked	If the value for this option is ON, users are prohibited from establishing new connections.	OFF	Only to users without DBA authority
max_connections	The maximum number of concurrent connections allowed for a user.	Unlimited	Only to users without DBA authority
max_failed_login_attempts	The maximum number of failed attempts, since the last successful attempt, to login to the user account before the account is locked.	Unlimited	Only to users without DBA authority
max_days_since_login	The maximum number of days that can elapse between two successive logins by the same user.	Unlimited	Only to users without DBA authority
max_non_dba_connections	The maximum number of concurrent connections that a user without DBA authority can make. This option is only supported in the DEFAULT login policy.	Unlimited	Only to users without DBA authority. Only to the default login policy.

Permissions

Must have DBA authority.

Side effects

None.

See also

- "ALTER LOGIN POLICY statement" on page 372
- "ALTER USER statement" on page 402
- "COMMENT statement" on page 425
- "CREATE USER statement" on page 546
- "DROP LOGIN POLICY statement" on page 577
- "DROP USER statement" on page 591
- "Managing login policies overview" [*SQL Anywhere Server - Database Administration*]
- "Creating a new login policy" [*SQL Anywhere Server - Database Administration*]
- "Assigning a login policy to an existing user" [*SQL Anywhere Server - Database Administration*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Examples

The following example creates the Test1 login policy. This example has an unlimited password life and allows the user a maximum of 5 attempts to enter a correct password before the account is locked.

```
CREATE LOGIN POLICY Test1
password_life_time=UNLIMITED
max_failed_login_attempts=5;
```

CREATE MATERIALIZED VIEW statement

Use this statement to create a materialized view.

Syntax

CREATE MATERIALIZED VIEW
[*owner*.]*materialized-view-name* [**(** *alt-column-names*, ... **)**]
[**IN** *dbspace-name*]
AS *select-statement*
[**CHECK** { **IMMEDIATE** | **MANUAL** } **REFRESH**]

alt-column-names :
(*column-name* [,...] **)**

Parameters

- **alt-column-names** Use this clause to specify alternate names for the columns in the materialized view. If you specify alternate columns names, the number of columns listed in *alt-column-names* must match the number of columns in *select-statement*. If you do not specify alternate column names, the names are set to those in *select-statement*.

- **IN clause** Use this clause to specify the dbspace in which to create the materialized view. If this clause is not specified, then the materialized view is created in the dbspace specified by the default_dbspace option. Otherwise, the system dbspace is used. For more information, see "Using additional dbspaces" [*SQL Anywhere Server - Database Administration*].
- **AS clause** Use this clause to specify, in the form of a SELECT statement, the data to use to populate the materialized view. A materialized view definition can only reference base tables, it cannot reference views, other materialized views, or temporary tables. *select-statement* must contain column names or have an alias name specified. If you specify *alt-column-names*, those names are used instead of the aliases specified in *select-statement*.

  Column names in the SELECT statement must be specified explicitly; you cannot use the `SELECT *` construct. For example, you cannot specify `CREATE MATERIALIZED VIEW matview AS SELECT * FROM table-name`. Also, you should fully qualify objects names in the *select-statement*. See "Restrictions on materialized views" [*SQL Anywhere Server - SQL Usage*].
- **CHECK clause** Use this clause to validate the statement without actually creating the view. When you specify the CHECK clause:
  - The database server performs the normal language checks that would be carried out if CREATE MATERIALIZED VIEW was executed without the clause, and any errors generated are returned as usual.
  - The database server does not carry out the actual creation of the view. This means that certain errors that would occur at creation time are not generated. For example, an error indicating that the specified view name already exists is not generated. This allows you to use the CHECK clause to test intended changes to the definition of the view, without a conflict with the naming of the view.
  - If CHECK IMMEDIATE REFRESH is used then the database server verifies that the syntax is valid for an immediate view and returns any errors.
  - No changes are made to the database, and nothing is recorded in the transaction log.
  - There is an implicit commit at the beginning of statement execution and a rollback at the end to release all locks obtained during execution.

Remarks

When you create a materialized view, it is a manual view and uninitialized. That is, it has a manual refresh type, and it has not been refreshed (populated with data). To initialize the view, execute a REFRESH MATERIALIZED VIEW statement, or use the sa_refresh_materialized_views system procedure. See "REFRESH MATERIALIZED VIEW statement" on page 703, and "sa_refresh_materialized_views system procedure" on page 926.

You can encrypt a materialized view, change its PCTFREE setting, change its refresh type, and enable or disable its use by the optimizer. However, you must create the materialized view first, and then use the ALTER MATERIALIZED VIEW to change these settings. The default values for materialized views at creation time are:

- NOT ENCRYPTED
- ENABLE USE IN OPTIMIZATION

- PCTFREE is set according to the database page size: 200 bytes for a 4 KB page size, and 100 bytes for a 2 KB page size.
- MANUAL REFRESH

Several database and server options must be in effect to create a materialized view. See "Restrictions on materialized views" [*SQL Anywhere Server - SQL Usage*].

The sa_recompile_views system procedure does not affect materialized views.

Permissions

You must have RESOURCE authority and SELECT permission on the tables in the materialized view definition. To create a materialized view for another user, you must also have DBA authority.

Side effects

While executing, the CREATE MATERIALIZED VIEW statement places exclusive locks, without blocking, on all tables referenced by the materialized view. If one of the referenced tables cannot be locked, the statement fails and an error is returned.

See also

- "Working with materialized views" [*SQL Anywhere Server - SQL Usage*]
- "Materialized view statuses and properties" [*SQL Anywhere Server - SQL Usage*]
- "ALTER MATERIALIZED VIEW statement" on page 373
- "DROP MATERIALIZED VIEW statement" on page 578
- "REFRESH MATERIALIZED VIEW statement" on page 703
- "CREATE VIEW statement" on page 547
- "sa_refresh_materialized_views system procedure" on page 926

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following example creates a materialized view containing confidential information about employees in the SQL Anywhere sample database. You must subsequently execute a REFRESH MATERIALIZED VIEW statement to initialize the view for use, as shown in the example.

```
CREATE MATERIALIZED VIEW EmployeeConfid2 AS
SELECT EmployeeID, Employees.DepartmentID,
    SocialSecurityNumber, Salary, ManagerID,
    Departments.DepartmentName, Departments.DepartmentHeadID
FROM Employees, Departments
WHERE Employees.DepartmentID=Departments.DepartmentID;
REFRESH MATERIALIZED VIEW EmployeeConfid2;
```

CREATE MESSAGE statement [T-SQL]

Use this statement to add a user-defined message to the ISYSUSERMESSAGE system table for use by PRINT and RAISERROR statements.

Syntax

CREATE MESSAGE *message-number* **AS** *message-text*

message-number : *integer*

message-text : *string*

Parameters

- **message_number** The message number of the message to add. The message number for a user-defined message must be 20000 or greater.
- **message_text** The text of the message to add. The maximum length is 255 bytes. PRINT and RAISERROR recognize placeholders in the message text. A single message can contain up to 20 unique placeholders in any order. These placeholders are replaced with the formatted contents of any arguments that follow the message when the text of the message is sent to the client.

  The placeholders are numbered to allow reordering of the arguments when translating a message to a language with a different grammatical structure. A placeholder for an argument appears as "%nn!": a percent sign (%), followed by an integer from 1 to 20, followed by an exclamation mark (!), where the integer represents the position of the argument in the argument list. "%1!" is the first argument, "%2!" is the second argument, and so on.

  There is no parameter corresponding to the *language* argument for sp_addmessage.

Remarks

CREATE MESSAGE associates a message number with a message string. The message number can be used in PRINT and RAISERROR statements.

To drop a message, see "DROP MESSAGE statement" on page 579.

Permissions

Must have RESOURCE authority

Side effects

Automatic commit.

See also

- "PRINT statement [T-SQL]" on page 697
- "RAISERROR statement" on page 699
- "ISYSUSERMESSAGE system table" on page 809

Standards and compatibility

- **SQL/2003** Vendor extension.

CREATE PROCEDURE statement

Use this statement to create a user-defined SQL procedure in the database. To create external procedure interfaces, see "CREATE PROCEDURE statement (external procedures)" on page 488. To create web services procedures, see "CREATE PROCEDURE statement (web services)" on page 495.

Syntax

CREATE [**OR REPLACE** | **TEMPORARY**] **PROCEDURE** [*owner*.]*procedure-name*
([*parameter*, ...] **)**
[**RESULT (** *result-column*, ... **)** | **NO RESULT SET**]
[**SQL SECURITY** { **INVOKER** | **DEFINER** }]
[**ON EXCEPTION RESUME**]
compound-statement | **AT** *location-string*

parameter :
  parameter-mode parameter-name data-type [**DEFAULT** *expression*]
| **SQLCODE**
| **SQLSTATE**

parameter-mode : **IN**
| **OUT**
| **INOUT**

result-column : *column-name data-type*

Parameters

- **CREATE PROCEDURE** You can create permanent or temporary (TEMPORARY) stored procedures. You can use PROC as a synonym for PROCEDURE.

  Parameter names must conform to the rules for other database identifiers such as column names. They must be a valid SQL data type. For a list of valid data types, see "SQL data types" on page 77.

  Parameters can be prefixed with one of the keywords IN, OUT, or INOUT. If you do not specify one of these values, parameters are INOUT by default. The keywords have the following meanings:

  - **IN** The parameter is an expression that provides a value to the procedure.
  - **OUT** The parameter is a variable that could be given a value by the procedure.
  - **INOUT** The parameter is a variable that provides a value to the procedure, and could be given a new value by the procedure.

  When procedures are executed using the CALL statement, not all parameters need to be specified. If a default value is provided in the CREATE PROCEDURE statement, missing parameters are assigned the default values. If an argument is not provided in the CALL statement, and no default is set, an error is given.

  SQLSTATE and SQLCODE are special OUT parameters that output the SQLSTATE or SQLCODE value when the procedure ends. The SQLSTATE and SQLCODE special values can be checked immediately after a procedure call to test the return status of the procedure.

  The SQLSTATE and SQLCODE special values are modified by the next SQL statement. Providing SQLSTATE or SQLCODE as procedure arguments allows the return code to be stored in a variable.

Specifying CREATE OR REPLACE PROCEDURE creates a new procedure, or replaces an existing procedure with the same name. This clause changes the definition of the procedure, but preserves existing permissions. You cannot use the OR REPLACE clause with temporary procedures. Also, an error is returned if the procedure being replaced is already in use.

Specifying CREATE TEMPORARY PROCEDURE means that the stored procedure is visible only by the connection that created it, and that it is automatically dropped when the connection is dropped. Temporary stored procedures can also be explicitly dropped. You cannot perform ALTER, GRANT, or REVOKE on them, and, unlike other stored procedures, temporary stored procedures are not recorded in the catalog or transaction log.

Temporary procedures execute with the permissions of their creator (current user), or specified owner. You can specify an owner for a temporary procedure when:

- the temporary procedure is created within a permanent stored procedure
- the owner of the temporary and permanent procedure is the same

To drop the owner of a temporary procedure, you must drop the temporary procedure first.

Temporary stored procedures can be created and dropped when connected to a read-only database, and they cannot be external procedures.

For example, the following temporary procedure drops the table called CustRank, if it exists. For this example, the procedure assumes that the table name is unique and can be referenced by the procedure creator without specifying the table owner:

```
CREATE TEMPORARY PROCEDURE drop_table( IN @TableName char(128) )
BEGIN
    IF EXISTS  ( SELECT * FROM SYS.SYSTAB WHERE table_name = @TableName )
THEN
   EXECUTE IMMEDIATE 'DROP TABLE "' || @TableName || '"';
   MESSAGE 'Table "' || @TableName || '" dropped' to client;
    END IF;
END;
CALL drop_table( 'CustRank' );
```

- **RESULT clause** The RESULT clause declares the number and type of columns in the result set. The parenthesized list following the RESULT keyword defines the result column names and types. This information is returned by the embedded SQL DESCRIBE or by ODBC SQLDescribeCol when a CALL statement is being described. For a list of data types, see "SQL data types" on page 77.

  For more information about returning result sets from procedures, see "Returning results from procedures" [*SQL Anywhere Server - SQL Usage*].

  Some procedures can produce more than one result set, with different numbers of columns, depending on how they are executed. For example, the following procedure returns two columns under some circumstances, and one in others.

```
CREATE PROCEDURE names( IN formal char(1))
BEGIN
   IF formal = 'n' THEN
      SELECT GivenName
      FROM Employees
   ELSE
      SELECT Surname, GivenName
      FROM Employees
   END IF
END;
```

Procedures with variable result sets must be written without a RESULT clause, or in Transact-SQL. Their use is subject to the following limitations:

- **Embedded SQL** You must DESCRIBE the procedure call after the cursor for the result set is opened, but before any rows are returned, to get the proper shape of result set. The CURSOR *cursor-name* clause on the DESCRIBE statement is required.
- **ODBC, OLE DB, ADO.NET** Variable result-set procedures can be used by applications using these interfaces. The proper description of the result sets is carried out by the driver or provider.
- **Open Client applications** Variable result-set procedures can be used by Open Client applications.

If your procedure returns only one result set, you should use a RESULT clause. The presence of this clause prevents ODBC and Open Client applications from re-describing the result set after a cursor is open.

To handle multiple result sets, ODBC must describe the currently executing cursor, not the procedure's defined result set. Therefore, ODBC does not always describe column names as defined in the RESULT clause of the procedure definition. To avoid this problem, use column aliases in the SELECT statement that generates the result set.

- **NO RESULT SET clause** Declares that no result set is returned by this procedure. This is useful when an external environment needs to know that a procedure does not return a result set.
- **SQL SECURITY clause** The SQL SECURITY clause defines whether the procedure is executed as the INVOKER (the user who is calling the procedure), or as the DEFINER (the user who owns the procedure). The default is DEFINER.

  When SQL SECURITY INVOKER is specified, more memory is used because annotation must be done for each user that calls the procedure. Also, when SQL SECURITY INVOKER is specified, name resolution is done as the invoker as well. Therefore, care should be taken to qualify all object names (tables, procedures, and so on) with their appropriate owner. For example, suppose user1 creates the following procedure:

  ```
  CREATE PROCEDURE user1.myProcedure()
     RESULT( columnA INT )
     SQL SECURITY INVOKER
     BEGIN
       SELECT columnA FROM table1;
     END;
  ```

  If user2 attempts to run this procedure and a table user2.table1 *does not* exist, a table lookup error results. Additionally, if a user2.table1 *does* exist, that table is used instead of the intended user1.table1. To prevent this situation, qualify the table reference in the statement (user1.table1, instead of just table1).
- **ON EXCEPTION RESUME clause** This clause enables Transact-SQL-like error handling to be used within a Watcom-SQL syntax procedure.

  If you use ON EXCEPTION RESUME, the procedure takes an action that depends on the setting of the on_tsql_error option. If on_tsql_error is set to Conditional (the default) the execution continues if the next statement handles the error; otherwise, it exits.

Error-handling statements include the following:

- IF
- SELECT @variable =
- CASE
- LOOP
- LEAVE
- CONTINUE
- CALL
- EXECUTE
- SIGNAL
- RESIGNAL
- DECLARE
- SET VARIABLE

You should not use explicit error handling code with an ON EXCEPTION RESUME clause.

See "on_tsql_error option [compatibility]" [*SQL Anywhere Server - Database Administration*].

- **AT *location-string* clause** Create a proxy stored procedure on the current database for a remote procedure specified by *location-string*. The AT clause supports the semicolon (;) as a field delimiter in *location-string*. If no semicolon is present, a period is the field delimiter. This allows file names and extensions to be used in the database and owner fields.

  Remote procedures accept input parameters up to 254 bytes in length, and return up to 254 characters in output variables.

  If a remote procedure can return a result set, even if it does not return one in all cases, then the local procedure definition must contain a RESULT clause.

  For information about remote servers, see "CREATE SERVER statement" on page 506. For information about using remote procedures, see "Using remote procedure calls (RPCs)" [*SQL Anywhere Server - SQL Usage*].

Remarks

The CREATE PROCEDURE statement creates a procedure in the database. Users with DBA authority can create procedures for other users by specifying an owner. A procedure is invoked with a CALL statement.

If a stored procedure returns a result set, it cannot also set output parameters or return a return value.

When referencing a temporary table from multiple procedures, a potential issue can arise if the temporary table definitions are inconsistent and statements referencing the table are cached. See "Referencing temporary tables within procedures" [*SQL Anywhere Server - SQL Usage*].

Permissions

Must have RESOURCE authority, unless creating a temporary procedure.

Must have DBA authority for external procedures or to create a procedure for another user.

Side effects

Automatic commit.

See also

- "ALTER PROCEDURE statement" on page 376
- "BEGIN statement" on page 413
- "CALL statement" on page 419
- "CREATE FUNCTION statement" on page 458
- "CREATE PROCEDURE statement [T-SQL]" on page 486
- "DROP PROCEDURE statement" on page 579
- "EXECUTE IMMEDIATE statement [SP]" on page 600
- "GRANT statement" on page 627
- "Using procedures, triggers, and batches" [*SQL Anywhere Server - SQL Usage*]

Standards and compatibility

- **SQL/2003** Persistent Stored Module feature. The syntax extensions for Java result sets are as specified in the optional J621 feature.

Examples

The following procedure uses a case statement to classify the results of a query.

```
CREATE PROCEDURE ProductType (IN product_ID INT, OUT type CHAR(10))
BEGIN
   DECLARE prod_name CHAR(20);
   SELECT name INTO prod_name FROM Products
   WHERE ID = product_ID;
   CASE prod_name
   WHEN 'Tee Shirt' THEN
      SET type = 'Shirt'
   WHEN 'Sweatshirt' THEN
      SET type = 'Shirt'
   WHEN 'Baseball Cap' THEN
      SET type = 'Hat'
   WHEN 'Visor' THEN
      SET type = 'Hat'
   WHEN 'Shorts' THEN
      SET type = 'Shorts'
   ELSE
      SET type = 'UNKNOWN'
   END CASE;
END;
```

The following example replaces the ProductType procedure created in the previous example. After replacing the procedure, the parameters for Tee Shirt and Sweatshirt are updated:

```
CREATE OR REPLACE PROCEDURE ProductType (IN product_ID INT, OUT type
CHAR(10))
BEGIN
   DECLARE prod_name CHAR(20);
   SELECT name INTO prod_name FROM Products
   WHERE ID = product_ID;
   CASE prod_name
   WHEN 'Tee Shirt' THEN
      SET type = 'T Shirt'
   WHEN 'Sweatshirt' THEN
      SET type = 'Long Sleeve Shirt'
   WHEN 'Baseball Cap' THEN
      SET type = 'Hat'
   WHEN 'Visor' THEN
      SET type = 'Hat'
      WHEN 'Shorts' THEN
         SET type = 'Shorts'
      ELSE
         SET type = 'UNKNOWN'
      END CASE;
   END;
```

The following procedure uses a cursor and loops over the rows of the cursor to return a single value.

```
CREATE PROCEDURE TopCustomer (OUT TopCompany CHAR(35), OUT TopValue INT)
BEGIN
   DECLARE err_notfound EXCEPTION
   FOR SQLSTATE '02000';
   DECLARE curThisCust CURSOR FOR
      SELECT CompanyName,
          CAST(SUM(SalesOrderItems.Quantity *
          Products.UnitPrice) AS INTEGER) VALUE
      FROM Customers
      LEFT OUTER JOIN SalesOrders
      LEFT OUTER JOIN SalesOrderItems
      LEFT OUTER JOIN Products
      GROUP BY CompanyName;
   DECLARE ThisValue INT;
   DECLARE ThisCompany CHAR(35);
   SET TopValue = 0;
   OPEN curThisCust;
   CustomerLoop:
   LOOP
      FETCH NEXT curThisCust
      INTO ThisCompany, ThisValue;
      IF SQLSTATE = err_notfound THEN
         LEAVE CustomerLoop;
      END IF;
      IF ThisValue > TopValue THEN
         SET TopValue = ThisValue;
         SET TopCompany = ThisCompany;
         END IF;
   END LOOP CustomerLoop;
   CLOSE curThisCust;
END;
```

CREATE PROCEDURE statement [T-SQL]

Use this statement to create a new procedure in the database in a manner compatible with Adaptive Server Enterprise.

Syntax 1

The following subset of the Transact-SQL CREATE PROCEDURE statement is supported in SQL Anywhere.

CREATE PROCEDURE [*owner*.]*procedure_name*
[**NO RESULT SET**]
[[**(**] *@parameter_name data-type* [= *default*] [**OUTPUT**], ... [**)**]]
[**WITH RECOMPILE**] **AS** *statement-list*

Parameters

- **NO RESULT SET clause** Declares that no result set is returned by this procedure. This is useful when an external environment needs to know that a procedure does not return a result set.

Remarks

The following differences between Transact-SQL and SQL Anywhere statements (Watcom-SQL) are listed to help those writing in both dialects.

- **Variable names prefixed by @** The "@" sign denotes a Transact-SQL variable name, while Watcom-SQL variables can be any valid identifier, and the @ prefix is optional.
- **Input and output parameters** Watcom-SQL procedure parameters are INOUT by default or can specified as IN, OUT, or INOUT. Transact-SQL procedure parameters are INPUT parameters by default or can be specified as OUTPUT. Those parameters that would be declared as INOUT or as OUT in SQL Anywhere should be declared with OUTPUT in Transact-SQL.
- **Parameter default values** Watcom-SQL procedure parameters are given a default value using the keyword DEFAULT, while Transact-SQL uses an equality sign (=) to provide the default value.
- **Returning result sets** Watcom-SQL uses a RESULT clause to specify returned result sets. In Transact-SQL procedures, the column names or alias names of the first query are returned to the calling environment.

  The following Transact-SQL procedure illustrates how result sets are returned from Transact-SQL stored procedures:

  ```
  CREATE PROCEDURE showdept @deptname varchar(30)
  AS
     SELECT Employees.Surname, Employees.GivenName
     FROM Departments, Employees
     WHERE Departments.DepartmentName = @deptname
     AND Departments.DepartmentID = Employees.DepartmentID;
  ```

  The following is the corresponding Watcom-SQL procedure:

  ```
  CREATE PROCEDURE showdept(in deptname
           varchar(30) )
  RESULT ( lastname char(20), firstname char(20))
  ON EXCEPTION RESUME
  BEGIN
     SELECT Employees.Surname, Employees.GivenName
     FROM Departments, Employees
     WHERE Departments.DepartmentName = deptname
     AND Departments.DepartmentID = Employees.DepartmentID
  END;
  ```

- **Procedure body** The body of a Transact-SQL procedure is a list of Transact-SQL statements prefixed by the AS keyword. The body of a Watcom-SQL procedure is a compound statement, bracketed by BEGIN and END keywords.

Permissions

Must have RESOURCE authority.

Side effects

Automatic commit.

See also

- "CREATE FUNCTION statement" on page 458
- "CREATE PROCEDURE statement" on page 481

Standards and compatibility

- **SQL/2003** Transact-SQL extension.
- **Sybase** SQL Anywhere supports a subset of the Adaptive Server Enterprise CREATE PROCEDURE statement syntax.

  If the Transact-SQL WITH RECOMPILE optional clause is supplied, it is ignored. SQL Anywhere always recompiles procedures the first time they are executed after a database is started, and stores the compiled procedure until the database is stopped.

  Groups of procedures are not supported.

CREATE PROCEDURE statement (external procedures)

Use this statement to create an interface to a native or external procedure. To create a SQL procedure, see "CREATE PROCEDURE statement" on page 481.

Syntax

```
CREATE [ OR REPLACE ] PROCEDURE [ owner.]procedure-name
   ( [ parameter, ... ] )
[ RESULT ( result-column, ... ) | NO RESULT SET ]
[ DYNAMIC RESULT SETS integer-expression ]
[ SQL SECURITY { INVOKER | DEFINER } ]
EXTERNAL NAME 'external-call' [ LANGUAGE environment-name  ]

parameter :
  [ parameter-mode ] parameter-name data-type [ DEFAULT expression ]
| SQLCODE
| SQLSTATE

parameter-mode : IN
| OUT
| INOUT

result-column : column-name data-type

environment-name :
  C_ESQL32
| C_ESQL64
| C_ODBC32
| C_ODBC64
| CLR
| JAVA
```

| PERL
| PHP

Parameters

- **CREATE PROCEDURE** You can create permanent stored procedures that call external or native procedures written in a variety of programming languages. You can use PROC as a synonym for PROCEDURE.

  Parameter names must conform to the rules for other database identifiers such as column names. They must be a valid SQL data type. For a list of valid data types, see "SQL data types" on page 77.

  Parameters can be prefixed with one of the keywords IN, OUT, or INOUT. If you do not specify one of these values, parameters are INOUT by default. The keywords have the following meanings:

  - **IN** The parameter is an expression that provides a value to the procedure.
  - **OUT** The parameter is a variable that could be given a value by the procedure.
  - **INOUT** The parameter is a variable that provides a value to the procedure, and could be given a new value by the procedure.

  When procedures are executed using the CALL statement, not all parameters need to be specified. If a default value is provided in the CREATE PROCEDURE statement, missing parameters are assigned the default values. If an argument is not provided in the CALL statement, and no default is set, an error is given.

  SQLSTATE and SQLCODE are special OUT parameters that output the SQLSTATE or SQLCODE value when the procedure ends. The SQLSTATE and SQLCODE special values can be checked immediately after a procedure call to test the return status of the procedure.

  The SQLSTATE and SQLCODE special values are modified by the next SQL statement. Providing SQLSTATE or SQLCODE as procedure arguments allows the return code to be stored in a variable.

  Specifying OR REPLACE (CREATE OR REPLACE PROCEDURE) creates a new procedure, or replaces an existing procedure with the same name. This clause changes the definition of the procedure, but preserves existing permissions. An error is returned if you attempt to replace a procedure that is already in use.

  You cannot create TEMPORARY external call procedures.

- **RESULT clause** The RESULT clause declares the number and type of columns in the result set. The parenthesized list following the RESULT keyword defines the result column names and types. This information is returned by the embedded SQL DESCRIBE or by ODBC SQLDescribeCol when a CALL statement is being described.For a list of data types, see "SQL data types" on page 77.

  Procedures that call into Embedded SQL (LANGUAGE C_ESQL32, LANGUAGE C_ESQL64) or ODBC (LANGUAGE C_ODBC32, LANGUAGE C_ODBC64) external functions can return 0 or 1 result sets.

  Procedures that call into Perl or PHP (LANGUAGE PERL, LANGUAGE PHP) external functions cannot return result sets. Procedures that call native functions loaded by the database server cannot return result sets.

  Procedures that call into CLR or Java (LANGUAGE CLR, LANGUAGE JAVA) external functions can return 0, 1, or more result sets.

Some procedures can produce more than one result set, with different numbers of columns, depending on how they are executed. For example, the following procedure returns two columns under some circumstances, and one in others.

```
CREATE PROCEDURE names( IN formal char(1))
BEGIN
   IF formal = 'n' THEN
      SELECT GivenName
      FROM Employees
   ELSE
      SELECT Surname, GivenName
      FROM Employees
   END IF
END;
```

Procedures with variable result sets must be written without a RESULT clause, or in Transact-SQL. Their use is subject to the following limitations:

- **Embedded SQL** You must DESCRIBE the procedure call after the cursor for the result set is opened, but before any rows are returned, to get the proper shape of result set. The CURSOR *cursor-name* clause on the DESCRIBE statement is required.
- **ODBC, OLE DB, ADO.NET** Variable result-set procedures can be used by applications using these interfaces. The proper description of the result sets is carried out by the driver or provider.
- **Open Client applications** Variable result-set procedures can be used by Open Client applications.

If your procedure returns only one result set, you should use a RESULT clause. The presence of this clause prevents ODBC and Open Client applications from re-describing the result set after a cursor is open.

To handle multiple result sets, ODBC must describe the currently executing cursor, not the procedure's defined result set. Therefore, ODBC does not always describe column names as defined in the RESULT clause of the procedure definition. To avoid this problem, use column aliases in the SELECT statement that generates the result set.

For more information about returning result sets from procedures, see "Returning results from procedures" [*SQL Anywhere Server - SQL Usage*].

- **NO RESULT SET clause** Declares that no result set is returned by this procedure. This declaration can lead to a performance improvement.
- **DYNAMIC RESULT SETS clause** Use this clause with LANGUAGE CLR and LANGUAGE JAVA calls. If the DYNAMIC RESULT SETS clause is not provided, it is assumed that the method returns no result set.

  Note that procedures that call into Perl or PHP (LANGUAGE PERL, LANGUAGE PHP) external functions cannot return result sets. Procedures that call native functions loaded by the database server cannot return result sets.
- **SQL SECURITY clause** The SQL SECURITY clause defines whether the procedure is executed as the INVOKER (the user who is calling the procedure), or as the DEFINER (the user who owns the procedure). The default is DEFINER. For external calls, this clause establishes the ownership context for unqualified object references in the external environment.

When SQL SECURITY INVOKER is specified, more memory is used because annotation must be done for each user that calls the procedure. Also, when SQL SECURITY INVOKER is specified, name resolution is done as the invoker as well. Therefore, care should be taken to qualify all object names (tables, procedures, and so on) with their appropriate owner. For example, suppose user1 creates the following procedure:

```
CREATE PROCEDURE user1.myProcedure()
   RESULT( columnA INT )
   SQL SECURITY INVOKER
   BEGIN
     SELECT columnA FROM table1;
   END;
```

If user2 attempts to run this procedure and a table user2.table1 *does not* exist, a table lookup error results. Additionally, if a user2.table1 *does* exist, that table is used instead of the intended user1.table1. To prevent this situation, qualify the table reference in the statement (user1.table1, instead of just table1).

- **EXTERNAL NAME '*native-call*' clause**

  EXTERNAL NAME '*native-call*'

  native-call :
  [*operating-system*:]*function-name@library*; ...

  operating-system : **Unix**
  A procedure using the EXTERNAL NAME clause with no LANGUAGE attribute defines an interface to a native function written in a programming language such as C. The native function is loaded by the database server into its address space.

  The *library* name can include the file extension, which is typically *.dll* on Windows and *.so* on Unix. In the absence of the extension, the software appends the platform-specific default file extension for libraries. The following is a formal example.

  ```
  CREATE PROCEDURE mystring( IN instr LONG VARCHAR )
  EXTERNAL NAME 'mystring@mylib.dll;Unix:mystring@mylib.so';
  ```

  A simpler way to write the above EXTERNAL NAME clause, using platform-specific defaults, is as follows:

  ```
  CREATE PROCEDURE mystring( IN instr LONG VARCHAR )
  EXTERNAL NAME 'mystring@mylib';
  ```

  When called, the library containing the function is loaded into the address space of the database server. The native function will execute as part of the server. In this case, if the function causes a fault, then the database server will be terminated. Because of this, loading and executing functions in an external environment using the LANGUAGE attribute is recommended. If a function causes a fault in an external environment, the database server will continue to run.

  For information about native library calls, see "Calling external libraries from procedures" [*SQL Anywhere Server - Programming*].

- **EXTERNAL NAME '*c-call*' LANGUAGE {C_ESQL32 | C_ESQL64 | C_ODBC32 | C_ODBC64 } clause**

  EXTERNAL NAME '*c-call***' LANGUAGE C_ESQL32**

EXTERNAL NAME '*c-call*' LANGUAGE C_ESQL64

EXTERNAL NAME '*c-call*' LANGUAGE C_ODBC32

EXTERNAL NAME '*c-call*' LANGUAGE C_ODBC64

c-call :
[*operating-system*:]*function-name*@*library*; ...

operating-system : **Unix**
To call a compiled native C function in an external environment instead of within the database server, the stored procedure or function is defined with the EXTERNAL NAME clause followed by the LANGUAGE attribute specifying one of C_ESQL32, C_ESQL64, C_ODBC32, or C_ODBC64.

When the LANGUAGE attribute is specified, then the library containing the function is loaded by an external process and the external function will execute as part of that external process. In this case, if the function causes a fault, then the database server will continue to run.

The following is a sample procedure definition.

```
CREATE PROCEDURE ODBCinsert(
  IN ProductName CHAR(30),
  IN ProductDescription CHAR(50)
)
NO RESULT SET
EXTERNAL NAME 'ODBCexternalInsert@extodbc.dll'
LANGUAGE C_ODBC32;
```

For more information, see "The ESQL and ODBC external environments" [*SQL Anywhere Server - Programming*].

- **EXTERNAL NAME *clr-call* LANGUAGE CLR clause**

  EXTERNAL NAME '*clr-call*' LANGUAGE CLR

  clr-call :
  dll-name::*function-name*(param-type-1, ...)
  To call a .NET function in an external environment, the procedure interface is defined with an EXTERNAL NAME clause followed by the LANGUAGE CLR attribute.

  A CLR stored procedure or function behaves the same as a SQL stored procedure or function with the exception that the code for the procedure or function is written in a .NET language such as C# or Visual Basic, and the execution of the procedure or function takes place outside the database server (that is, within a separate .NET executable).

  The following is a sample procedure definition.

```
CREATE PROCEDURE clr_interface(
    IN p1 INT,
    IN p2 UNSIGNED SMALLINT,
    OUT p3 LONG VARCHAR)
NO RESULT SET
EXTERNAL NAME 'CLRlib.dll::CLRproc.Run( int, ushort, out string )'
LANGUAGE CLR;
```

  For more information, see "The CLR external environment" [*SQL Anywhere Server - Programming*].

- **EXTERNAL NAME *perl-call* LANGUAGE PERL clause**

  EXTERNAL NAME '*perl-call***' LANGUAGE PERL**

  perl-call :
  <file=*perl-file***> $sa_perl_return =** *perl-sub***($sa_perl_arg0**, ... **)**
  To call a Perl function in an external environment, the procedure interface is defined with an EXTERNAL NAME clause followed by the LANGUAGE PERL attribute.

  A Perl stored procedure or function behaves the same as a SQL stored procedure or function with the exception that the code for the procedure or function is written in Perl and the execution of the procedure or function takes place outside the database server (that is, within a Perl executable instance).

  The following is a sample procedure definition.

  ```
  CREATE PROCEDURE PerlWriteToConsole( IN str LONG VARCHAR)
  NO RESULT SET
  EXTERNAL NAME '<file=PerlConsoleExample>
      WriteToServerConsole( $sa_perl_arg0 )'
  LANGUAGE PERL;
  ```

  For more information, see "The PERL external environment" [*SQL Anywhere Server - Programming*].

- **EXTERNAL NAME *php-call* LANGUAGE PHP clause**

  EXTERNAL NAME '*php-call***' LANGUAGE PHP**

  php-call :
  <file=*php-file***> print** *php-func***($argv[1]**, ... **)**
  To call a PHP function in an external environment, the procedure interface is defined with an EXTERNAL NAME clause followed by the LANGUAGE PHP attribute.

  A PHP stored procedure or function behaves the same as a SQL stored procedure or function with the exception that the code for the procedure or function is written in PHP and the execution of the procedure or function takes place outside the database server (that is, within a PHP executable instance).

  The following is a sample procedure definition.

  ```
  CREATE PROCEDURE PHPPopulateTable()
  NO RESULT SET
  EXTERNAL NAME '<file=ServerSidePHPExample> ServerSidePHPSub()'
  LANGUAGE PHP;
  ```

  For more information, see "The PHP external environment" [*SQL Anywhere Server - Programming*].

- **EXTERNAL NAME *java-call* LANGUAGE JAVA clause**

  EXTERNAL NAME '*java-call***' LANGUAGE JAVA**

  java-call :
  [*package-name*.]*class-name*.*method-name method-signature*

  method-signature :
  ([*field-descriptor*, ...] **)** *return-descriptor*

  field-descriptor and *return-descriptor* :
  Z

| **B**
| **S**
| **I**
| **J**
| **F**
| **D**
| **C**
| **V**
| **[***descriptor*
| **L***class-name***;**

To call a Java method in an external environment, the procedure interface is defined with an EXTERNAL NAME clause followed by the LANGUAGE JAVA attribute.

A Java-interfacing stored procedure or function behaves the same as a SQL stored procedure or function with the exception that the code for the procedure or function is written in Java and the execution of the procedure or function takes place outside the database server (that is, within a Java Virtual Machine).

The following is a sample procedure definition.

```
CREATE PROCEDURE HelloDemo( IN name LONG VARCHAR )
NO RESULT SET
EXTERNAL NAME 'Hello.main([Ljava/lang/String;)V'
LANGUAGE JAVA;
```

For more information, see "The Java external environment" [*SQL Anywhere Server - Programming*].

Remarks

The CREATE PROCEDURE statement creates a procedure in the database. Users with DBA authority can create procedures for other users by specifying an owner. A procedure is invoked with a CALL statement.

If a stored procedure returns a result set, it cannot also set output parameters or return a return value.

When referencing a temporary table from multiple procedures, a potential issue can arise if the temporary table definitions are inconsistent and statements referencing the table are cached. See "Referencing temporary tables within procedures" [*SQL Anywhere Server - SQL Usage*].

Permissions

Must have RESOURCE authority, unless creating a temporary procedure.

Must have DBA authority for external procedures or to create a procedure for another user.

Side effects

Automatic commit.

See also

- "ALTER PROCEDURE statement" on page 376
- "CALL statement" on page 419
- "CREATE FUNCTION statement" on page 458
- "CREATE FUNCTION statement (external procedures)" on page 462
- "CREATE PROCEDURE statement" on page 481
- "DROP PROCEDURE statement" on page 579
- "GRANT statement" on page 627
- "Overview of external environments" [*SQL Anywhere Server - Programming*]

Standards and compatibility

- **SQL/2003** Persistent Stored Module feature. The syntax extensions for Java result sets are as specified in the optional J621 feature.

CREATE PROCEDURE statement (web services)

Use this statement to create a web services client procedure which makes an HTTP or SOAP request. To create a SQL procedure, see "CREATE PROCEDURE statement" on page 481.

```
CREATE [ OR REPLACE ] PROCEDURE [ owner.]procedure-name ( [ parameter, ... ] )
URL url-string
[ TYPE { http-type-spec-string | soap-type-spec-string } ]
[ HEADER header-string ]
[ CERTIFICATE certificate-string ]
[ CLIENTPORT clientport-string ]
[ PROXY proxy-string ]
[ SET protocol-option-string ]
[ SOAPHEADER soap-header-string ]
[ NAMESPACE namespace-string ]

http-type-spec-string :
HTTP[: { GET
  | POST[:MIME-type ]
  | PUT[:MIME-type ]
  | DELETE
  | HEAD } ]

soap-type-spec-string :
SOAP[:{ RPC | DOC }

parameter :
  parameter-mode parameter-name data-type [ DEFAULT expression ]

parameter-mode :
IN
| OUT
| INOUT

url-string :
{ HTTP | HTTPS | HTTPS_FIPS }://[user:password@]hostname[:port][/path]
```

protocol-option-string
[*http-option-list*]
[**,** *soap-option-list*]

http-option-list :
HTTP(
[**CH**[**UNK**]**={ ON** | **OFF** | **AUTO** }]
[**; VER**[**SION**]**={ 1.0** | **1.1** }]
)

soap-option-list:
SOAP(OP[**ERATION**]**=***soap-operation-name***)**

Parameters

- **CREATE PROCEDURE** You can create or replace a web services client procedure. You can use PROC as a synonym for PROCEDURE.

  For SOAP requests, the procedure name is used as the SOAP operation name by default. See the SET clause below for more information.

  Parameter names must conform to the rules for other database identifiers such as column names. They must be a valid SQL data type. For a list of valid data types, see "SQL data types" on page 77. Only SOAP requests support the transmission of typed data such as FLOAT, INT, and so on. HTTP requests support the transmission of strings only, so you are limited to CHAR types. For more information about supported SOAP types, see "Working with data types" [*SQL Anywhere Server - Programming*] and "Working with structured data types" [*SQL Anywhere Server - Programming*].

  Parameters can be prefixed with one of the keywords IN, OUT, or INOUT. If you do not specify one of these values, parameters are INOUT by default. The keywords have the following meanings:

  - **IN** The parameter is an expression that provides a value to the procedure.
  - **OUT** The parameter is a variable that could be given a value by the procedure.
  - **INOUT** The parameter is a variable that provides a value to the procedure, and could be given a new value by the procedure.

  When procedures are executed using the CALL statement, not all parameters need to be specified. If a default value is provided in the CREATE PROCEDURE statement, missing parameters are assigned the default values. If an argument is not provided in the CALL statement, and no default is set, an error is given.

  Specifying OR REPLACE (CREATE OR REPLACE PROCEDURE) creates a new procedure, or replaces an existing procedure with the same name. This clause changes the definition of the procedure, but preserves existing permissions. An error is returned if you attempt to replace a procedure that is already in use.

  You cannot create TEMPORARY web services procedures.

- **URL clause** Specifies the URI of the web service. The optional user name and password parameters provide a means of supplying the credentials needed for HTTP basic authentication. HTTP basic authentication base-64 encodes the user and password information and passes it in the Authentication header of the HTTP request. When specified in this way, the user name and password are passed unencrypted, as part of the URL.

Specifying HTTPS as the URI scheme configures the procedure for secure communication over Secure Socket Layer. Such a URI requires an appropriate CERTIFICATE clause to authenticate the server and establish a secure data channel.

Specifying HTTPS_FIPS forces the system to use the FIPS libraries. If HTTPS_FIPS is specified, but no FIPS libraries are present, non-FIPS libraries are used instead.

When specified in this way, the user name and password are passed unencrypted, as part of the URL.

- **TYPE clause** Used to specify the format used when making the web service request. If SOAP is specified or no type clause is included, the default type SOAP:RPC is used. HTTP implies HTTP:POST. Since SOAP requests are always sent as XML documents, HTTP:POST is always used to send SOAP requests. See "Creating web service client functions and procedures" [*SQL Anywhere Server - Programming*].
- **HEADER clause** When creating HTTP web service client procedures, use this clause to add, modify, or delete HTTP request header entries. The specification of headers closely resembles the format specified in RFC2616 Hypertext Transfer Protocol — HTTP/1.1, and RFC822 Standard for ARPA Internet Text Messages, including the fact that only printable ASCII characters can be specified for HTTP headers, and they are case-insensitive.

  For more information about using HTTP headers, see "Working with HTTP headers" [*SQL Anywhere Server - Programming*].
- **CERTIFICATE clause** To make a secure (HTTPS) request, a client must have access to the certificate used by the HTTPS server. The necessary information is specified in a string of semicolon-separated key/value pairs. The certificate can be placed in a file and the name of the file provided using the file key, or the whole certificate can be placed in a string, but not both. The following keys are available:

Key	Abbreviation	Description
file		The file name of the certificate.
certificate	cert	The certificate itself.
company	co	The company specified in the certificate.
unit		The company unit specified in the certificate.
name		The common name specified in the certificate.

  Certificates are required only for requests that are either directed to an HTTPS server, or can be redirected from a non-secure to a secure server.
- **CLIENTPORT clause** Identifies the port number on which the HTTP client procedure communicates using TCP/IP. It is provided for and recommended only for connections through firewalls that filter "outgoing" TCP/IP connections. You can specify a single port number, ranges of port numbers, or a combination of both; for example, CLIENTPORT '85,90-97'. See "ClientPort protocol option [CPORT]" [*SQL Anywhere Server - Database Administration*].

- **PROXY clause** Specifies the URI of a proxy server. For use when the client must access the network through a proxy. Indicates that the procedure is to connect to the proxy server and send the request to the web service through it.
- **SET clause** Specifies protocol-specific behavior options for HTTP and SOAP. The following list describes the supported SET options. CHUNK and VERSION apply to the HTTP protocol, and OPERATION applies to the SOAP protocol. Parameter substitution is supported for this clause.
  - **'HTTP(CH[UNK]=option)'** (HTTP or SOAP) This option allows you to specify whether to use chunking. Chunking allows HTTP messages to be broken up into several parts. Possible values are ON (always chunk), OFF (never chunk), and AUTO (chunk only if the contents, excluding auto-generated markup, exceeds 8196 bytes). For example, the following SET clause enables chunking:

    ```
    SET 'HTTP(CHUNK=ON)'
    ```

    If the CHUNK option is not specified, the default behavior is AUTO. If a chunked request fails in AUTO mode with a status of 505 **HTTP Version Not Supported**, or with 501 **Not Implemented**, or with 411 **Length Required**, the client retries the request without chunked transfer-coding.

    Set the CHUNK option to OFF (never chunk) if the HTTP server does not support chunked transfer-coded requests.

    Since CHUNK mode is a transfer encoding supported starting in HTTP version 1.1, setting CHUNK to ON requires that the version (VER) be set to 1.1, or not be set at all, in which case 1.1 is used as the default version.
  - **' HTTP(VER[SION]=ver)'** (HTTP or SOAP) This option allows you to specify the version of HTTP protocol that is used for the format of the HTTP message. For example, the following SET clause sets the HTTP version to 1.1:

    ```
    SET 'HTTP(VERSION=1.1)'
    ```

    Possible values are 1.0 and 1.1. If VERSION is not specified:

    - if CHUNK is set to ON, 1.1 is used as the HTTP version
    - if CHUNK is set to OFF, 1.0 is used as the HTTP version
    - if CHUNK is set to AUTO, either 1.0 or 1.1 is used, depending on whether the client is sending in CHUNK mode
  - **' SOAP(OP[ERATION]=soap-operation-name)** (SOAP only) This option allows you to specify the name of the SOAP operation, if it is different from the name of the procedure you are creating. The value of OPERATION is analogous to the name of a remote procedure call. For example, if you wanted to create a procedure called accounts_login that calls a SOAP operation called login, you would specify something like the following:

    ```
    CREATE PROCEDURE accounts_login(
          name LONG VARCHAR,
          pwd LONG VARCHAR )
       SET 'SOAP(OPERATION=login)'
    ```

    If the OPERATION option is not specified, the name of the SOAP operation must match the name of the procedure you are creating.

The following statement shows how several *protocol-option* settings are combined in the same SET clause:

```
CREATE PROCEDURE accounts_login(
      name LONG VARCHAR,
      pwd LONG VARCHAR )
   SET 'HTTP ( CHUNK=ON; VERSION=1.1 ), SOAP( OPERATION=login )'
   ...
```

- **SOAPHEADER clause** (SOAP format only) When declaring a SOAP web service as a procedure, use this clause to specify one or more SOAP request header entries. A SOAP header can be declared as a static constant, or can be dynamically set using the parameter substitution mechanism (declaring IN, OUT, or INOUT parameters for hd1, hd2, and so on). A web service procedure can define one or more IN mode substitution parameters, and a single INOUT or OUT substitution parameter.

  The following example illustrates how a client can specify the sending of several header entries with parameters and receiving the response SOAP header data:

  ```
  CREATE PROCEDURE soap_client
    (INOUT hd1 LONG VARCHAR, IN hd2 LONG VARCHAR, IN hd3 LONG VARCHAR)
    URL 'localhost/some_endpoint'
    SOAPHEADER '!hd1!hd2!hd3';
  ```

  For more information about using SOAP headers, see "Working with SOAP headers" [*SQL Anywhere Server - Programming*].

- **NAMESPACE clause** (SOAP format only) This clause identifies the method namespace usually required for both SOAP:RPC and SOAP:DOC requests. The SOAP server handling the request uses this namespace to interpret the names of the entities in the SOAP request message body. The namespace can be obtained from the WSDL (Web Services Description Language) of the SOAP service available from the web service server. The default value is the procedure's URL, up to but not including the optional path component. For more information about using SOAP namespaces, see "Working with structured data types" [*SQL Anywhere Server - Programming*].

For more information about creating web services, including examples, see "SQL Anywhere web services" [*SQL Anywhere Server - Programming*].

Remarks

Parameter values are passed as part of the request. The syntax used depends on the type of request. For HTTP:GET, the parameters are passed as part of the URL; for HTTP:POST requests, the values are placed in the body of the request. Parameters to SOAP requests are always bundled in the request body.

Permissions

Must have RESOURCE authority.

Must have DBA authority to create a procedure for another user.

Side effects

Automatic commit.

See also

- "ALTER PROCEDURE statement" on page 376
- "CALL statement" on page 419
- "CREATE FUNCTION statement" on page 458
- "CREATE FUNCTION statement (web services)" on page 467
- "CREATE PROCEDURE statement" on page 481
- "DROP PROCEDURE statement" on page 579
- "GRANT statement" on page 627
- "SQL Anywhere web services" [*SQL Anywhere Server - Programming*]
- "Creating web service client functions and procedures" [*SQL Anywhere Server - Programming*]

Standards and compatibility

- **SQL/2003** Persistent Stored Module feature. The syntax extensions for Java result sets are as specified in the optional J621 feature.

Example

The following example creates a web services client procedure named FtoC.

```
CREATE PROCEDURE FtoC( IN temperature FLOAT,
    INOUT inoutheader LONG VARCHAR,
    IN inheader LONG VARCHAR )
  URL 'http://localhost:8082/FtoCService'
  TYPE 'SOAP:DOC'
  SOAPHEADER '!inoutheader!inheader';
```

CREATE PUBLICATION statement [MobiLink] [SQL Remote]

Use this statement to create a publication. In MobiLink, a publication identifies synchronized data in a SQL Anywhere remote database. In SQL Remote, publications identify replicated data in both consolidated and remote databases.

Syntax 1 (MobiLink general use)

CREATE PUBLICATION [*owner*.]*publication-name*
(*article-definition*, ... **)**

article-definition :
 TABLE *table-name* [**(** *column-name*, ... **)**]
[**WHERE** *search-condition*]

Syntax 2 (MobiLink scripted upload)

CREATE PUBLICATION [*owner*.]*publication-name*
WITH SCRIPTED UPLOAD
(*article-definition*, ... **)**

article-definition :
 TABLE *table-name* [**(** *column-name*, ... **)**]
[**USING (** [**PROCEDURE**] [*owner***.**][*procedure-name*]
  FOR UPLOAD { **INSERT** | **DELETE** | **UPDATE** }, ... **)**]

Syntax 3 (MobiLink download-only publications)

CREATE PUBLICATION [*owner*.]*publication-name*
FOR DOWNLOAD ONLY
(*article-definition*, ... **)**

article-definition : **TABLE** *table-name* [**(** *column-name*, ... **)**]

Syntax 4 (SQL Remote)

CREATE PUBLICATION [*owner*.]*publication-name*
(*article-definition*, ... **)**

article-definition :
  TABLE *table-name* [**(** *column-name*, ... **)**]
[**WHERE** *search-condition*]
[**SUBSCRIBE BY** *expression*]

Parameters

- **article-definition** Publications are built from articles. To include more than one article, separate article definitions with commas. Each article is a table or part of a table. An article can be a vertical partition of a table (a subset of the table's columns), a horizontal partition (a subset of the table's rows based on a WHERE clause) or a vertical and horizontal partition.

  In Syntax 2, which is used for publications that perform scripted uploads, the article description also registers the scripts that are used to define the upload. See "Creating publications for scripted upload" [*MobiLink - Client Administration*].

  In Syntax 3, which is used for download-only publications, the article specifies only the tables and columns to be downloaded.

- **WHERE clause** The WHERE clause is a way of defining the subset of rows of a table to be included in an article.

  In MobiLink applications, the WHERE clause affects the rows included in the upload. (The download is defined by the download_cursor script.) In MobiLink SQL Anywhere remote databases, the WHERE clause can only refer to columns included in the article, and cannot contain subqueries, variables, or non-deterministic functions.

- **SUBSCRIBE BY clause** In SQL Remote, one way of defining a subset of rows of a table to be included in an article is to use a SUBSCRIBE BY clause. This clause allows many different subscribers to receive different rows from a table in a single publication definition.

Remarks

The CREATE PUBLICATION statement creates a publication in the database. A publication can be created for another user by specifying an owner name.

In MobiLink, publications are required in SQL Anywhere remote databases, and are optional in UltraLite databases. These publications and the subscriptions to them determine which data is uploaded to the MobiLink server.

You set options for a MobiLink publication with the ADD OPTION clause in the CREATE SYNCHRONIZATION SUBSCRIPTION statement or ALTER SYNCHRONIZATION SUBSCRIPTION statement.

Syntax 2 creates a publication for scripted uploads. Use the USING clause to register the stored procedures that you want to use to define the upload. For each table, you can use up to three stored procedures: one each for inserts, deletes, and updates.

Syntax 3 creates a download-only publication that can be synchronized with no log file. When download-only publications are synchronized, downloaded rows may overwrite changes that were made to those rows in the remote database.

In SQL Remote, publishing is a two-way operation, as data can be entered at both consolidated and remote databases. In a SQL Remote installation, any consolidated database and all remote databases must have the same publication defined. Running the SQL Remote extraction utility from a consolidated database automatically executes the correct CREATE PUBLICATION statement in the remote database.

Permissions

Must have DBA authority. Requires exclusive access to all tables referred to in the statement.

Side effects

Automatic commit.

See also

- "ALTER PUBLICATION statement [MobiLink] [SQL Remote]" on page 377
- "DROP PUBLICATION statement [MobiLink] [SQL Remote]" on page 580
- "CREATE SYNCHRONIZATION SUBSCRIPTION statement [MobiLink]" on page 519
- "ALTER SYNCHRONIZATION SUBSCRIPTION statement [MobiLink]" on page 386
- SQL Anywhere MobiLink clients: "Publishing data" [*MobiLink - Client Administration*]
- UltraLite MobiLink clients: "UltraLite CREATE PUBLICATION statement" [*UltraLite - Database Management and Reference*]
- SQL Remote: "Publications and articles" [*SQL Remote*]
- "Scripted upload" [*MobiLink - Client Administration*]
- "Download-only publications" [*MobiLink - Client Administration*]
- "ISYSSYNC system table" on page 807

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement publishes all columns and rows of two tables.

```
CREATE PUBLICATION pub_contact (
   TABLE Contacts,
   TABLE Company
);
```

The following statement publishes only some columns of one table.

```
CREATE PUBLICATION pub_customer (
   TABLE Customers ( ID, CompanyName, City )
);
```

The following statement publishes only the active customer rows by including a WHERE clause that tests the Status column of the Customers table.

```
CREATE PUBLICATION pub_customer (
   TABLE Customers ( ID, CompanyName, City, State, Status )
   WHERE Status = 'active'
);
```

The following statement publishes only some rows by providing a subscribe-by value. This method can be used only with SQL Remote.

```
CREATE PUBLICATION pub_customer (
   TABLE Customers ( ID, CompanyName, City, State )
   SUBSCRIBE BY State
);
```

The subscribe-by value is used as follows when you create a SQL Remote subscription.

```
CREATE SUBSCRIPTION TO pub_customer ( 'NY' )
   FOR jsmith;
```

The following example creates a MobiLink publication that uses scripted uploads:

```
CREATE PUBLICATION pub WITH SCRIPTED UPLOAD (
      TABLE t1 (a, b, c) USING (
         PROCEDURE my.t1_ui FOR UPLOAD INSERT,
         PROCEDURE my.t1_ud FOR UPLOAD DELETE,
         PROCEDURE my.t1_uu FOR UPLOAD UPDATE
      ),
      TABLE t2 AS my_t2 USING (
         PROCEDURE my.t2_ui FOR UPLOAD INSERT
      )
   );
```

The following example creates a download-only publication:

```
CREATE PUBLICATION p1 FOR DOWNLOAD ONLY (
   TABLE t1
);
```

CREATE REMOTE MESSAGE TYPE statement [SQL Remote]

Use this statement to identify a message-link and return address for outgoing messages from a database.

Syntax

CREATE REMOTE MESSAGE TYPE *message-system*
[**ADDRESS** *address-string*]

message-system:
FILE
| **FTP**
| **SMTP**

Parameters

- **message-system** One of the supported message systems.

- **address-string** The address for the specified message system.

Remarks

The Message Agent sends outgoing messages from a database using one of the supported message links. Return messages for users employing the specified link are sent to the specified address as long as the remote database is created by the extraction utility. The Message Agent starts links only if it has remote users for those links.

The address is the publisher's address under the specified message system. If it is an email system, the address string must be a valid email address. If it is a file-sharing system, the address string is a subdirectory of the directory set in the SQLREMOTE environment variable, or of the current directory if that is not set. You can override this setting on the GRANT CONSOLIDATE statement at the remote database.

To remove the address, execute a CREATE REMOTE MESSAGE TYPE statement without an ADDRESS clause.

The Initialization utility (dbinit) creates message types automatically, without an address. Unlike other CREATE statements, the CREATE REMOTE MESSAGE TYPE statement does not give an error if the type exists; instead it alters the type.

Permissions

Must have DBA authority.

Side effects

Automatic commit.

See also

- “GRANT PUBLISH statement [SQL Remote]” on page 633
- “GRANT REMOTE statement [SQL Remote]” on page 634
- “GRANT CONSOLIDATE statement [SQL Remote]” on page 632
- “DROP REMOTE MESSAGE TYPE statement [SQL Remote]” on page 581
- “SQL Remote message systems” [*SQL Remote*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

When remote databases are extracted using the extraction utility, the following statement sets all recipients of file message-system messages to send messages back to the *company* subdirectory.

The statement also instructs dbremote to look in the *company* subdirectory for incoming messages.

```
CREATE REMOTE MESSAGE TYPE file
ADDRESS 'company';
```

CREATE SCHEMA statement

Use this statement to create a collection of tables, views, and permissions for a database user.

Syntax

```
CREATE SCHEMA AUTHORIZATION userid
[
  create-table-statement
  | create-view-statement
  | grant-statement
 ] ... ;
```

Remarks

The CREATE SCHEMA statement creates a schema. A schema is a collection of tables, views, and their associated permissions.

The *userid* must be the user ID of the current connection. You cannot create a schema for another user.

If any statement contained in the CREATE SCHEMA statement fails, the entire CREATE SCHEMA statement is rolled back.

The CREATE SCHEMA statement is simply a way of collecting together individual CREATE and GRANT statements into one operation. There is no SCHEMA database object created in the database, and to drop the objects you must use individual DROP TABLE or DROP VIEW statements. To revoke permissions, you must use a REVOKE statement for each permission granted.

The individual CREATE or GRANT statements are not separated by statement delimiters. The statement delimiter marks the end of the CREATE SCHEMA statement itself.

The individual CREATE or GRANT statements must be ordered such that the objects are created before permissions are granted on them.

Although you can currently create more than one schema for a user, this is not recommended, and may not be supported in future releases.

Permissions

Must have RESOURCE authority.

Side effects

Automatic commit.

See also

- "CREATE TABLE statement" on page 523
- "CREATE VIEW statement" on page 547
- "GRANT statement" on page 627

Standards and compatibility

- **SQL/2003** Core feature.
- **Sybase** SQL Anywhere does not support the use of REVOKE statements within the CREATE SCHEMA statement, and does not allow its use within Transact-SQL batches or procedures.

Example

The following CREATE SCHEMA statement creates a schema consisting of two tables. The statement must be executed by the user ID sample_user, who must have RESOURCE authority. If the statement creating table t2 fails, neither table is created.

```
CREATE SCHEMA AUTHORIZATION sample_user
CREATE TABLE t1 ( id1 INT PRIMARY KEY )
CREATE TABLE t2 ( id2 INT PRIMARY KEY );
```

The statement delimiter in the following CREATE SCHEMA statement is placed after the first CREATE TABLE statement. As the statement delimiter marks the end of the CREATE SCHEMA statement, the example is interpreted as a two statement batch by the database server. If the statement creating table t2 fails, the table t1 is still created.

```
CREATE SCHEMA AUTHORIZATION sample_user
CREATE TABLE t1 ( id1 INT PRIMARY KEY );
CREATE TABLE t2 ( id2 INT PRIMARY KEY );
```

CREATE SERVER statement

Use this statement to create a remote server.

Syntax 1

CREATE SERVER *server-name*
CLASS *server-class-string*
USING *connection-info-string*
[**READ ONLY**]

server-class-string :
'SAODBC'
| **'ASEODBC'**
| **'DB2ODBC'**
| **'MSSODBC'**
| **'ORAODBC'**
| **'MSACCESSODBC'**
| **'MYSQLODBC'**
| **'ULODBC'**
| **'ADSODBC'**
| **'ODBC'**
| **'SAJDBC'**
| **'ASEJDBC'**

connection-info-string :
{ *host-name*:*port-number* [/*dbname*] | *data-source-name* | *sqlanywhere-connection-string* }

Syntax 2

CREATE SERVER *server-name*
CLASS 'DIRECTORY'
USING *using-string*

using-string :
'ROOT = *path*

[;SUBDIRS = *n* **]**
[;READONLY = { YES | NO }]'

Parameters

- **CLASS clause** Specifies the server class you want to use for a remote connection. Server classes contain detailed server capability information. The DIRECTORY class is used in Syntax 2 to access a directory on the local computer.

- **USING clause** In Syntax 1, the USING clause supplies a connection string for the database server. The appropriate connection string depends on the driver being used, which in turn depends on the *server-class-string*.

  If an ODBC-based server class is used, the USING clause is the *data-source-name*. The *data-source-name* is the ODBC Data Source Name.

  For SQL Anywhere remote servers (SAODBC server classes), the *connection-info-string* parameter can be any valid SQL Anywhere connection string. You can use any SQL Anywhere connection parameters. For example, if you have connection problems, you can include a LOG connection parameter to troubleshoot the connection attempt.

  For more information about SQL Anywhere connection strings, see "Connection parameters" [*SQL Anywhere Server - Database Administration*].

  On Unix platforms, you need to reference the ODBC driver manager as well. For example, using the supplied iAnywhere Solutions ODBC drivers, the syntax is as follows:

  ```
  USING 'driver=SQL Anywhere 11;dsn=my_dsn'
  ```

  If a JDBC-based server class is used, the USING clause is of the form *host-name*:*port-number* [/*dbname*], where:

  - **host-name** The computer the remote server runs on.

  - **port-number** The TCP/IP port number the remote server listens on. The default port number for SQL Anywhere is 2638.

  - **dbname** For SQL Anywhere remote servers, if you do not specify a *dbname*, then the default database is used. For Adaptive Server Enterprise, the default is the master database, and an alternative to using *dbname* is to another database by some other means (for example, in the FORWARD TO statement).

    In Syntax 2, the USING clause specifies the following values for the local directory:

- **ROOT clause** The path, relative to the database server, that is the root of the directory access class. When you create a proxy table using the directory access server name, the proxy table is relative to this root path.

- **SUBDIRS clause** A number between 0 and 10 that represents the number of levels of directories within the root that the database server can access. If SUBDIRS is omitted or set to 0, then only the files in the root directory are accessible via the directory access server. You can create proxy tables to any of the directories or subdirectories available via the directory access server.

- **READONLY clause** Specifies whether the files accessed by the directory access server can be modified. By default, this is set to NO.

- **CREATEDIRS clause** Specifies whether directories can be created using the directory access server. By default, this is set to NO.

Remarks

When you create a remote server, it is added to the ISYSSERVER system table.

Syntax 1 The CREATE SERVER statement defines a remote server.

For more information about server classes and how to configure a server, see "Server classes for remote data access" [*SQL Anywhere Server - SQL Usage*].

Syntax 2 The CREATE SERVER statement lets you create a directory access server that accesses the local directory structure on the computer where the database server is running. You must create an external login for each database user that needs to use the directory access server. On Unix, the database server runs as a specific user, so file permissions are based on the permissions granted to the database server user.

For more information about directory access servers, see "Using directory access servers" [*SQL Anywhere Server - SQL Usage*].

Permissions

Must have DBA authority to execute this command.

Not supported on Windows Mobile.

Side effects

Automatic commit.

See also

- "ALTER SERVER statement" on page 380
- "DROP SERVER statement" on page 581
- "Server classes for remote data access" [*SQL Anywhere Server - SQL Usage*]
- "ISYSSERVER system table" on page 807

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following example creates a SQL Anywhere remote server named testsa, located on the computer named apple and listening on port number 2638, use:

```
CREATE SERVER testsa
CLASS 'SAJDBC'
USING 'apple:2638';
```

The following example creates a remote server for the JDBC-based Adaptive Server named ase_prod. Its computer name is banana and its port number is 3025.

```
CREATE SERVER ase_prod
CLASS 'asejdbc'
USING 'banana:3025';
```

The following example creates a remote server for the Oracle server named oracle723. Its ODBC Data Source Name is oracle723.

```
CREATE SERVER oracle723
CLASS 'oraodbc'
USING 'oracle723';
```

The following example creates a directory access server that only sees files within the directory *c:\temp*:

```
CREATE SERVER diskserver0
CLASS 'directory'
USING 'root=c:\temp';
CREATE EXTERNLOGIN DBA TO diskserver0;
CREATE EXISTING TABLE diskdir0 AT 'diskserver0;;;.';

-- Get a list of those files.
SELECT permissions, file_name, size FROM diskdir0;
```

The following example creates a directory access server that sees nine levels of directories:

```
-- Create a directory server that sees 9 levels of directories.
CREATE SERVER diskserver9
CLASS 'directory'
USING 'ROOT=c:\temp;SUBDIRS=9';
CREATE EXTERNLOGIN DBA TO diskserver9;
CREATE EXISTING TABLE diskdir9 AT 'diskserver9;;;.';
```

CREATE SERVICE statement

Use this statement to permit a database server to act as a web server.

Syntax 1 - DISH service

CREATE SERVICE *service-name*
TYPE 'DISH'
[**GROUP** { *group-name* | **NULL** }]
[**FORMAT** { **'DNET'** | **'CONCRETE'** [**EXPLICIT** { **ON** | **OFF** }] | **'XML'** | **NULL** }]
[*common-attributes*]

Syntax 2 - SOAP service

CREATE SERVICE *service-name*
TYPE 'SOAP'
[**DATATYPE** { **ON** | **OFF** | **IN** | **OUT** }]
[**FORMAT** { **'DNET'** | **'CONCRETE'** [**EXPLICIT** { **ON** | **OFF** }] | **'XML'** | **NULL** }]
[*common-attributes*]
AS *statement*

Syntax 3 - Miscellaneous services

CREATE SERVICE *service-name*
TYPE { **'RAW'** | **'HTML'** | **'JSON'** | **'XML'** }
[**URL** [**PATH**] { **ON** | **OFF** | **ELEMENTS** }]
[*common-attributes*]
[**AS** { *statement* | **NULL** }]

common-attributes:
[**AUTHORIZATION** { **ON** | **OFF** }]

[**ENABLE** | **DISABLE**]
[**METHODS '***method,...***'**]
[**SECURE** { **ON** | **OFF** }]
[**USER** { *user-name* | **NULL** }]

method:
DEFAULT
| **POST**
| **GET**
| **HEAD**
| **PUT**
| **DELETE**
| **NONE**
| *****

Parameters

- **service-name** Web service names can be any sequence of alphanumeric characters or /, -, _, ., !, ~, *, ', (, or), except that the first character must not begin with a slash (/) and the name must not contain two or more consecutive slash characters.

  Unlike other services, you cannot specify a forward slash (/) in a DISH service name.

- **TYPE clause** Identifies the type of the service by the result set that is returned. The type must be one of the listed service types. There is no default value.

  - **'SOAP'** The result set is returned as a SOAP response. The format of the data is determined by the FORMAT clause. A request to a SOAP service must be a valid SOAP request, not just a simple HTTP request. For more information about the SOAP standards, see http://www.w3.org/TR/2000/NOTE-SOAP-20000508/.

  - **'DISH'** A DISH service (Determine SOAP Handler) acts as a proxy for those SOAP services identified by the GROUP clause, and generates the WSDL (Web Services Description Language) for each of these SOAP services upon request.

  - **'RAW'** The result set is sent to the client without any further formatting. You can produce formatted documents by generating the required tags explicitly within your procedure. If you create a service with TYPE 'RAW', you should set the HTTP header Content-Type using sa_set_http_header. Otherwise, some browsers display the content as plain text. See "Using procedures that provide HTML documents" [*SQL Anywhere Server - Programming*], and "sa_set_http_header system procedure" on page 944.

  - **'HTML'** The result set of a statement or procedure is automatically formatted into an HTML document that contains a table.

  - **'JSON'** The result set is returned in JavaScript Object Notation (JSON). JSON web services are similar to XML web services in that both accept "generic" HTTP requests (that is, the requests do not have to contain a specially-formatted body like SOAP web services). For more information about JSON, visit http://www.json.org.

  - **'XML'** The result set is returned as XML. If the result set is already XML, no additional formatting is applied. If it is not already XML, it is automatically formatted as XML. The effect is similar to that of using the FOR XML RAW clause in a SELECT statement.

- **GROUP clause** Applies to DISH services only. Specifies a common prefix that controls which SOAP services the DISH service exposes. For example, specifying GROUP *xyz* exposes only SOAP services *xyz/aaaa*, *xyz/bbbb*, or *xyz/cccc*, but does not expose *abc/aaaa* or *xyzaaaa*. If no group name is specified, the DISH service exposes all the SOAP services in the database. SOAP services can be exposed by more than one DISH service. The same characters are permitted in group names as in service names.

  When creating a service, GROUP NULL is identical to not specifying a GROUP clause. However, if GROUP NULL is used in an ALTER SERVICE statement, GROUP NULL removes any existing grouping whereas the absence of a GROUP clause does not.

- **DATATYPE clause** Applies to SOAP services only. Controls whether data typing is supported for parameter inputs and/or result set outputs (responses) for all SOAP service formats. When supported, data typing allows a SOAP toolkit to parse and cast the data to the appropriate type. Parameter data types are exposed in the schema section of the WSDL generated by the DISH service. Output data types are represented as XML schema type attributes for each column of data.

  The following values are permitted for the DATATYPE clause:

  - **ON** Generate data typing for input parameters and result set responses.
  - **OFF** Do not generate data typing of input parameters and result set responses (the default).
  - **IN** Generate data typing of input parameters only.
  - **OUT** Generate data typing of result set responses only.

  For more information about SOAP services, see "Using SOAP services" [*SQL Anywhere Server - Programming*].

  For more information about mapping XMLSchema types to SQL data types, see "Working with data types" [*SQL Anywhere Server - Programming*].

- **URL clause** Determines whether URL paths are accepted and, if so, how they are processed.

  - **OFF** OFF indicates that nothing can follow the service name in a URL request. Specify OFF if the remainder of the URL path is not permitted, or if the service name ends with a forward slash (/). For example, if OFF is selected and you have the URL path *http://<host-name>/<service-name>/aaa/bbb/ccc*, only *http://<host-name>/<service-name>* is permitted. The remainder of the URL path, */<aaa/bbb/ccc* is not permitted.

  - **ON** ON indicates that the remainder of the URL is interpreted as the value of a variable named URL. Specify ON if the remainder of the URI path is permitted and is set as a single parameter. For example, in the URL path *http://<host-name>/<service-name>/aaa/bbb/ccc*, the parameter value is *aaa/bbb/ccc*.

  - **ELEMENTS** ELEMENTS indicates that the remainder of the URL path is to be split at the slash characters into a list of up to 10 elements set as multiple parameters. For example, in the URL path *http://<host-name>/<service-name>/aaa/bbb/ccc*, each element of the path is treated as a separate parameter. For example, url1=aaa, url2=bbb, url3=ccc, and so on. The values are assigned to variables named url plus a numeric suffix of between 1 and 10. If fewer than 10 values are supplied, the remaining variables are set to NULL.

  If the service name ends with the forward slash character /, then url must be set to OFF. The default value is OFF.

For more information about URLs, see "Understanding how URLs are interpreted" [*SQL Anywhere Server - Programming*], and "Working with variables" [*SQL Anywhere Server - Programming*].

- **FORMAT clause** Applies to DISH and SOAP services only. Generates output formats compatible with various types of SOAP clients, such as .NET or Java JAX-RPC. If the format of a SOAP service is not specified, the format is inherited from the service's DISH service declaration. If the DISH service also does not declare a format, it defaults to DNET, which is compatible with .NET clients. A SOAP service that does not declare a format can be used with different types of SOAP clients by defining multiple DISH services, each having a different FORMAT type.

  The following formats are supported:

  - **'DNET'** The Microsoft DataSet format for use with .NET SOAP clients. DNET is the default FORMAT value and was the only format available before SQL Anywhere 9.0.2.
  - **'CONCRETE'** A platform-neutral DataSet format for use with clients such as JAX-WS, or with clients that automatically generate interfaces based on the format of the returned data structure. Specifying this format type exposes either an explicit dataset element or a SimpleDataset element within the WSDL.

    When **EXPLICIT ON** is specified (the default), the WSDL describes an explicit DataSet element when the following criteria are met:

    - the CREATE SERVICE statement calls a stored procedure
    - a RESULT clause is specified in the stored procedure

    When the service statement or the stored procedure is not defined correctly, the result set mapping generated by the DISH services reverts to the SimpleDataset element.

    When **EXPLICIT OFF** is specified, the WSDL describes the SimpleDataset element. The SimpleDataset element describes the result set as a containment hierarchy of a rowset composed of an array of rows, each of which contains an array of column elements. The explicit dataset element extends this by including the actual names and types of each of the columns.
  - **'XML'** A simple XML string format. The DataSet is returned as a string that can be passed to an XML parser. This format is the most portable between SOAP clients.
  - **NULL** For SOAP services, the format inherits from the DISH service format (if it has one) or default to DNET. For DISH services, the format defaults to DNET. When creating a new service, the effect of specifying FORMAT NULL is identical to not specifying any format. However, if FORMAT NULL is used in an ALTER SERVICE statement, FORMAT NULL overrides any previous format whereas the absence of a FORMAT clause does not override the previous format.

- ***statement*** If the statement is NULL, the URL must specify the statement to be executed. Otherwise, the specified SQL statement is the only one that can be executed through the service. SOAP services must have statements; DISH services cannot have statements. The default value is NULL.

  It is strongly recommended that all services used in production systems define a statement. The statement can be NULL only if authorization is enabled.

- **AUTHORIZATION clause** Determines whether users must specify a user name and password through basic HTTP authorization when connecting to the service. The default value is ON. If authorization is OFF, the AS clause is required and a single user must be identified by the USER clause. All requests are run using that user's account and permissions. If authorization is ON, all users must provide a user name and password. Optionally, you can limit the users that are permitted to use the service by providing a user or group name using the USER clause. If the user name is NULL, all known users can access the service. It is recommended that production systems be run with authorization turned on and that you grant permission to use the service by adding users to a group.

When the authorization value is ON, an HTTP client connecting to a web service uses basic authentication (RFC 2617) which obfuscates the user and password information using base-64 encoding. It is recommended that you use the HTTPS protocol for increased security.

- **ENABLE and DISABLE clauses** Determines whether the service is available for use. By default, when a service is created, it is enabled. When creating or altering a service, you may include an ENABLE or DISABLE clause. Disabling a service effectively takes the service off line. Later, it can be enabled using ALTER SERVICE with the ENABLE clause.
- **METHODS clause** Indicates what type of requests are supported by the service. Valid request types are DEFAULT, POST, GET, HEAD, PUT, DELETE, and NONE. An asterisk (*****) may be used as a short form to represent the POST, GET, and HEAD attributes which are default request types for the RAW, HTML and XML service types. The default request types for SOAP services are POST and HEAD. The default request types for DISH services are GET and HEAD. Not all request types are valid for all the service types. The following table summarizes the valid request types for each service type.

Request type	Applies to service	Description
DEFAULT	all	Use DEFAULT to reset the set of request types to the default set for the given service type. It cannot be included in a list with other request types.
POST	SOAP, DISH, RAW, HTML, XML	Enabled by default for SOAP, RAW, HTML and XML.
GET	DISH, RAW, HTML, XML	Enabled by default for DISH, RAW, HTML and XML.
HEAD	SOAP, DISH, RAW, HTML, XML	Enabled by default for SOAP, DISH, RAW, HTML and XML.
PUT	RAW, HTML, XML	Not enabled by default.

Request type	**Applies to service**	**Description**
DELETE	RAW, HTML, XML	Not enabled by default.
NONE	all	Use NONE to disable access to a service. When applied to a SOAP service, the service cannot be directly accessed by a SOAP request. Always use a DISH service as the SOAP endpoint for one or more SOAP services. To enforce exclusive access to a SOAP operation through a DISH service endpoint, specify **NONE** for each SOAP service.
*	RAW, HTML, XML	Same as specifying **'POST,GET,HEAD'**.

For example, to specify the full list of request types for the RAW service type, you can use either of the following clauses:

```
METHODS '*,PUT,DELETE'
METHODS 'POST,GET,HEAD,PUT,DELETE'
```

To reset the list of request types for any service type to its default, you can use the following clause:

```
METHODS 'DEFAULT'
```

- **SECURE clause** Indicates whether unsecured connections are accepted. ON indicates that only HTTPS connections are to be accepted. Service requests received on the HTTP port are automatically redirected to the HTTPS port. If set to OFF, both HTTP and HTTPS connections are accepted. The default value is OFF.
- **USER clause** If authorization is disabled, this parameter becomes mandatory and specifies the user ID used to execute all service requests. If authorization is enabled (the default), this optional clause identifies the user or group permitted to access the service. The default value is NULL, which grants access to all users.

Remarks

The CREATE SERVICE statement causes the database server to act as a web server. A new entry is created in the ISYSWEBSERVICE system table.

Permissions

Must have DBA authority.

Side effects

None.

See also

- “ALTER SERVICE statement” on page 382
- “DROP SERVICE statement” on page 582
- “ISYSWEBSERVICE system table” on page 809
- “SQL Anywhere web services” [*SQL Anywhere Server - Programming*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Examples

To set up a web server quickly, start a database server with the -xs option (for example, -xs http), then execute the following statement:

```
CREATE SERVICE tables TYPE 'HTML'
   AUTHORIZATION OFF
   USER DBA
   AS SELECT *
      FROM SYS.SYSTAB;
```

After executing this statement, use any web browser to open the URL http://localhost/tables.

The following example demonstrates how to write a Hello World program.

```
CREATE PROCEDURE hello_world_proc( )
RESULT (html_doc long varchar)
BEGIN
   CALL dbo.sa_set_http_header( 'Content-Type', 'text/html' );
   SELECT '<html>\n'
       || '<head><title>Hello World</title></head>\n'
       || '<body>\n'
       || '<h1>Hello World!</h1>\n'
       || '</body>\n'
       || '</html>\n';
END;

CREATE SERVICE hello_world TYPE 'RAW'
AUTHORIZATION OFF
USER DBA
AS CALL hello_world_proc;
```

After executing this statement, use any web browser to open the URL *http://localhost/hello_world*.

The following example demonstrates how to create a JSON service.

```
CREATE PROCEDURE ListEmployees()
RESULT (
 EmployeeID            integer,
 Surname               person_name_t,
 GivenName             person_name_t,
 StartDate             date,
 TerminationDate       date )
BEGIN
  SELECT EmployeeID, Surname, GivenName,
         StartDate, TerminationDate
  FROM Employees
END;

CREATE SERVICE "JSON/EmployeeList"
TYPE 'JSON'
```

```
AUTHORIZATION OFF
SECURE OFF
USER DBA
AS CALL ListEmployees();
```

After executing this statement, use any web browser to open the URL *http://localhost/JSON/EmployeeList* and save the JSON response to a file so that it can be browsed using a text editor.

CREATE STATISTICS statement

Recreates the column statistics used by the optimizer, and stores them in the ISYSCOLSTAT system table.

Syntax

CREATE STATISTICS *object-name* [**(** *column-list* **)**]

object-name :
table-name | *materialized-view-name* | *temp-table-name*

Remarks

The CREATE STATISTICS statement recreates the column statistics that SQL Anywhere uses to optimize database queries, and can be performed on base tables, materialized views, local temporary tables, and global temporary tables. You cannot create statistics on proxy tables. Column statistics include histograms, which reflect the distribution of data in the database for the specified columns. By default, column statistics are automatically created for tables with five or more rows.

In rare circumstances, when your database queries are very variable, and when data distribution is not uniform or the data is changing frequently, you can improve performance by running the CREATE STATISTICS statement against a table or column.

When executing, the CREATE STATISTICS statement updates existing column statistics regardless of the size of the table, unless the table is empty, in which case nothing is done. If column statistics exist for an empty table, they remain unchanged by the CREATE STATISTICS statement. To remove column statistics for an empty table, execute the DROP STATISTICS statement.

The process of running CREATE STATISTICS performs a complete scan of the table. For this reason, careful consideration should be made before issuing a CREATE STATISTICS statement.

If you drop statistics, it is recommended that you recreate them using the CREATE STATISTICS statement. Without statistics, the optimizer can generate inefficient data access plans, causing poor database performance.

Permissions

Must have DBA authority.

Side effects

Execution plans may change.

See also

- "Optimizer estimates and column statistics" [*SQL Anywhere Server - SQL Usage*]
- "DROP STATISTICS statement" on page 583
- "LOAD TABLE statement" on page 660
- "ISYSCOLSTAT system table" on page 801
- "Histogram utility (dbhist)" [*SQL Anywhere Server - Database Administration*]
- "sa_get_histogram system procedure" on page 876

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement updates the column statistics for the ProductID column of the SalesOrderItems table:

```
CREATE STATISTICS SalesOrderItems ( ProductID );
```

CREATE SUBSCRIPTION statement [SQL Remote]

Use this statement to create a subscription for a user to a publication.

Syntax

CREATE SUBSCRIPTION
TO *publication-name* [**(** *subscription-value* **)**]
FOR *subscriber-id*

publication-name: *identifier*

subscription-value : *string*

subscriber-id: *string*

Parameters

- **publication-name** The name of the publication to which the user is being subscribed. This can include the owner of the publication.

- **subscription-value** A string that is compared to the subscription expression of the publication. The subscriber receives all rows for which the subscription expression matches the subscription value.

- **subscriber-id** The user ID of the subscriber to the publication. This user must have been granted REMOTE permissions.

Remarks

In a SQL Remote installation, data is organized into publications for replication. To receive SQL Remote messages, a subscription must be created for a user ID with REMOTE permissions.

If a string is supplied in the subscription, it is matched against each SUBSCRIBE BY expression in the publication. The subscriber receives all rows for which the value of the expression is equal to the supplied string.

In SQL Remote, publications and subscriptions are two-way relationships. If you create a subscription for a remote user to a publication on a consolidated database, you should also create a subscription for the consolidated database on the remote database. The extraction utility carries this out automatically.

Permissions

Must have DBA authority.

Side effects

Automatic commit.

See also

- "DROP SUBSCRIPTION statement [SQL Remote]" on page 584
- "GRANT REMOTE statement [SQL Remote]" on page 634
- "SYNCHRONIZE SUBSCRIPTION statement [SQL Remote]" on page 765
- "START SUBSCRIPTION statement [SQL Remote]" on page 756
- "ISYSSUBSCRIPTION system table" on page 807

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement creates a subscription for the user p_chin to the publication pub_sales. The subscriber receives all rows for which the subscription expression has a value of Eastern.

```
CREATE SUBSCRIPTION
TO pub_sales ( 'Eastern' )
FOR p_chin;
```

CREATE SYNCHRONIZATION PROFILE statement [MobiLink]

Use this statement to create a SQL Anywhere synchronization profile. Synchronization profiles define how a SQL Anywhere database synchronizes with the MobiLink server.

Syntax

CREATE SYNCHRONIZATION PROFILE *name string*

Parameters

- **name** Specifies the name of the synchronization profile to create. Each profile must have a unique name.

- **string** Specify a valid option string as described below. Option strings are specified as semicolon delimited lists of elements of the form <option name>=<option value>. For example *publication=p1;verbosity=high*.

Remarks

For a listing of the synchronization profile options supported by dbmlsync, see "MobiLink synchronization profiles" [*MobiLink - Client Administration*].

For options that take a Boolean value, setting the value to TRUE is equivalent to specifying the corresponding option on the command line.

The following values can be used to specify TRUE: TRUE, ON, 1, YES.

The following values can be used to specify FALSE: FALSE, OFF, 0, NO.

Permissions

Must have DBA authority.

Side effects

Automatic commit.

See also

- "ALTER SYNCHRONIZATION PROFILE statement [MobiLink]" on page 384
- "DROP SYNCHRONIZATION PROFILE statement [MobiLink]" on page 586

CREATE SYNCHRONIZATION SUBSCRIPTION statement [MobiLink]

Use this statement in a SQL Anywhere remote database to create a subscription between a MobiLink user and a publication.

Syntax

CREATE SYNCHRONIZATION SUBSCRIPTION
TO *publication-name*
[**FOR** *ml_username*, ...]
[**TYPE** *network-protocol*]
[**ADDRESS** *protocol-options*]
[**OPTION** *option*=*value*, ...]

ml_username: *identifier*

network-protocol: **http** | **https** | **tls** | **tcpip**

protocol-options: *string*

value: *string* | *integer*

Parameters

- **TO clause** Specify the name of a publication.
- **FOR clause** Specify one or more MobiLink user names. If you specify more than one user name, a separate subscription is created for each user.

  ml_username is a user who is authorized to synchronize with the MobiLink server.

  For more information about synchronization user names, see "Introduction to MobiLink users" [*MobiLink - Client Administration*].

  Omit the FOR clause to set the protocol type, protocol options, and extended options for a publication.

  For information about how dbmlsync processes options that are specified in different locations, see "Priority order" [*MobiLink - Client Administration*].
- **TYPE clause** This clause specifies the network protocol to use for synchronization. The default protocol is tcpip.

  For more information about network protocols, see "CommunicationType (ctp) extended option" [*MobiLink - Client Administration*].
- **ADDRESS clause** This clause specifies network protocol options such as the location of the MobiLink server. Multiple options must be separated with semi-colons.

  For a complete list of protocol options, see "MobiLink client network protocol option summary" [*MobiLink - Client Administration*].
- **OPTION clause** This clause allows you to set extended options for the subscription. If no FOR clause is provided, the extended options act as default settings for the publication.

  For information about how dbmlsync processes options that are specified in different locations, see "Priority order" [*MobiLink - Client Administration*].

  For a complete list of options, see "MobiLink SQL Anywhere client extended options" [*MobiLink - Client Administration*].

Remarks

The *network-protocol*, *protocol-options*, and *options* can be set in several places.

For information about how dbmlsync processes options that are specified in different locations, see "Priority order" [*MobiLink - Client Administration*].

This statement causes options and other information to be stored in the SQL Anywhere ISYSSYNC system table. Anyone with DBA authority for the database can view the information, which could include passwords and encryption certificates. To avoid this potential security issue, you can specify the information on the dbmlsync command line.

See "dbmlsync syntax" [*MobiLink - Client Administration*].

Permissions

Must have DBA authority. Requires exclusive access to all tables referred to in the publication.

Side effects

Automatic commit.

See also

- "ALTER SYNCHRONIZATION SUBSCRIPTION statement [MobiLink]" on page 386
- "DROP SYNCHRONIZATION SUBSCRIPTION statement [MobiLink]" on page 586
- SQL Anywhere MobiLink clients: "Creating synchronization subscriptions" [*MobiLink - Client Administration*]
- UltraLite MobiLink clients: "Designing synchronization in UltraLite" [*UltraLite - Database Management and Reference*]
- "ISYSSYNC system table" on page 807

Standards and compatibility

- **SQL/2003** Vendor extension.

Examples

The following example creates a subscription between the MobiLink user ml_user1 and the publication called sales_publication and sets the memory to 3 MB:

```
CREATE SYNCHRONIZATION SUBSCRIPTION
   TO sales_publication
   FOR ml_user1
   OPTION memory='3m';
```

The following example omits the FOR clause, and so stores settings for the publication called sales_publication:

```
CREATE SYNCHRONIZATION SUBSCRIPTION
   TO sales_publication
   ADDRESS 'host=test.internal;port=2439;
      security=ecc_tls'
   OPTION memory='2m';
```

CREATE SYNCHRONIZATION USER statement [MobiLink]

Use this statement in a SQL Anywhere remote database to create a MobiLink user.

Syntax

CREATE SYNCHRONIZATION USER *ml_username*
[**TYPE** *network-protocol*]
[**ADDRESS** *protocol-options*]
[**OPTION** *option*=*value*, ...]

ml_username: *identifier*

network-protocol :
tcpip
| **http**
| **https**
| **tls**

protocol-options : *string*

value: *string* | *integer*

Parameters

- **ml_username** A name identifying a MobiLink user.

  For more information about MobiLink users, see "Introduction to MobiLink users" [*MobiLink - Client Administration*].

- **TYPE clause** This clause specifies the network protocol to use for synchronization. The default protocol is tcpip.

  For more information about communication protocols, see "CommunicationType (ctp) extended option" [*MobiLink - Client Administration*].

- **ADDRESS clause** This clause specifies *protocol-options* in the form *keyword*=*value*, separated by semi-colons. Which settings you supply depends on the communication protocol you are using (TCPIP, TLS, HTTP, or HTTPS).

  For a complete list of protocol options, see "MobiLink client network protocol option summary" [*MobiLink - Client Administration*].

- **OPTION clause** The OPTION clause allows you to set extended options using *option*=*value* in a comma-separated list.

  The values for each option cannot contain equal signs or semicolons. The database server accepts any option that you enter without checking for its validity. Therefore, if you misspell an option or enter an invalid value, no error message appears until you run the dbmlsync command to perform synchronization.

  Options set for a synchronization user can be overridden in individual subscriptions or on the dbmlsync command line.

  For information about extended options, see "MobiLink SQL Anywhere client extended options" [*MobiLink - Client Administration*].

  The *network-protocol*, *protocol-options*, and *options* can be set in several places.

  For information about how dbmlsync processes options that are specified in different locations, see "Priority order" [*MobiLink - Client Administration*].

  This statement causes options and other information to be stored in the SQL Anywhere ISYSSYNC system table. Anyone with DBA authority for the database can view the information, which could include passwords and encryption certificates. To avoid this potential security issue, you can specify the information on the dbmlsync command line.

  See "dbmlsync syntax" [*MobiLink - Client Administration*].

Permissions

Must have DBA authority.

Side effects

Automatic commit.

See also

- "ALTER SYNCHRONIZATION USER statement [MobiLink]" on page 388
- "DROP SYNCHRONIZATION USER statement [MobiLink]" on page 587
- "Encrypting MobiLink client/server communications" [*SQL Anywhere Server - Database Administration*]
- "ISYSSYNC system table" on page 807

Standards and compatibility

- **SQL/2003** Vendor extension.

Examples

The following example creates a MobiLink user named SSinger, who synchronizes over TCP/IP with a server computer named mlserver.mycompany.com using the password Sam. The use of a password in the user definition is *not* secure.

```
CREATE SYNCHRONIZATION USER SSinger
TYPE http
ADDRESS 'host=mlserver.mycompany.com'
OPTION MobiLinkPwd='Sam';
```

CREATE TABLE statement

Use this statement to create a new table in the database and, optionally, to create a table on a remote server.

Syntax

CREATE [**GLOBAL TEMPORARY**] **TABLE** [**IF NOT EXISTS**] [*owner*.]*table-name*
({ *column-definition* | *table-constraint* | *pctfree* }, ... **)**
[{ **IN** | **ON** } *dbspace-name*]
[**ENCRYPTED**]
[**ON COMMIT** { **DELETE** | **PRESERVE** } **ROWS**
  | **NOT TRANSACTIONAL**]
[**AT** *location-string*]
[**SHARE BY ALL**]

column-definition :
column-name data-type
[**COMPRESSED**]
[**INLINE** { *inline-length* | **USE DEFAULT** }]
[**PREFIX** { *prefix-length* | **USE DEFAULT** }]
[[**NO**] **INDEX**]
[[**NOT**] **NULL**]
[**DEFAULT** *default-value* | **IDENTITY**]
[*column-constraint* ...]

default-value :
 special-value
| *string*
| *global variable*
| [**-**] *number*
| **(** *constant-expression* **)**
| *built-in-function***(** *constant-expression* **)**

```
| AUTOINCREMENT
| CURRENT DATABASE
| CURRENT REMOTE USER
| CURRENT UTC TIMESTAMP
| GLOBAL AUTOINCREMENT [ ( partition-size ) ]
| NULL
| TIMESTAMP
| UTC TIMESTAMP
| LAST USER

special-value:
CURRENT {
  DATE
  | TIME
  | TIMESTAMP
  | UTC TIMESTAMP
  | USER
  | PUBLISHER
}
| USER

column-constraint :
[ CONSTRAINT constraint-name ] {
  UNIQUE [ CLUSTERED ]
 | PRIMARY KEY [ CLUSTERED ] [ ASC | DESC ]
 | REFERENCES table-name [ ( column-name ) ]
   [ MATCH [ UNIQUE ] { SIMPLE | FULL } ]
   [ action-list ] [ CLUSTERED ]
  }
| [ CONSTRAINT constraint-name ] CHECK ( condition )
| COMPUTE ( expression )

table-constraint :
[ CONSTRAINT constraint-name ] {
   UNIQUE [ CLUSTERED ] ( column-name [ ASC | DESC ], ... )
 | PRIMARY KEY [ CLUSTERED ] ( column-name [ ASC | DESC ], ... )
 | CHECK ( condition )
 | foreign-key-constraint
}

foreign-key-constraint :
[ NOT NULL ] FOREIGN KEY [ role-name ]
  [ ( column-name [ ASC | DESC ], ... ) ]
  REFERENCES table-name
  [ ( column-name, ... ) ]
  [ MATCH [ UNIQUE] { SIMPLE | FULL } ]
  [ action-list ] [ CHECK ON COMMIT ] [ CLUSTERED ] [ FOR OLAP WORKLOAD ]

action-list :
[ ON UPDATE action ]
[ ON DELETE action ]

action :
CASCADE
| SET NULL
| SET DEFAULT
| RESTRICT
```

location-string :
 remote-server-name.[*db-name*].[*owner*].*object-name*
| *remote-server-name*;[*db-name*];[*owner*];*object-name*

pctfree : **PCTFREE** *percent-free-space*

percent-free-space : *integer*

Parameters

- **IN clause** Use this clause to specify the dbspace in which the base table is located. If this clause is not specified, then the base table is created in the dbspace specified by the default_dbspace option.

  Temporary tables can only be created in the temporary dbspace. If you are creating a GLOBAL TEMPORARY table, and specify IN, the table is created in the temporary dbspace. If you specify a user-defined dbspace, an error is returned.

  For more information about dbspaces, see:

  - "CREATE DBSPACE statement" on page 439
  - "Using additional dbspaces" [*SQL Anywhere Server - Database Administration*]
  - "default_dbspace option [database]" [*SQL Anywhere Server - Database Administration*]

- **ENCRYPTED clause** The encrypted clause specifies that the table should be encrypted. You must enable table encryption when you create a database if you want to encrypt tables. The table is encrypted using the encryption key and algorithm specified at database creation time. See "Enabling table encryption in the database" [*SQL Anywhere Server - Database Administration*].

- **ON COMMIT clause** The ON COMMIT clause is allowed only for temporary tables. By default, the rows of a temporary table are deleted on COMMIT. If the SHARE BY ALL clause is specified, either ON COMMIT PRESERVE ROWS or NOT TRANSACTIONAL must be specified.

- **NOT TRANSACTIONAL clause** The NOT TRANSACTIONAL clause is allowed when creating a global temporary table. A table created using NOT TRANSACTIONAL is not affected by either COMMIT or ROLLBACK. If the SHARE BY ALL clause is specified, either ON COMMIT PRESERVE ROWS or NOT TRANSACTIONAL must be specified. For information about the benefits of the NOT TRANSACTIONAL clause, see "Working with temporary tables" [*SQL Anywhere Server - SQL Usage*].

- **AT clause** Create a remote table on a different server specified by *location-string*, and a proxy table on the current database that maps to the remote table. The AT clause supports the semicolon (;) as a field delimiter in *location-string*. If no semicolon is present, a period is the field delimiter. This syntax allows file names and extensions to be used in the database and owner fields.

  For example, the following statement maps the table a1 to the Microsoft Access file *mydbfile.mdb*:

  ```
  CREATE TABLE a1
  AT 'access;d:\mydbfile.mdb;;a1';
  ```

  For information about remote servers, see "CREATE SERVER statement" on page 506. For information about proxy tables, see "CREATE EXISTING TABLE statement" on page 455 and "Specify proxy table locations" [*SQL Anywhere Server - SQL Usage*].

  Windows Mobile does not support the AT clause.

Foreign key definitions are ignored on remote tables. Foreign key definitions on local tables that refer to remote tables are also ignored. Primary key definitions are sent to the remote server if the database server supports primary keys.

- **SHARE BY ALL clause** Use this clause only when creating global temporary tables to allow the table to be shared by all connections to the database. If the SHARE BY ALL clause is specified, either ON COMMIT PRESERVE ROWS or NOT TRANSACTIONAL must be specified.

  For information about the characteristics of temporary tables, see "Working with temporary tables" [*SQL Anywhere Server - SQL Usage*].

- **IF NOT EXISTS clause** Use this clause to create permanent, global temporary, and local temporary tables. No changes are made if the named table already exists, and an error is not returned.

  For information about the characteristics of temporary tables, see "Working with temporary tables" [*SQL Anywhere Server - SQL Usage*].

- ***column-definition*** Define a column in the table. The following are part of column definitions.

  - ***column-name*** The column name is an identifier. Two columns in the same table cannot have the same name. See "Identifiers" on page 8.

  - ***data-type*** The type of data stored in the column. See "SQL data types" on page 77.

  - **COMPRESSED** Compress the column. For example, the following statement creates a table, t, with two columns: filename and contents. The contents column is LONG BINARY and is compressed:

    ```
    CREATE TABLE t (
      filename VARCHAR(255),
      contents LONG BINARY COMPRESSED
    );
    ```

- **INLINE and PREFIX clauses** The INLINE clause specifies the maximum BLOB size, in bytes, to store within the row. BLOBs smaller than or equal to the value specified by the INLINE clause are stored within the row. BLOBs that exceed the value specified by the INLINE clause are stored outside the row in table extension pages. Also, a copy of some bytes from the beginning of the BLOB may be kept in the row when a BLOB is larger than the INLINE value. Use the PREFIX clause to specify how many bytes are kept in the row. The PREFIX clause can improve the performance of requests that need the prefix bytes of a BLOB to determine if a row is accepted or rejected.

  The prefix data for a compressed column is stored uncompressed, so if all the data required to satisfy a request is stored in the prefix, no decompression is necessary.

  If neither INLINE nor PREFIX is specified, or if USE DEFAULT is specified, default values are applied as follows:

  - For character data type columns, such as CHAR, NCHAR, and LONG VARCHAR, the default value of INLINE is 256, and the default value of PREFIX is 8.

  - For binary data type columns, such as BINARY, LONG BINARY, VARBINARY, BIT, VARBIT, LONG VARBIT, BIT VARYING, and UUID, the default value of INLINE is 256, and the default value of PREFIX is 0.

> **Note**
> It is strongly recommended that you use the default values unless there are specific circumstances that require a different setting. The default values have been chosen to balance performance and disk space requirements. For example, if you set INLINE to a large value, and all the BLOBs are stored inline, row processing performance may degrade. If you set PREFIX too high, you increase the amount of disk space required to store BLOBs since the prefix data is a duplicate of a portion of the BLOB.

If only one of the values is specified, the other value is automatically set to the largest amount that does not conflict with the specified value. Neither the INLINE nor PREFIX value can exceed the database page size. Also, there is a small amount of overhead reserved in a table page that cannot be used to store row data. Therefore, specifying an INLINE value approximate to the database page size can result in a slightly smaller number of bytes being stored inline.

- **INDEX and NO INDEX clauses** When storing BLOBs (character or binary types only), specify INDEX to create BLOB indexes on inserted values that exceed the internal BLOB size threshold (approximately eight database pages). This is the default behavior.

  BLOB indexes can improve performance when random access searches within the BLOBs are required. However, for some types of BLOB values, such as images and multimedia files that will never require random-access, performance can improve if BLOB indexing is turned off. To turn off BLOB indexing for a column, specify NO INDEX.

  > **Note**
  > A BLOB index is not the same as a table index. A table index is created to index values in one or more columns.

- **NULL and NOT NULL clauses** If NULL is specified, NULL values are allowed in the column. This is the default behavior.

  If NOT NULL is specified, NULL values are not allowed.

  If the column is part of a UNIQUE or PRIMARY KEY constraint, the column cannot contain NULL, even if NULL is specified.

- **DEFAULT clause** For more information about the *special-value*, see "Special values" on page 58.

  If a DEFAULT value is specified, it is used as the value for the column in any INSERT statement that does not specify a value for the column. If no DEFAULT value is specified, it is equivalent to DEFAULT NULL.

  Following is a list of possible values for DEFAULT:

  - **Constant expressions** Constant expressions that do not reference database objects are allowed in a DEFAULT clause, so functions such as GETDATE or DATEADD can be used. If the expression is not a function or simple value, it must be enclosed in parentheses.

  - **CURRENT REMOTE USER** The CURRENT REMOTE USER default is set by the SQL Remote Message Agent (dbremote) when it executes the undocumented REMOTE USER statement. The value will be NULL except for connections from DBREMOTE. See "Message Agent (dbremote)" *[SQL Remote]*.

- **AUTOINCREMENT** When using AUTOINCREMENT, the column must be one of the integer data types, or an exact numeric type.

  On inserts into the table, if a value is not specified for the AUTOINCREMENT column, a unique value larger than any other value in the column is generated. If an INSERT specifies a value for the column that is larger than the current maximum value for the column, that value is inserted and then used as a starting point for subsequent inserts.

  Deleting rows does not decrement the AUTOINCREMENT counter. Gaps created by deleting rows can only be filled by explicit assignment when using an insert. After an explicit insert of a column value less than the maximum, subsequent rows without explicit assignment are still automatically incremented with a value of one greater than the previous maximum.

  You can find the most recently inserted value of the column by inspecting the @@identity global variable. See "@@identity global variable" on page 71.

  AUTOINCREMENT values are maintained as signed 64-bit integers, corresponding to the data type of the max_identity column in the SYSTABCOL system view. When the next value to be generated exceeds the maximum value that can be stored in the column to which the AUTOINCREMENT is assigned, NULL is returned. If the column has been declared to not allow NULLs, as is the case for primary key columns, a SQL error is generated.

  For information about rebuilding databases that use AUTOINCREMENT, see "Reloading tables with autoincrement columns" [*SQL Anywhere 11 - Changes and Upgrading*].

- **IDENTITY** The IDENTITY default is a Transact-SQL-compatible alternative to using the AUTOINCREMENT default. In SQL Anywhere, a column defined as IDENTITY is implemented as AUTOINCREMENT. See "The special IDENTITY column" [*SQL Anywhere Server - SQL Usage*].

- **GLOBAL AUTOINCREMENT** This default is intended for use when multiple databases are used in a MobiLink synchronization environment or SQL Remote replication.

  This option is similar to AUTOINCREMENT, except that the domain is partitioned. Each partition contains the same number of values. You assign each copy of the database a unique global database identification number. SQL Anywhere supplies default values in a database only from the partition uniquely identified by that database's number.

  The partition size can be specified in parentheses immediately following the AUTOINCREMENT keyword. The partition size can be any positive integer, although the partition size is generally chosen so that the supply of numbers within any one partition will rarely, if ever, be exhausted.

  If the column is of type BIGINT or UNSIGNED BIGINT, the default partition size is 2^{32} = 4294967296; for columns of all other types, the default partition size is 2^{16} = 65536. Since these defaults may be inappropriate, especially if your column is not of type INT or BIGINT, it is best to specify the partition size explicitly.

  When using this default, the value of the public option global_database_id in each database must be set to a unique, non-negative integer. This value uniquely identifies the database and indicates from which partition default values are to be assigned. The range of allowed values is $np + 1$ to $p(n + 1)$, where n is the value of the public option global_database_id and p is the partition size. For example, if you define the partition size to be 1000 and set global_database_id to 3, then the range is from 3001 to 4000.

If the previous value is less than *p*(*n* + 1), the next default value is one greater than the previous largest value in the column. If the column contains no values, the first default value is *np* + 1. Default column values are not affected by values in the column outside the current partition; that is, by numbers less than *np* + 1 or greater than *p*(*n* + 1). Such values may be present if they have been replicated from another database via MobiLink or SQL Remote.

You can find the most recently inserted value of the column by inspecting the @@identity global variable.

GLOBAL AUTOINCREMENT values are maintained as signed 64-bit integers, corresponding to the data type of the max_identity column in the SYSTABCOL system view. When the supply of values within the partition has been exhausted, NULL is returned. If the column has been declared to not allow NULLs, as is the case for primary key columns, a SQL error is generated. In this case, a new value of global_database_id should be assigned to the database to allow default values to be chosen from another partition. To detect that the supply of unused values is low and handle this condition, create an event of type GlobalAutoincrement. See "Understanding events" [*SQL Anywhere Server - Database Administration*].

Because the public option global_database_id cannot be set to a negative value, the values chosen are always positive. The maximum identification number is restricted only by the column data type and the partition size.

If the public option global_database_id is set to the default value of 2147483647, a NULL value is inserted into the column. If NULL values are not permitted, attempting to insert the row causes an error.

- **TIMESTAMP clause** Provides a way of indicating when each row in the table was last modified. When a column is declared with DEFAULT TIMESTAMP, a default value is provided for inserts, and the value is updated with the current date and time whenever the row is updated.

  To provide a default value on insert, but not update the column whenever the row is updated, use DEFAULT CURRENT TIMESTAMP instead of DEFAULT TIMESTAMP.

  For more information about timestamp columns, see "The special Transact-SQL timestamp column and data type" [*SQL Anywhere Server - SQL Usage*].

  Columns declared with DEFAULT TIMESTAMP contain unique values, so that applications can detect near-simultaneous updates to the same row. If the current timestamp value is the same as the last value, it is incremented by the value of the default_timestamp_increment option. See "default_timestamp_increment option [database] [MobiLink client]" [*SQL Anywhere Server - Database Administration*].

  You can automatically truncate timestamp values in SQL Anywhere based on the default_timestamp_increment option. This is useful for maintaining compatibility with other database software that records less precise timestamp values. See "default_timestamp_increment option [database] [MobiLink client]" [*SQL Anywhere Server - Database Administration*].

  The global variable @@dbts returns a TIMESTAMP value representing the last value generated for a column using DEFAULT TIMESTAMP. See "Global variables" on page 68.

- **UTC TIMESTAMP clause** The behavior of UTC TIMESTAMP is the same as TIMESTAMP except that a UTC TIMESTAMP value is in Coordinated Universal (UTC) time.

- **string** See "Strings" on page 9.

- **global-variable** See "Global variables" on page 68.
- **column-constraint and table-constraint clauses** Column and table constraints help ensure the integrity of data in the database. If a statement would cause a violation of a constraint, execution of the statement does not complete, any changes made by the statement before error detection are undone, and an error is reported. There are two classes of constraints that can be created: **check constraint**, and **referential integrity (RI) constraints**. Check constraints are used to specify conditions that must be satisfied by values of columns being put into the database. RI constraints establish a relationship between data in different tables that must be maintained in addition to specifying uniqueness requirements for data.

  There are three types of RI constraints: primary key, foreign key, and unique constraint. When you create an RI constraint (primary key, foreign key or unique constraint), the database server enforces the constraint by implicitly creating an index on the columns that make up the key of the constraint. The index is created on the key for the constraint as specified. A key consists of an ordered list of columns and a sequencing of values (ASC/DESC) for each column.

  Constraints can be specified on columns or tables. Generally speaking, a column constraint is one that refers to one column in a table, while a table constraint can refer to one or more columns in a table.

  - **PRIMARY KEY constraint** A primary key uniquely defines each row in the table. Primary keys comprise one or more columns. A table cannot have more than one primary key. In a *column-constraint* clause, specifying PRIMARY KEY indicates that the column is the primary key for the table. In a *table-constraint,* you use the PRIMARY KEY clause to specify one or more columns that, when combined in the specified order, make up the primary key for the table.

    The ordering of columns in a primary key need not match the respective ordinal numbers of the columns. That is, the columns in a primary key need not have the same physical order in the row. Additionally, you cannot specify duplicate column names.

    When you create a primary key, an index for the key is automatically created. You can specify the sequencing of values in the index by specifying ASC (ascending) or DESC (descending) for each column. You can also specify whether to cluster the index, using the CLUSTERED keyword. For more information about the CLUSTERED option and clustered indexes, see "Using clustered indexes" [*SQL Anywhere Server - SQL Usage*].

    Columns included in primary keys cannot allow NULL. Each row in the table has a unique primary key value.

    It is recommended that you do not use approximate data types such as FLOAT and DOUBLE for primary keys. Approximate numeric data types are subject to rounding errors after arithmetic operations.

  - **Foreign key** A foreign key restricts the values for a set of columns to match the values in a primary key or a unique constraint of another table (the primary table). For example, a foreign key constraint could be used to ensure that a customer number in an invoice table corresponds to a customer number in the Customers table.

    For information about how the database server can select columns automatically for the foreign key, see "Omitting column names at foreign key creation (SQL)" [*SQL Anywhere Server - SQL Usage*].

    The foreign key column order does not need to reflect the order of columns in the table.

    Duplicate column names are not allowed in the foreign key specification.

The default *action* is RESTRICT if no action is specified for an UPDATE or DELETE operation.

When you create a foreign key, an index for the key is automatically created. You can specify the sequencing of values in the index by specifying ASC (ascending) or DESC (descending) for each column. You can also specify whether to cluster the index, using the CLUSTERED keyword. For more information about the CLUSTERED option and clustered indexes, see "Using clustered indexes" [*SQL Anywhere Server - SQL Usage*].

A temporary table cannot have a foreign key that references a base table and a base table cannot have a foreign key that references a temporary table.

- **NOT NULL option** Disallow NULLs in the foreign key columns. A NULL in a foreign key means that no row in the primary table corresponds to this row in the foreign table.
- **role-name clause** The role name is the name of the foreign key. The main function of the role name is to distinguish between two foreign keys to the same table. If no role name is specified, the role name is assigned as follows:
    1. If there is no foreign key with a role name the same as the table name, the table name is assigned as the role name.
    2. If the table name is already taken, the role name is the table name concatenated with a zero-padded three-digit number unique to the table.
- **REFERENCES clause** A foreign key constraint can be implemented using a REFERENCES column constraint (single column only) or a FOREIGN KEY table constraint, in which case the constraint can specify one or more columns. If you specify *column-name* in a REFERENCES column constraint, it must be a column in the primary table, must be subject to a unique constraint or primary key constraint, and that constraint must consist of only that one column. If you do not specify *column-name*, the foreign key column references the single primary key column of the primary table.
- **MATCH clause** The MATCH clause allows you to control what is considered a match when using a multi-column foreign key. It also allows you to specify uniqueness for the key, thereby eliminating the need to declare uniqueness separately. Following is a list match types you can specify:
    - **UNIQUE option** The referencing table can have only one match for non-NULL key values (keys with at least one non-NULL column value are implicitly unique).
    - **SIMPLE option** A match occurs for a row in the referencing table if at least one column in the key is NULL, or all the column values match the corresponding column values present in a row of the referenced table.
    - **FULL option** A match occurs for a row in the referencing table if all column values in the key are NULL, or if all the column values match the values present in a row of the referenced table.
    - **SIMPLE UNIQUE option** A match occurs if the criteria for both SIMPLE and UNIQUE are met.
    - **FULL UNIQUE option** A match occurs if the criteria for both FULL and UNIQUE are met.

- ○ **UNIQUE constraint** In a *column-constraint* clause, a UNIQUE constraint specifies that the values in the column must be unique. In a *table-constraint* clause, the UNIQUE constraint identifies one or more columns that uniquely identify each row in the table. No two rows in the table can have the same values in all the named column(s). A table can have more than one UNIQUE constraint.

  A UNIQUE constraint is not the same as a unique index. Columns of a unique index are allowed to be NULL, while columns in a UNIQUE constraint are not. Also, a foreign key can reference either a primary key or a UNIQUE constraint, but cannot reference a unique index since a unique index can include multiple instances of NULL.

  Columns in a UNIQUE constraint can be specified in any order. Additionally, you can specify the sequencing of values in the corresponding index that is automatically created, by specifying ASC (ascending) or DESC (descending) for each column. You cannot specify duplicate column names, however.

  It is recommended that you do not use approximate data types such as FLOAT and DOUBLE for columns with unique constraints. Approximate numeric data types are subject to rounding errors after arithmetic operations.

  You can also specify whether to cluster the constraint, using the CLUSTERED keyword. For more information about the CLUSTERED option, see "Using clustered indexes" [*SQL Anywhere Server - SQL Usage*].

  For information about unique indexes, see "CREATE INDEX statement" on page 470.

- ○ **CHECK constraint** This constraint allows arbitrary conditions to be verified. For example, a CHECK constraint could be used to ensure that a column called Sex only contains the values M or F.

  No row in a table is allowed to violate a CHECK constraint. If an INSERT or UPDATE statement would cause a row to violate the constraint, the operation is not permitted and the effects of the statement are undone. The change is rejected only if a CHECK constraint condition evaluates to FALSE, and the change is allowed if a CHECK constraint condition evaluates to TRUE or UNKNOWN.

  For more information about TRUE, FALSE, and UNKNOWN conditions, see "NULL value" on page 73, and "Search conditions" on page 35.

- ○ **COMPUTE clause** The COMPUTE clause is only for use in a *column-constraint* clause. When a column is created using a COMPUTE clause, its value in any row is the value of the supplied expression. Columns created with this constraint are read-only columns for applications: the value is changed by the database server whenever the row is modified. The COMPUTE expression should not return a non-deterministic value. For example, it should not include a special value such as CURRENT TIMESTAMP, or a non-deterministic function. If a COMPUTE expression returns a non-deterministic value, then it cannot be used to match an expression in a query. See "Working with computed columns" [*SQL Anywhere Server - SQL Usage*].

  The COMPUTE clause is ignored for remote tables.

  Any UPDATE statement that attempts to change the value of a computed column fires any triggers associated with the column.

- ● **CHECK ON COMMIT option** The CHECK ON COMMIT option overrides the wait_for_commit database option, and causes the database server to wait for a COMMIT before checking RESTRICT actions on a foreign key. The CHECK ON COMMIT option does not delay CASCADE, SET NULL, or SET DEFAULT actions.

If you use CHECK ON COMMIT without specifying any actions, then RESTRICT is implied as an action for UPDATE and DELETE.

- **FOR OLAP WORKLOAD option** When you specify FOR OLAP WORKLOAD in the REFERENCES clause of a foreign key definition, the database server performs certain optimizations and gathers statistics on the key to help improve performance for OLAP workloads, particularly when the optimization_workload is set to OLAP. See "optimization_workload option [database]" [*SQL Anywhere Server - Database Administration*].

  For more information, see "OLAP support" [*SQL Anywhere Server - SQL Usage*].

- **PCTFREE clause** Specifies the percentage of free space you want to reserve for each table page. The free space is used if rows increase in size when the data is updated. If there is no free space in a table page, every increase in the size of a row on that page requires the row to be split across multiple table pages, causing row fragmentation and possible performance degradation.

  The value *percent-free-space* is an integer between 0 and 100. The former value specifies that no free space is to be left on each page—each page is to be fully packed. A high value causes each row to be inserted into a page by itself. If PCTFREE is not set, or is later dropped, the default PCTFREE value is applied according to the database page size (200 bytes for a 4 KB (and up) page size). The value for PCTFREE is stored in the ISYSTAB system table.

Remarks

The CREATE TABLE statement creates a new table. A table can be created for another user by specifying an owner name. If GLOBAL TEMPORARY is specified, the table is a temporary table. Otherwise, the table is a base table.

Tables created by preceding the table name in a CREATE TABLE statement with a pound sign (#) are declared temporary tables, which are available only in the current connection. Temporary tables created with the pound sign (#) are identical to those created with the ON COMMIT PRESERVE ROWS clause. See "DECLARE LOCAL TEMPORARY TABLE statement" on page 557.

Columns in SQL Anywhere allow NULLs by default. This setting can be controlled using the allow_nulls_by_default database option. See "allow_nulls_by_default option [compatibility]" [*SQL Anywhere Server - Database Administration*].

Permissions

Must have RESOURCE authority.

Must have DBA authority to create a table for another user.

Side effects

Automatic commit.

See also

- "CREATE LOCAL TEMPORARY TABLE statement" on page 474
- "ALTER TABLE statement" on page 389
- "CREATE DBSPACE statement" on page 439
- "CREATE EXISTING TABLE statement" on page 455
- "DECLARE LOCAL TEMPORARY TABLE statement" on page 557
- "DROP TABLE statement" on page 588
- "Special values" on page 58
- "SQL data types" on page 77
- "Create tables" [*SQL Anywhere Server - SQL Usage*]
- "allow_nulls_by_default option [compatibility]" [*SQL Anywhere Server - Database Administration*]
- "Working with temporary tables" [*SQL Anywhere Server - SQL Usage*]

Standards and compatibility

- **SQL/2003** Core feature.

  The following are vendor extensions:

  - The { IN | ON } *dbspace-name* clause.
  - The ON COMMIT clause.
  - Some of the default values.

Examples

The following example creates a table for a library database to hold book information.

```
CREATE TABLE library_books (
   -- NOT NULL is assumed for primary key columns
   isbn CHAR(20) PRIMARY KEY,
   copyright_date DATE,
   title CHAR(100),
   author CHAR(50),
   -- column(s) corresponding to primary key of room
   -- are created automatically
   FOREIGN KEY location REFERENCES room
);
```

The following example creates a table for a library database to hold information on borrowed books. The default value for date_borrowed indicates that the book is borrowed on the day the entry is made. The date_returned column is NULL until the book is returned.

```
CREATE TABLE borrowed_book (
   date_borrowed DATE NOT NULL DEFAULT CURRENT DATE,
   date_returned DATE,
   book CHAR(20)
   REFERENCES library_books (isbn),
   -- The check condition is UNKNOWN until
   -- the book is returned, which is allowed
CHECK( date_returned >= date_borrowed )
);
```

The following example creates tables for a sales database to hold order and order item information.

```
CREATE TABLE Orders (
   order_num INTEGER NOT NULL PRIMARY KEY,
   date_ordered DATE,
   name CHAR(80)
);
CREATE TABLE Order_item (
   order_num INTEGER NOT NULL,
   item_num SMALLINT NOT NULL,
   PRIMARY KEY ( order_num, item_num ),
   -- When an order is deleted, delete all of its
   -- items.
   FOREIGN KEY ( order_num )
   REFERENCES Orders ( order_num )
   ON DELETE CASCADE
);
```

The following example creates a table named t1 at the remote server SERVER_A and creates a proxy table named t1 that is mapped to the remote table.

```
CREATE TABLE t1
( a INT,
  b CHAR(10) )
AT 'SERVER_A.db1.joe.t1';
```

CREATE TEXT CONFIGURATION statement

Creates a text configuration object.

Syntax

CREATE TEXT CONFIGURATION [*owner.*]*new-config-name*
  FROM [*owner*.]*existing-config-name*

Parameters

- **FROM clause** Specify the name of a text configuration object to use as the template for creating the new one. The names of the default text configuration objects are default_char and default_nchar. See "Default text configuration objects" [*SQL Anywhere Server - SQL Usage*].

Remarks

You create a text configuration object using another text configuration object as a template and then alter the options as needed using the ALTER TEXT CONFIGURATION statement.

To view the list of all text configuration objects and their settings in the database, query the SYSTEXTCONFIG system view. See "SYSTEXTCONFIG system view" on page 1034.

Permissions

Must have DBA or RESOURCE authority.

All text configuration objects have PUBLIC access. Any user with permission to create a text index can use any text configuration object.

Side effects

Automatic commit

See also

- "Full text searching" [*SQL Anywhere Server - SQL Usage*]
- "Text configuration object settings" [*SQL Anywhere Server - SQL Usage*]
- "Tutorial: Performing a full text search on a GENERIC text index" [*SQL Anywhere Server - SQL Usage*]
- "Tutorial: Performing a fuzzy full text search" [*SQL Anywhere Server - SQL Usage*]
- "ALTER TEXT CONFIGURATION statement" on page 397
- "DROP TEXT CONFIGURATION statement" on page 589
- "sa_refresh_text_indexes system procedure" on page 926

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following CREATE TEXT CONFIGURATION statement creates a text configuration object, max_term_sixteen, using the default_char text configuration object. The subsequent ALTER TEXT CONFIGURATION statement changes the maximum term length for max_term_sixteen to 16.

```
CREATE TEXT CONFIGURATION max_term_sixteen FROM default_char;
ALTER TEXT CONFIGURATION max_term_sixteen
   MAXIMUM TERM LENGTH 16;
```

CREATE TEXT INDEX statement

Creates a text index.

Syntax

CREATE TEXT INDEX *text-index-name*
ON [*owner*.]*table-name***(** *column-name*, ... **)**
[**IN** *dbspace-name*]
[**CONFIGURATION** [*owner*.]*text-configuration-name*]
[{ **IMMEDIATE REFRESH**
  | **MANUAL REFRESH**
  | **AUTO REFRESH** [**EVERY** *integer* { **MINUTES** | **HOURS** }]]
}

Parameters

- **ON clause** Use this clause to specify the table and columns on which to build the text index.

- **IN clause** Use this clause to specify the dbspace in which the text index is located. If this clause is not specified, then the text index is created in the dbspace specified by the default_dbspace option.

- **CONFIGURATION clause** Use this clause to specify the text configuration object to use when creating the text index. If this clause is not specified, the default_nchar text configuration object is used if any of the columns in the index are NCHAR; otherwise, the default_char text configuration object is used.

- **REFRESH clause** Use this clause to specify the refresh type for the text index. If you do not specify a REFRESH clause, IMMEDIATE REFRESH is used as the default. Following are the list of refresh types you can specify:
  - **IMMEDIATE REFRESH** Specify IMMEDIATE REFRESH to refresh the text index each time changes in the underlying table impact data in the text index.
  - **AUTO REFRESH** Use this clause to refresh the materialized view automatically using an internal server event. Use the EVERY subclause to specify the refresh interval in minutes or hours. If you specify AUTO REFRESH without supplying interval information, the database server refreshes the text index every 60 minutes. A text index may be refreshed earlier than specified by the AUTO REFRESH clause if the pending_size value, as returned by the sa_text_index_stats system procedure, exceeds 20% of the text index size at the last refresh. An internal event executes once per minute to check this condition for all AUTO REFRESH text indexes.
  - **MANUAL REFRESH** Use this clause to specify that the text index is refreshed manually.

  For more information about refresh types, see "Text index refresh types" [*SQL Anywhere Server - SQL Usage*].

Remarks

You cannot create a text index on views, materialized views, or temporary tables.

An IMMEDIATE REFRESH text index is populated at creation time and an exclusive lock is held on the table during this initial refresh. IMMEDIATE REFRESH text indexes provide full support for queries that use snapshot isolation.

MANUAL and AUTO REFRESH text indexes must be initialized (refreshed) after creation.

Refreshes for AUTO REFRESH text indexes scan the table using isolation level 0. See "isolation_level option [database] [compatibility]" [*SQL Anywhere Server - Database Administration*].

Once a text index is created, you cannot change it to, or from, being defined as IMMEDIATE REFRESH. If either of these changes is required, you must drop and recreate the text index.

You can choose to manually refresh an AUTO REFRESH text index using the REFRESH TEXT INDEX statement. See "REFRESH TEXT INDEX statement" on page 706.

To view text indexes and the text configuration objects they refer to, see "View text indexes in the database" [*SQL Anywhere Server - SQL Usage*].

Permissions

Must be the owner of the underlying table, or have DBA authority, or have REFERENCES permission.

This statement cannot be executed when there are cursors opened with the WITH HOLD clause that use either statement or transaction snapshots. See "Snapshot isolation" [*SQL Anywhere Server - SQL Usage*].

Side effects

Automatic commit

See also

- "Tutorial: Performing a full text search on a GENERIC text index" [*SQL Anywhere Server - SQL Usage*]
- "Tutorial: Performing a fuzzy full text search" [*SQL Anywhere Server - SQL Usage*]
- "Text indexes" [*SQL Anywhere Server - SQL Usage*]
- "SYSTEXTCONFIG system view" on page 1034
- "CREATE TEXT INDEX statement" on page 536
- "ALTER TEXT INDEX statement" on page 399
- "DROP TEXT INDEX statement" on page 590
- "REFRESH TEXT INDEX statement" on page 706
- "TRUNCATE TEXT INDEX statement" on page 770
- "sa_char_terms system procedure" on page 843
- "sa_nchar_terms system procedure" on page 914
- "sa_refresh_text_indexes system procedure" on page 926
- "sa_text_index_stats system procedure" on page 957

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following example creates a text index, myTxtIdx, on the Description column of the MarketingInformation table in the demo database. The MarketingTextConfig text configuration object is used, and the refresh interval is set to every 24 hours.

```
CREATE TEXT INDEX myTxtIdx ON MarketingInformation ( Description )
   CONFIGURATION MarketingTextConfig
   AUTO REFRESH EVERY 24 HOURS;
```

CREATE TRIGGER statement

Use this statement to create a trigger on a table.

Syntax

CREATE [**OR REPLACE**] **TRIGGER** *trigger-name trigger-type*
{ *trigger-event-list* | **UPDATE OF** *column-list* }
[**ORDER** *integer*] **ON** *table-name*
[**REFERENCING** [**OLD AS** *old-name*]
   [**NEW AS** *new-name*]
   [**REMOTE AS** *remote-name*]]
[**FOR EACH** { **ROW** | **STATEMENT** }]
[**WHEN (** *search-condition* **)**]
trigger-body

column-list : *column-name*[, ...]

trigger-type :
BEFORE
| **AFTER**

| INSTEAD OF
| RESOLVE

trigger-event-list : *trigger-event*[, ...]

trigger-event :
DELETE
| INSERT
| UPDATE

trigger-body : a BEGIN statement. See "BEGIN statement" on page 413.

Parameters

- **OR REPLACE clause** Specifying OR REPLACE (CREATE OR REPLACE TRIGGER) creates a new trigger, or replaces an existing trigger with the same name.
- ***trigger-event*** Triggers can be fired by the following events. You can define either multiple triggers for DELETE, INSERT, or UPDATE events, or one trigger for an UPDATE OF *column-list* event:
  - **DELETE** Invoked whenever a row of the associated table is deleted.
  - **INSERT** Invoked whenever a new row is inserted into the table associated with the trigger.
  - **UPDATE** Invoked whenever a row of the associated table is updated.
  - **UPDATE OF *column-list*** Invoked whenever a row of the associated table is updated and a column in the *column-list* is modified. This type of trigger event cannot be used in a *trigger-event-list*; it must be the only trigger event defined for the trigger. This clause cannot be used in an INSTEAD OF trigger.

    You can write separate triggers for each event that you need to handle or, if you have some shared actions and some actions that depend on the event, you can create a trigger for all events and use an IF statement to distinguish the action taking place. For more information about trigger operations, see "Trigger operation conditions" on page 55.
- ***trigger-type*** Row-level triggers can be defined to execute BEFORE, AFTER, or INSTEAD OF an insert, update, or delete operation. Statement-level triggers can be defined to execute INSTEAD OF or AFTER the statement.

  BEFORE UPDATE triggers fire any time an UPDATE occurs on a row, whether the new value differs from the old value. That is, if a *column-list* is specified for a BEFORE UPDATE trigger, the trigger fires if any of the columns in *column-list* appear in the SET clause of the UPDATE statement. If a *column-list* is specified for an AFTER UPDATE trigger, the trigger is fired only if the value of any of the columns in *column-list* is *changed* by the UPDATE statement.

  INSTEAD OF triggers are the only form of trigger that you can define on a regular view. INSTEAD OF triggers replace the triggering action with another action. When an INSTEAD OF trigger fires, the triggering action is skipped and the specified action is performed. INSTEAD OF triggers can be defined at a row-level or a statement-level. A statement-level INSTEAD OF trigger replaces the entire statement, including all row-level operations. If a statement-level INSTEAD OF trigger fires, no row-level triggers fire as a result of that statement. However, the body of the statement-level trigger could perform other operations that, in turn, cause other row-level triggers to fire.

If you are defining an INSTEAD OF trigger, you cannot use the UPDATE OF *column-list* clause, the ORDER clause, or the WHEN clause.

For more information about the capabilities of, and restrictions for, INSTEAD OF triggers, see "INSTEAD OF triggers" [*SQL Anywhere Server - SQL Usage*].

The RESOLVE trigger type is for use with SQL Remote: it fires before row-level UPDATE or UPDATE OF *column-list* only.

- **FOR EACH clause** To declare a trigger as a row-level trigger, use the FOR EACH ROW clause. To declare a trigger as a statement-level trigger, you can either use a FOR EACH STATEMENT clause or omit the FOR EACH clause. For clarity, it is recommended that you specify the FOR EACH STATEMENT clause if you are declaring a statement-level trigger.

- **ORDER clause** When defining additional triggers of the same type (insert, update, or delete) to fire at the same time (before, after, or resolve), you must specify an ORDER clause to tell the database server the order in which to fire the triggers. Order numbers must be unique among same-type triggers configured to fire at the same time. If you specify an order number that is not unique, an error is returned. Order numbers do not need to be in consecutive order (for example, you could specify 1, 12, 30). The database server fires the triggers starting with the lowest number.

  If you omit the ORDER clause, or specify 0, the database server assigns the order of 1. However, if another same-type trigger is already set to 1, an error is returned.

  When adding additional triggers, you may need to modify the existing same-type triggers for the event, depending on whether the actions of the triggers interact. If they do not interact, the new trigger must have an ORDER value higher than the existing triggers. If they do interact, you need to consider what the other triggers do, and you may need to change the order in which they fire.

  The ORDER clause is not supported for INSTEAD OF triggers since there can only be one INSTEAD OF trigger of each type (insert, update, or delete) defined on a table or view.

- **REFERENCING clause** The REFERENCING OLD and REFERENCING NEW clauses allow you to refer to the inserted, deleted, or updated rows. For the purposes of this clause, an UPDATE is treated as a delete followed by an insert.

  An INSERT takes the REFERENCING NEW clause, which represents the inserted row. There is no REFERENCING OLD clause.

  A DELETE takes the REFERENCING OLD clause, which represents the deleted row. There is no REFERENCING NEW clause.

  An UPDATE takes the REFERENCING OLD clause, which represents the row before the update, and it takes the REFERENCING NEW clause, which represents the row after the update.

  The meaning of REFERENCING OLD and REFERENCING NEW differs, depending on whether the trigger is a row-level or a statement-level trigger. For row-level triggers, the REFERENCING OLD clause allows you to refer to the values in a row prior to an update or delete, and the REFERENCING NEW clause allows you to refer to the inserted or updated values. The OLD and NEW rows can be referenced in BEFORE and AFTER triggers. The REFERENCING NEW clause allows you to modify the new row in a BEFORE trigger before the insert or update operation takes place.

For statement-level triggers, the REFERENCING OLD and REFERENCING NEW clauses refer to declared temporary tables holding the old and new values of the rows. The default names for these tables are deleted and inserted.

The REFERENCING REMOTE clause is for use with SQL Remote. It allows you to refer to the values in the VERIFY clause of an UPDATE statement. It should be used only with RESOLVE UPDATE or RESOLVE UPDATE OF column-list triggers.

- **WHEN clause** The trigger fires only for rows where the search-condition evaluates to true. The WHEN clause can be used only with row level triggers. This clause cannot be used in an INSTEAD OF trigger.
- **trigger-body** The trigger body contains the actions to take when the triggering action occurs, and consists of a BEGIN statement. See "BEGIN statement" on page 413.

  You can include trigger operation conditions in the BEGIN statement. Trigger operation conditions carry out actions depending on the trigger event that caused the trigger to fire. For example, if the trigger is defined to fire for both updates and deletes, you can specify different actions for the two conditions. For more information about trigger operation conditions, including an example, see "Trigger operation conditions" on page 55.

Remarks

The CREATE TRIGGER statement creates a trigger associated with a table in the database, and stores the trigger in the database.

You cannot define a trigger on a materialized view. If you do, a SQLE_INVALID_TRIGGER_MATVIEW error is returned.

The trigger is declared as either a row-level trigger, in which case it executes before or after each row is modified, or as a statement-level trigger, in which case it executes after the entire triggering statement is completed.

Permissions

Must have RESOURCE authority and have ALTER permissions on the table, or must be the owner of the table or have DBA authority. CREATE TRIGGER puts a table lock on the table, and requires exclusive use of the table.

Side effects

Automatic commit.

See also

- "BEGIN statement" on page 413
- "CREATE PROCEDURE statement (web services)" on page 495
- "CREATE TRIGGER statement [T-SQL]" on page 544
- "DROP TRIGGER statement" on page 591
- "Using procedures, triggers, and batches" [*SQL Anywhere Server - SQL Usage*]
- "UPDATE statement" on page 777

Standards and compatibility

- **SQL/2003** Persistent Stored Module feature. Some clauses are vendor extensions.

Example

This example creates a statement-level trigger. You must first create a table:

```
CREATE TABLE t0
( id integer NOT NULL,
 times timestamp NULL DEFAULT current timestamp,
 remarks text NULL,
 PRIMARY KEY ( id )
);
```

Next, create a statement-level trigger for this table:

```
CREATE TRIGGER myTrig AFTER INSERT ORDER 4 ON t0
REFERENCING NEW AS new_name
FOR EACH STATEMENT
BEGIN
  DECLARE @id1 INTEGER;
  DECLARE @times1 TIMESTAMP;
  DECLARE @remarks1 LONG VARCHAR;
  DECLARE @err_notfound EXCEPTION FOR SQLSTATE VALUE '02000';
//declare a cursor for table new_name
  DECLARE new1 CURSOR FOR
   SELECT id, times, remarks FROM new_name;
  OPEN new1;
 //Open the cursor, and get the value
  LoopGetRow:
  LOOP
      FETCH NEXT new1 INTO @id1, @times1,@remarks1;
      IF SQLSTATE = @err_notfound THEN
   LEAVE LoopGetRow
      END IF;
      //print the value or for other use
      PRINT (@remarks1);
  END LOOP LoopGetRow;
  CLOSE new1
END;
```

The following example replaces the myTrig trigger created in the previous example.

```
CREATE OR REPLACE TRIGGER myTrig AFTER INSERT ORDER 4 ON t0
REFERENCING NEW AS new_name
FOR EACH STATEMENT
BEGIN
FOR L1 AS new1 CURSOR FOR
   SELECT id, times, remarks FROM new_name;
DO
     //print the value or for other use
     PRINT (@remarks1);
END FOR;
```

The next example shows how you can use REFERENCING NEW in a BEFORE UPDATE trigger. This example ensures that postal codes in the new Employees table are in uppercase:

```
CREATE TRIGGER emp_upper_postal_code
BEFORE UPDATE OF PostalCode
ON Employees
REFERENCING NEW AS new_emp
```

```
FOR EACH ROW
WHEN ( ISNUMERIC( new_emp.PostalCode ) = 0 )
BEGIN
   -- Ensure postal code is uppercase (employee might be
   -- in Canada where postal codes contain letters)
   SET new_emp.PostalCode = UPPER(new_emp.PostalCode)
END;

UPDATE Employees SET state='ON', PostalCode='n2x 4y7' WHERE EmployeeID=191;
SELECT PostalCode FROM Employees WHERE EmployeeID = 191;
```

The next example shows how you can use REFERENCING OLD in a BEFORE DELETE trigger. This example prevents deleting an employee from the Employees table who has not been terminated.

```
CREATE TRIGGER TR_check_delete_employee
BEFORE DELETE
ON Employees
REFERENCING OLD AS current_employees
FOR EACH ROW /* WHEN( search_condition ) */
BEGIN
   IF current_employees.TerminationDate IS NULL THEN
    RAISERROR 30001 'You cannot delete an employee who has not been fired';
    END IF;
END;
```

The next example shows how you can use REFERENCING NEW and REFERENCING OLD in a BEFORE UPDATE trigger. This example prevents a decrease in an employee's salary.

```
CREATE TRIGGER TR_check_salary_decrease
    BEFORE UPDATE
      ON Employees
    REFERENCING OLD AS before_update
    NEW AS after_update
FOR EACH ROW
BEGIN
   IF after_update.salary < before_update.salary THEN
    RAISERROR 30002 'You cannot decrease a salary';
    END IF;
END;
```

The next example shows how you can use REFERENCING NEW and REFERENCING OLD in a BEFORE UPDATE trigger. This example also disallows decreasing an employee's salary, but this trigger is more efficient because it fires only when the salary column is updated.

```
CREATE TRIGGER TR_check_salary_decrease_column
    BEFORE UPDATE OF Salary
      ON Employees
    REFERENCING OLD AS before_update
    NEW AS after_update
FOR EACH ROW /* WHEN( search_condition ) */
BEGIN
   IF after_update.salary < before_update.salary THEN
    RAISERROR 30002 'You cannot decrease a salary';
End IF;
END;
```

The next example shows how you can use REFERENCING NEW and in a BEFORE INSERT and UPDATE trigger. The following example creates a trigger that will fire before a row in the SalesOrderItems table is inserted or updated.

```
CREATE TRIGGER TR_update_date
   BEFORE INSERT, UPDATE
```

```
     ON SalesOrderItems
   REFERENCING NEW AS new_row
FOR EACH ROW
BEGIN
   SET new_row.ShipDate = CURRENT TIMESTAMP;
END;
```

CREATE TRIGGER statement [T-SQL]

Use this statement to create a new trigger in the database in a manner compatible with Adaptive Server Enterprise.

Syntax 1

CREATE TRIGGER [*owner*.]*trigger_name*
ON [*owner*.]*table_name*
FOR { **INSERT**, **UPDATE**, **DELETE** }
AS *statement-list*

Syntax 2

CREATE TRIGGER [*owner*.]*trigger_name*
ON [*owner*.]*table_name*
FOR {**INSERT**, **UPDATE**}
AS
[**IF UPDATE (** *column_name* **)**
[{ **AND** | **OR** } **UPDATE (** *column_name* **)**] ...]
  statement-list
[**IF UPDATE (** *column_name* **)**
[{ **AND** | **OR**} **UPDATE (** *column_name* **)**] ...]
  statement-list

Remarks

The rows deleted or inserted are held in two temporary tables. In the Transact-SQL form of triggers, they can be accessed using the table names deleted, and inserted, as in Adaptive Server Enterprise. In the Watcom-SQL CREATE TRIGGER statement, these rows are accessed using the REFERENCING clause.

Trigger names must be unique in the database.

Transact-SQL triggers are executed AFTER the triggering statement.

Permissions

Must have RESOURCE authority and have ALTER permissions on the table, or must have DBA authority.

CREATE TRIGGER locks all the rows on the table, and requires exclusive use of the table.

Side effects

Automatic commit.

See also

- "CREATE TRIGGER statement" on page 538

Standards and compatibility

- **SQL/2003** Vendor extension.

CREATE VARIABLE statement

Use this statement to create a SQL variable.

Syntax

CREATE VARIABLE *identifier data-type*

Remarks

The CREATE VARIABLE statement creates a new variable of the specified data type. The variable contains the NULL value until it is assigned a different value by the SET statement.

A variable can be used in a SQL expression anywhere a column name is allowed. Name resolution is performed as follows:

1. Match any aliases specified in the query's SELECT list.
2. Match column names for any referenced tables.
3. Assume the name is a variable.

Variables belong to the current connection, and disappear when you disconnect from the database or when you use the DROP VARIABLE statement. Variables are not visible to other connections. Variables are not affected by COMMIT or ROLLBACK statements.

Variables are useful for creating large text or binary objects for INSERT or UPDATE statements from embedded SQL programs.

Local variables in procedures and triggers are declared within a compound statement (see "Using compound statements" [*SQL Anywhere Server - SQL Usage*]).

Permissions

None.

Side effects

None.

See also

- "BEGIN statement" on page 413
- "SQL data types" on page 77
- "DROP VARIABLE statement" on page 592
- "SET statement" on page 735
- "VAREXISTS function [Miscellaneous]" on page 339

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

This example creates a variable called first_name, of data type VARCHAR(50).

```
CREATE VARIABLE first_name VARCHAR(50);
```

This example creates a variable called 'birthday', of data type DATE.

```
CREATE VARIABLE birthday DATE;
```

CREATE USER statement

Use this statement to create a user.

Syntax

CREATE USER *user-name* [**IDENTIFIED BY** *password*]
[**LOGIN POLICY** *policy-name*]
[**FORCE PASSWORD CHANGE** { **ON** | **OFF** }]

Parameters

- **user-name** The name of the user you are creating.
- **IDENTIFIED BY clause** The password of the user you are creating. A user without a password cannot connect to the database.
- **policy-name** The name of the login policy to assign the user. If no login policy is specified, the DEFAULT login policy is applied.
- **FORCE PASSWORD CHANGE clause** Controls whether the user must specify a new password when they log in. This setting overrides the password_expiry_on_next_login option setting in their policy.

Remarks

You do not have to specify a password for the user. A user without a password cannot connect to the database. This is useful if you are creating a group and do not want anyone to connect to the database using the group user ID. A user ID must be a valid identifier.

User IDs and passwords cannot:

- begin with white space, single quotes, or double quotes
- end with white space
- contain semicolons

A password can be either a valid identifier, or a string (maximum 255 bytes) placed in single quotes. Passwords are case sensitive. It is recommended that the password be composed of 7-bit ASCII characters, as other characters may not work correctly if the database server cannot convert them from the client's character set to UTF-8.

Permissions

Must have DBA authority.

Side effects

None.

See also

- "CREATE LOGIN POLICY statement" on page 475
- "ALTER LOGIN POLICY statement" on page 372
- "ALTER USER statement" on page 402
- "COMMENT statement" on page 425
- "DROP LOGIN POLICY statement" on page 577
- "DROP USER statement" on page 591
- "Managing login policies overview" [*SQL Anywhere Server - Database Administration*]
- "Creating a user and assigning a login policy" [*SQL Anywhere Server - Database Administration*]
- "GRANT statement" on page 627
- "min_password_length option [database]" [*SQL Anywhere Server - Database Administration*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following example creates a user named SQLTester with the password welcome. The SQLTester user is assigned to the Test1 login policy and the password expires on the next login.

```
CREATE USER SQLTester IDENTIFIED BY welcome
LOGIN POLICY Test1
FORCE PASSWORD CHANGE ON;
```

The following example creates a group named MyGroup

```
CREATE USER MyGroup;
GRANT GROUP TO MyGroup;
```

CREATE VIEW statement

Use this statement to create a view on the database. Views are used to give a different perspective on the data, even though it is not stored that way.

Syntax

CREATE [**OR REPLACE**] **VIEW**
[*owner*.]*view-name* [**(** *column-name*, ... **)**]
AS *select-statement*
[**WITH CHECK OPTION**]

Parameters

- **OR REPLACE clause** Specifying OR REPLACE (CREATE OR REPLACE VIEW) creates a new view, or replaces an existing view with the same name. Existing permissions are preserved when you use the OR REPLACE clause.
- **AS clause** The SELECT statement on which the view is based. The SELECT statement must not refer to local temporary tables. Also, the SELECT statement can have an ORDER BY or GROUP BY clause, and can be a UNION. However, in some cases, particularly when combined with the FIRST or TOP clause, using a SELECT with an ORDER BY clause does affect the results of a view definition.
- **WITH CHECK OPTION clause** The WITH CHECK OPTION clause rejects any updates and inserts to the view that do not meet the criteria of the view as defined by its SELECT statement.

Remarks

The CREATE VIEW statement creates a view with the given name. You can create a view owned by another user by specifying the owner. You must have DBA authority to create a view for another user.

A view name can be used in place of a table name in SELECT, DELETE, UPDATE, and INSERT statements. Views, however, do not physically exist in the database as tables. They are derived each time they are used. The view is derived as the result of the SELECT statement specified in the CREATE VIEW statement. Table names used in a view should be qualified by the user ID of the table owner. Otherwise, a different user ID might not be able to find the table or might get the wrong table.

Views can be updated unless the SELECT statement defining the view contains a GROUP BY clause, an aggregate function, or involves a UNION clause. An update to the view causes the underlying table(s) to be updated.

The columns in the view are given the names specified in the *column-name* list. If the column name list is not specified, the view columns are given names from the select list items. All items in the select list must have unique names. To use the names from the select list items, each item must be a simple column name or have an alias-name specified. See "SELECT statement" on page 728.

Typically, a view references tables and views (and their respective attributes) that are defined in the catalog. However, a view can also reference SQL variables. In this case, when a query that references the view is executed, the value of the SQL variable is used. Views that reference SQL variables are called **parameterized views** since the variables act as parameters to the execution of the view.

Parameterized views offer an alternative to embedding the body of an equivalent SELECT block in a query as a derived table in the query's FROM clause. Parameterized views can be especially useful for queries embedded in stored procedures where the SQL variables referenced in the view are input parameters to the procedure.

It is not necessary for the SQL variable to exist when the CREATE VIEW statement is executed. However, if the SQL variable is not defined when a query that refers to the view is executed, a **Column Not Found** error is returned.

Permissions

Must have RESOURCE authority and SELECT permission on the tables in the view definition.

Side effects

Automatic commit.

See also

- "CREATE VIEW statement" on page 547
- "CREATE TABLE statement" on page 523
- "CREATE MATERIALIZED VIEW statement" on page 477

Standards and compatibility

- **SQL/2003** Core feature. However, parameterized views are a vendor extension.

Example

The following example creates a view showing information for male employees only. This view has the same column names as the base table.

```
CREATE VIEW MaleEmployees
AS SELECT *
FROM Employees
WHERE Sex = 'M';
```

The following example creates a view showing employees and the departments they belong to.

```
CREATE VIEW EmployeesAndDepartments
  AS SELECT Surname, GivenName, DepartmentName
  FROM Employees JOIN Departments
  ON Employees.DepartmentID = Departments.DepartmentID;
```

The following example replaces the EmployeesAndDepartments view created in the previous example. After replacing the view, the view shows the city, state, and country location for each employee:

```
CREATE OR REPLACE VIEW EmployeesAndDepartments
  AS SELECT Surname, GivenName, City, State, Country
  FROM Employees JOIN Departments
  ON Employees.DepartmentID = Departments.DepartmentID;
```

The following example creates a parameterized view based on the variables var1 and var2, which are neither attributes of the Employees nor Departments tables:

```
CREATE VIEW EmployeesByState
  AS SELECT Surname, GivenName, DepartmentName
  FROM Employees JOIN Departments
  ON Employees.DepartmentID = Departments.DepartmentID
  WHERE Employees.State = var1 and Employees.Status = var2;
```

Variables can appear in the view's SELECT statement in any context where a variable is a permitted expression. For example, the following parameterized view utilizes the parameter var1 as the pattern for a LIKE predicate:

```
CREATE VIEW ProductsByDescription
  AS SELECT *
  FROM Products
  WHERE Products.Description LIKE var1;
```

To use this view, the variable var1 must be defined before the query referencing the view is executed. For example, the following could be placed in a procedure, function, or a batch statement:

```
BEGIN
DECLARE var1 CHAR(20);
SET var1 = '%cap%';
```

```
SELECT * FROM ProductsByDescription
END
```

DEALLOCATE statement

This statement has no effect in SQL Anywhere, and is ignored. It is provided for compatibility with Adaptive Server Enterprise and Microsoft SQL Server. Refer to your Adaptive Server Enterprise or Microsoft SQL Server documentation for more information about this statement.

Standards and compatibility

- **SQL/2003** Vendor extension.

DEALLOCATE DESCRIPTOR statement [ESQL]

Use this statement to free memory associated with a SQL descriptor area.

Syntax

DEALLOCATE DESCRIPTOR *descriptor-name*

descriptor-name : *identifier*

Remarks

Frees all memory associated with a descriptor area, including the data items, indicator variables, and the structure itself.

Permissions

None.

Side effects

None.

See also

- "ALLOCATE DESCRIPTOR statement [ESQL]" on page 357
- "The SQL descriptor area (SQLDA)" [*SQL Anywhere Server - Programming*]
- "SET DESCRIPTOR statement [ESQL]" on page 741

Standards and compatibility

- **SQL/2003** Core feature.

Example

For an example, see "ALLOCATE DESCRIPTOR statement [ESQL]" on page 357.

Declaration section [ESQL]

Use this statement to declare host variables in an embedded SQL program. Host variables are used to exchange data with the database.

Syntax

EXEC SQL BEGIN DECLARE SECTION;
C declarations
EXEC SQL END DECLARE SECTION;

Remarks

A declaration section is simply a section of C variable declarations surrounded by the BEGIN DECLARE SECTION and END DECLARE SECTION statements. A declaration section makes the SQL preprocessor aware of C variables that are used as host variables. Not all C declarations are valid inside a declaration section. See "Using host variables" [*SQL Anywhere Server - Programming*].

Permissions

None.

See also

- "BEGIN statement" on page 413

Standards and compatibility

- **SQL/2003** Core feature.

Example

```
EXEC SQL BEGIN DECLARE SECTION;
char *surname, initials[5];
int dept;
EXEC SQL END DECLARE SECTION;
```

DECLARE statement

Use this statement to declare a SQL variable within a compound statement (BEGIN ... END).

Syntax

DECLARE *variable-name data-type*

Remarks

Variables used in the body of a procedure, trigger, or batch can be declared using the DECLARE statement. The variable persists for the duration of the compound statement in which it is declared.

The body of a Watcom-SQL procedure or trigger is a compound statement, and variables must be declared with other declarations, such as a cursor declaration (DECLARE CURSOR), immediately following the BEGIN keyword. In a Transact-SQL procedure or trigger, there is no such restriction.

See also

- “DECLARE CURSOR statement [ESQL] [SP]” on page 552
- “DECLARE CURSOR statement [T-SQL]” on page 556

Standards and compatibility

- **SQL/2003** Persistent Stored Module feature.

Example

The following batch illustrates the use of the DECLARE statement and prints a message on the database server messages window:

```
BEGIN
  DECLARE varname CHAR(61);
  SET varname = 'Test name';
  MESSAGE varname;
END
```

DECLARE CURSOR statement [ESQL] [SP]

Use this statement to declare a cursor. Cursors are the primary means for manipulating the results of queries.

Syntax 1 [ESQL]

DECLARE *cursor-name*
[**UNIQUE**]
[**NO SCROLL**
 | **DYNAMIC SCROLL**
 | **SCROLL**
 | **INSENSITIVE**
 | **SENSITIVE**
]
CURSOR FOR
{ *select-statement*
| *statement-name* [**FOR** { **UPDATE** [*cursor-concurrency*] | **READ ONLY** }]
| *call-statement* }

Syntax 2 [SP]

DECLARE *cursor-name*
[**NO SCROLL**
 | **DYNAMIC SCROLL**
 | **SCROLL**
 | **INSENSITIVE**
 | **SENSITIVE**
]
CURSOR
{ **FOR** *select-statement* [**FOR** { **UPDATE** [*cursor-concurrency*] | **READ ONLY** }]
| **FOR** *call-statement*
| **USING** *variable-name* }

cursor-name : *identifier*

statement-name : *identifier* | *hostvar* *variable-name* : *identifier*

cursor-concurrency :
BY { **VALUES** | **TIMESTAMP** | **LOCK** }

Parameters

- **UNIQUE clause** When a cursor is declared UNIQUE, the query is forced to return all the columns required to uniquely identify each row. Often this means ensuring that all columns in the primary key or a uniqueness table constraint are returned. Any columns that are required but were not specified in the query are added to the result set.

  A DESCRIBE done on a UNIQUE cursor sets the following additional options in the indicator variables:

  - **DT_KEY_COLUMN** The column is part of the key for the row
  - **DT_HIDDEN_COLUMN** The column was added to the query because it was required to uniquely identify the rows

- **NO SCROLL clause** A cursor declared NO SCROLL is restricted to moving forward through the result set using FETCH NEXT and FETCH RELATIVE 0 seek operations.

  As rows cannot be returned to once the cursor leaves the row, there are no sensitivity restrictions on the cursor. When a NO SCROLL cursor is requested, SQL Anywhere supplies the most efficient kind of cursor, which is an asensitive cursor. See "Asensitive cursors" [*SQL Anywhere Server - Programming*].

- **DYNAMIC SCROLL clause** DYNAMIC SCROLL is the default cursor type. DYNAMIC SCROLL cursors can use all formats of the FETCH statement.

  When a DYNAMIC SCROLL cursor is requested, SQL Anywhere supplies an asensitive cursor. When using cursors there is always a trade-off between efficiency and consistency. Asensitive cursors provide efficient performance at the expense of consistency. See "Asensitive cursors" [*SQL Anywhere Server - Programming*].

- **SCROLL clause** A cursor declared SCROLL can use all formats of the FETCH statement. When a SCROLL cursor is requested, SQL Anywhere supplies a value-sensitive cursor. See "Value-sensitive cursors" [*SQL Anywhere Server - Programming*].

  SQL Anywhere must execute value-sensitive cursors in such a way that result set membership is guaranteed. DYNAMIC SCROLL cursors are more efficient and should be used unless the consistent behavior of SCROLL cursors is required.

- **INSENSITIVE clause** A cursor declared INSENSITIVE has its membership fixed when it is opened; a temporary table is created with a copy of all the original rows. FETCHING from an INSENSITIVE cursor does not see the effect of any other INSERT, UPDATE, or DELETE statement, or any other PUT, UPDATE WHERE CURRENT, DELETE WHERE CURRENT operations on a different cursor. It does see the effect of PUT, UPDATE WHERE CURRENT, DELETE WHERE CURRENT operations on the same cursor. See "Insensitive cursors" [*SQL Anywhere Server - Programming*].

- **SENSITIVE clause** A cursor declared SENSITIVE is sensitive to changes to membership or values of the result set. See "Sensitive cursors" [*SQL Anywhere Server - Programming*].

- **FOR statement-name** Statements are named using the PREPARE statement. Cursors can be declared only for a prepared SELECT or CALL.

- **FOR UPDATE | READ ONLY** A cursor declared FOR READ ONLY cannot be used in an UPDATE (positioned) or a DELETE (positioned) operation. FOR UPDATE is the default.

  Cursors default to FOR UPDATE for single-table queries without an ORDER BY clause, or if the ansi_update_constraints option is set to Off. When the ansi_update_constraints option is set to Cursors or Strict, then cursors over a query containing an ORDER BY clause default to READ ONLY. However, you can explicitly mark cursors as updatable using the FOR UPDATE clause. Because it is expensive to allow updates over cursors with an ORDER BY clause or a join, cursors over a query containing a join of two or more tables are READ ONLY and cannot be made updatable.

  In response to any request for a cursor that specifies FOR UPDATE, SQL Anywhere provides either a value-sensitive cursor or a sensitive cursor. Insensitive and asensitive cursors are not updatable.

- **USING variable-name** For use within stored procedures only. The variable is a string containing a SELECT statement for the cursor. The variable must be available when the DECLARE is processed, and so must be one of the following:

  A parameter to the procedure. For example,

  ```
  CREATE FUNCTION GetRowCount( IN qry LONG VARCHAR)
  RETURNS INT
  BEGIN
    DECLARE crsr CURSOR USING qry;
    DECLARE rowcnt INT;

    SET rowcnt = 0;
    OPEN crsr;
    lp: LOOP
      fetch crsr;
      IF SQLCODE <> 0 THEN LEAVE lp END IF;
      SET rowcnt = rowcnt + 1;
    END LOOP;
    RETURN rowcnt;
  END;
  ```

  Nested inside another BEGIN... END after the variable has been assigned a value. For example,

  ```
  CREATE PROCEDURE get_table_name(
    IN id_value INT, OUT tabname CHAR(128)
  )
  BEGIN
    DECLARE qry LONG VARCHAR;

    SET qry = 'SELECT table_name FROM SYS.SYSTAB ' ||
              'WHERE table_id=' || string(id_value);
    BEGIN
      DECLARE crsr CURSOR USING qry;
      OPEN crsr;
      FETCH crsr INTO tabname;
      CLOSE crsr;
    END
  END;
  ```

- **BY VALUES | TIMESTAMP | LOCK clauses** In embedded SQL, a concurrency specification can be set by including syntax within the SELECT statement itself, or in the cursor declaration. Pessimistic or optimistic concurrency can be chosen at the cursor level either through options with DECLARE CURSOR or FOR statements, or though the concurrency setting API in specific programming interfaces.

If a statement is updatable and the cursor does not specify a particular concurrency control mechanism, the statement's specification is used. The syntax is as follows:

- **FOR UPDATE BY LOCK clause** The database server acquires intent row locks on FETCHed rows of the result set. These are long-term locks that are held until transaction COMMIT or ROLLBACK.
- **FOR UPDATE BY { VALUES | TIMESTAMP }** The database server utilizes a keyset-driven cursor to enable the application to be informed when rows have been modified or deleted as the result set is scrolled.

Remarks

The DECLARE CURSOR statement declares a cursor with the specified name for a SELECT statement or a CALL statement. In a Watcom-SQL procedure, trigger, or batch, a DECLARE CURSOR statement must appear with other declarations, immediately following the BEGIN keyword. Also, cursor names must be unique.

If a cursor is declared inside a compound statement, it exists only for the duration of that compound statement (whether it is declared in a Watcom-SQL or Transact-SQL compound statement).

Permissions

None.

Side effects

None.

See also

- "PREPARE statement [ESQL]" on page 694
- "OPEN statement [ESQL] [SP]" on page 683
- "EXPLAIN statement [ESQL]" on page 603
- "SELECT statement" on page 728
- "CALL statement" on page 419
- "FOR statement" on page 608

Standards and compatibility

- **SQL/2003** Core feature.

Example

The following example illustrates how to declare a scroll cursor in embedded SQL:

```
EXEC SQL DECLARE cur_employee SCROLL CURSOR
FOR SELECT * FROM Employees;
```

The following example illustrates how to declare a cursor for a prepared statement in embedded SQL:

```
EXEC SQL PREPARE employee_statement
FROM 'SELECT Surname FROM Employees';
EXEC SQL DECLARE cur_employee CURSOR
FOR employee_statement;
```

The following example illustrates the use of cursors in a stored procedure:

```
BEGIN
  DECLARE cur_employee CURSOR FOR
      SELECT Surname
      FROM Employees;
  DECLARE name CHAR(40);
  OPEN cur_employee;
  lp: LOOP
    FETCH NEXT cur_employee INTO name;
    IF SQLCODE <> 0 THEN LEAVE lp END IF;
    ...
  END LOOP;
  CLOSE cur_employee;
END
```

DECLARE CURSOR statement [T-SQL]

Use this statement to declare a cursor in a manner compatible with Adaptive Server Enterprise.

Syntax

DECLARE *cursor-name*
 CURSOR FOR *select-statement*
 [**FOR** { **READ ONLY** | **UPDATE** }]

cursor-name : *identifier*

select-statement : *string*

Remarks

DECLARE CURSOR statements in Transact-SQL procedures are treated as executable statements and can appear anywhere in a procedure. The cursor declaration takes effect when the statement is executed and remains in effect until a DEALLOCATE CURSOR statement is executed or until the procedure completes.

In SQL Anywhere, if a cursor is declared inside a compound statement, it exists only for the duration of that compound statement (whether it is declared in a Watcom-SQL or Transact-SQL compound statement).

In a Transact-SQL procedure, trigger, or batch, a DECLARE CURSOR statement can appear after other executable statements.

Permissions

None.

Side effects

None.

See also

- “DECLARE CURSOR statement [ESQL] [SP]” on page 552

Standards and compatibility

- **SQL/2003** Core feature. The FOR UPDATE and FOR READ ONLY options are Transact-SQL extensions.

DECLARE LOCAL TEMPORARY TABLE statement

Use this statement to declare a local temporary table.

Syntax

DECLARE LOCAL TEMPORARY TABLE *table-name*
({ *column-definition* [*column-constraint* ...] | *table-constraint* | *pctfree* }, ... **)**
[**ON COMMIT** { **DELETE** | **PRESERVE** } **ROWS**
  | **NOT TRANSACTIONAL**]

pctfree : **PCTFREE** *percent-free-space*

percent-free-space : *integer*

Parameters

For definitions of *column-definition*, *column-constraint*, *table-constraint*, and *pctfree*, see "CREATE TABLE statement" on page 523.

- **ON COMMIT clause** By default, the rows of a temporary table are deleted on a COMMIT. You can use the ON COMMIT clause to preserve rows on a COMMIT.
- **NOT TRANSACTIONAL clause** A table created using this clause is not affected by either COMMIT or ROLLBACK. The NOT TRANSACTIONAL clause provides performance improvements in some circumstances because operations on non-transactional temporary tables do not cause entries to be made in the rollback log. For example, NOT TRANSACTIONAL can be useful if procedures that use the temporary table are called repeatedly with no intervening COMMITs or ROLLBACKs.

Remarks

The DECLARE LOCAL TEMPORARY TABLE statement declares a temporary table.

The rows of a declared temporary table are deleted when the table is explicitly dropped or when the table goes out of scope. You can also explicitly delete rows using TRUNCATE or DELETE.

Declared local temporary tables within compound statements exist within the compound statement. (See "Using compound statements" [*SQL Anywhere Server - SQL Usage*]). Otherwise, the declared local temporary table exists until the end of the connection.

If you want a procedure to create a local temporary table that persists after the procedure completes, use the CREATE LOCAL TEMPORARY TABLE statement instead. (See "CREATE LOCAL TEMPORARY TABLE statement" on page 474).

Permissions

None.

Side effects

None.

See also

- “CREATE TABLE statement” on page 523
- “CREATE LOCAL TEMPORARY TABLE statement” on page 474
- “Using compound statements” [*SQL Anywhere Server - SQL Usage*]

Standards and compatibility

- **SQL/2003** SQL/foundation feature outside core SQL.

Example

The following example illustrates how to declare a temporary table in a stored procedure:

```
BEGIN
  DECLARE LOCAL TEMPORARY TABLE TempTab ( number INT );
  ...
END
```

DELETE statement

Use this statement to delete rows from the database.

Syntax

DELETE [*row-limitation*]
[**FROM**] [*owner*.]*table-expression*
[**FROM** *table-list* [,...]]
[**WHERE** *search-condition*]
[**ORDER BY** { *expression* | *integer* } [**ASC** | **DESC**], ...]
[**OPTION(** *query-hint*, ... **)**]

table-list :
table-name [,...]

table-name :
[*owner*.]*base-table-name* [[**AS**] *correlation-name*]
| [*owner*.]*view-name* [[**AS**] *correlation-name*]
| *derived-table*

derived-table :
(*select-statement* **)**
[**AS**] *correlation-name* [**(** *column-name* [,...] **)**]

row-limitation :
 FIRST | **TOP** *n* [**START AT** *m*]

query-hint :
MATERIALIZED VIEW OPTIMIZATION *option-value*
| **FORCE OPTIMIZATION**
| **FORCE NO OPTIMIZATION**
| *option-name* = *option-value*

table-expression : A full table expression that can include joins. See “FROM clause” on page 613.

option-name : *identifier*

option-value : *hostvar* (indicator allowed), *string*, *identifier*, or *number*

Parameters

- **row-limitation clause** The row limiting clause allows you to return only a subset of the rows that satisfy the WHERE clause. The TOP and START AT values can be a host variable, integer constant, or integer variable. The TOP value must be greater than or equal to 0. The START AT value must be greater than 0. Normally, when specifying these clauses, an ORDER BY clause is specified as well to order the rows in a meaningful manner. See "Explicitly limiting the number of rows returned by a query" [*SQL Anywhere Server - SQL Usage*].

- **FROM clause** The FROM clause indicates the table from which to delete rows. The second FROM clause in the DELETE statement qualifies the rows to be deleted from the specified table based on joins. If the second FROM clause is present, the WHERE clause qualifies the rows of this second FROM clause.

  The FROM *table-expression* clause allows deletions based on joins. *table-expression* can contain arbitrary complex table expressions, such as KEY and NATURAL joins. For a full description of the FROM clause and joins, see "FROM clause" on page 613.

  The following statement illustrates a potential ambiguity in table names in DELETE statements with two FROM clauses that use correlation names:

  ```
  DELETE
  FROM table_1
  FROM table_1 AS alias_1, table_2 AS alias_2
  WHERE ...
  ```

  table table_1 doesn't have a correlation name in the first FROM clause but does in the second FROM clause. In this case, table_1 in the first clause is identified with alias_1 in the second clause—there is only one instance of table_1 in this statement. This is allowed as an exception to the general rule that where a table is identified with a correlation name and without a correlation name in the same statement, two instances of the table are considered.

  However, in the following example, there are two instances of table_1 in the second FROM clause. The statement fails with a syntax error because it is not clear which instance of the table_1 from the second FROM clause matches the first instance of table_1 in the first FROM clause.

  ```
  DELETE
  FROM table_1
  FROM table_1 AS alias_1, table_1 AS alias_2
  WHERE ...
  ```

- **WHERE clause** The DELETE statement deletes all the rows that satisfy the conditions in the WHERE clause. If no WHERE clause is specified, all rows from the named table are deleted. If a second FROM clause is present, the WHERE clause qualifies the rows of the second FROM clause.

- **ORDER BY clause** Specifies the sort order for the rows. Normally, the order in which rows are updated does not matter. However, in conjunction with the FIRST or TOP clause the order can be significant.

  You cannot use ordinal column numbers in the ORDER BY clause.

Each item in the ORDER BY list can be labeled as ASC for ascending order (the default) or DESC for descending order.

- **OPTION clause** Use this clause to specify hints for executing the statement. The following hints are supported:

  - MATERIALIZED VIEW OPTIMIZATION *option-value*
  - FORCE OPTIMIZATION
  - FORCE NO OPTIMIZATION
  - *option-name* = *option-value*

  For a description of these options, see the OPTIONS clause of the "SELECT statement" on page 728.

Remarks

Deleting a significant amount of data using the DELETE statement causes an update to column statistics.

The DELETE statement can be used on views, provided the SELECT statement defining the view has only one table in the FROM clause and does not contain a GROUP BY clause, an aggregate function, or involve a UNION clause.

DELETEs can be performed on views if the query specification defining the view is updatable. For more information about identifying views that are inherently *non-updatable*, see "Working with regular views" [*SQL Anywhere Server - SQL Usage*].

Permissions

Must have DELETE permission on the table.

Side effects

None.

See also

- "TRUNCATE statement" on page 768
- "INSERT statement" on page 649
- "INPUT statement [Interactive SQL]" on page 644
- "FROM clause" on page 613
- "Locking during deletes" [*SQL Anywhere Server - SQL Usage*]

Standards and compatibility

- **SQL/2003** Core feature. The use of more than one table in the FROM clause is a vendor extension.

Example

Remove all data prior to 2000 from the FinancialData table.

```
DELETE
FROM FinancialData
WHERE Year < 2000;
```

Remove the first 10 orders from SalesOrderItems table where ship date is older than 2001-01-01 and their region is Central.

```
DELETE TOP 10
FROM SalesOrderItems
FROM SalesOrders
WHERE SalesOrderItems.ID = SalesOrders.ID
  and ShipDate < '2001-01-01' and Region ='Central'
ORDER BY ShipDate ASC;
```

Remove department 600 from the database, executing the statement at isolation level 3.

```
DELETE FROM Departments
WHERE DepartmentID = 600
OPTION( isolation_level = 3 );
```

DELETE (positioned) statement [ESQL] [SP]

Use this statement to delete the data at the current location of a cursor.

Syntax

DELETE [**FROM** *table-spec*] **WHERE CURRENT OF** *cursor-name*

cursor-name : *identifier* | *hostvar*

table-spec : [*owner*.]*correlation-name*

owner : *identifier*

Remarks

This form of the DELETE statement deletes the current row of the specified cursor. The current row is defined to be the last row fetched from the cursor.

The table from which rows are deleted is determined as follows:

- If no FROM clause is included, the cursor must be on a single table only.
- If the cursor is for a joined query (including using a view containing a join), then the FROM clause must be used. Only the current row of the specified table is deleted. The other tables involved in the join are not affected.
- If a FROM clause is included, and no table owner is specified, *table-spec* is first matched against any correlation names.
  - If a correlation name exists, *table-spec* is identified with the correlation name.
  - If a correlation name does not exist, *table-spec* must be unambiguously identifiable as a table name in the cursor.
- If a FROM clause is included, and a table owner is specified, *table-spec* must be unambiguously identifiable as a table name in the cursor.

- The positioned DELETE statement can be used on a cursor open on a view as long as the view is updatable.

Permissions

Must have DELETE permission on tables used in the cursor.

Side effects

None.

See also

- "UPDATE statement" on page 777
- "UPDATE (positioned) statement [ESQL] [SP]" on page 782
- "INSERT statement" on page 649
- "PUT statement [ESQL]" on page 697

Standards and compatibility

- **SQL/2003** Core feature. The range of cursors that can be updated may contain vendor extensions if the ansi_update_constraints option is set to Off.

Example

The following statement removes the current row from the database.

```
DELETE
WHERE CURRENT OF cur_employee;
```

DESCRIBE statement [ESQL]

Use this statement to get information about the host variables required to store data retrieved from the database, or host variables required to pass data to the database.

Syntax

DESCRIBE
[**USER TYPES**]
[**ALL** | **BIND VARIABLES FOR** | **INPUT** | **OUTPUT**
| **SELECT LIST FOR**]
[**LONG NAMES** [*long-name-spec*] | **WITH VARIABLE RESULT**]
[**FOR**] { *statement-name* | **CURSOR** *cursor-name* }
INTO *sqlda-name*

long-name-spec :
OWNER.TABLE.COLUMN
| **TABLE.COLUMN**
| **COLUMN**

statement-name : *identifier* or *hostvar*

cursor-name : declared cursor

sqlda-name : *identifier*

Parameters

- **USER TYPES clause** A DESCRIBE statement with the USER TYPES clause returns information about domains of a column. Typically, such a DESCRIBE is done when a previous DESCRIBE returns an indicator of DT_HAS_USERTYPE_INFO.

  The information returned is the same as for a DESCRIBE without the USER TYPES keywords, except that the sqlname field holds the name of the domain, instead of the name of the column.

  If the DESCRIBE uses the LONG NAMES clause, the sqldata field holds this information.

- **ALL clause** DESCRIBE ALL allows you to describe INPUT and OUTPUT with one request to the database server. This has a performance benefit. The OUTPUT information is filled in the SQLDA first, followed by the INPUT information. The sqld field contains the total number of INPUT and OUTPUT variables. The DT_DESCRIBE_INPUT bit in the indicator variable is set for INPUT variables and clear for OUTPUT variables.

- **INPUT clause** A bind variable is a value supplied by the application when the database executes the statements. Bind variables can be considered parameters to the statement. DESCRIBE INPUT fills in the name fields in the SQLDA with the bind variable names. DESCRIBE INPUT also puts the number of bind variables in the sqlda field of the SQLDA.

  DESCRIBE uses the indicator variables in the SQLDA to provide additional information. DT_PROCEDURE_IN and DT_PROCEDURE_OUT are bits that are set in the indicator variable when a CALL statement is described. DT_PROCEDURE_IN indicates an IN or INOUT parameter and DT_PROCEDURE_OUT indicates an INOUT or OUT parameter. Procedure RESULT columns will have both bits clear. After a describe OUTPUT, these bits can be used to distinguish between statements that have result sets (need to use OPEN, FETCH, RESUME, CLOSE) and statements that do not (need to use EXECUTE). DESCRIBE INPUT only sets DT_PROCEDURE_IN and DT_PROCEDURE_OUT appropriately when a bind variable is an argument to a CALL statement; bind variables within an expression that is an argument in a CALL statement will not set the bits.

- **OUTPUT clause** The DESCRIBE OUTPUT statement fills in the data type and length for each select list item in the SQLDA. The name field is also filled in with a name for the select list item. If an alias is specified for a select list item, the name will be that alias. Otherwise, the name is derived from the select list item: if the item is a simple column name, it is used; otherwise, a substring of the expression is used. DESCRIBE will also put the number of select list items in the sqld field of the SQLDA.

  If the statement being described is a UNION of two or more SELECT statements, the column names returned for DESCRIBE OUTPUT are the same column names which would be returned for the first SELECT statement.

  If you describe a CALL statement, the DESCRIBE OUTPUT statement fills in the data type, length, and name in the SQLDA for each INOUT or OUT parameter in the procedure. DESCRIBE OUTPUT also puts the number of INOUT or OUT parameters in the sqld field of the SQLDA.

  If you describe a CALL statement with a result set, the DESCRIBE OUTPUT statement fills in the data type, length, and name in the SQLDA for each RESULT column in the procedure definition. DESCRIBE OUTPUT will also put the number of result columns in the sqld field of the SQLDA.

- **LONG NAMES clause** The LONG NAMES clause is provided to retrieve column names for a statement or cursor. Without this clause, there is a 29-character limit on the length of column names; with the clause, names of an arbitrary length are supported.

  If LONG NAMES is used, the long names are placed into the SQLDATA field of the SQLDA, as if you were fetching from a cursor. None of the other fields (SQLLEN, SQLTYPE, and so on) are filled in. The SQLDA must be set up like a FETCH SQLDA: it must contain one entry for each column, and the entry must be a string type. If there is an indicator variable, truncation is indicated in the usual fashion.

  The default specification for the long names is **TABLE.COLUMN**.

- **WITH VARIABLE RESULT clause** This clause is used to describe procedures that can have more than one result set, with different numbers or types of columns.

  If WITH VARIABLE RESULT is used, the database server sets the SQLCOUNT value after the DESCRIBE statement to one of the following values:

  - **0** The result set may change. The procedure call should be described again following each OPEN statement.

  - **1** The result set is fixed. No re-describing is required.

    For more information about the use of the SQLDA structure, see "The SQL descriptor area (SQLDA)" [*SQL Anywhere Server - Programming*].

Remarks

The DESCRIBE statement sets up the named SQLDA to describe either the OUTPUT (equivalently SELECT LIST) or the INPUT (BIND VARIABLES) for the named statement.

In the INPUT case, DESCRIBE BIND VARIABLES does not set up the data types in the SQLDA: this needs to be done by the application. The ALL keyword allows you to describe INPUT and OUTPUT in one SQLDA.

If you specify a statement name, the statement must have been previously prepared using the PREPARE statement with the same statement name and the SQLDA must have been previously allocated (see "ALLOCATE DESCRIPTOR statement [ESQL]" on page 357).

If you specify a cursor name, the cursor must have been previously declared and opened. The default action is to describe the OUTPUT. Only SELECT statements and CALL statements have OUTPUT. A DESCRIBE OUTPUT on any other statement, or on a cursor that is not a dynamic cursor, indicates no output by setting the sqld field of the SQLDA to zero.

In embedded SQL, NCHAR, NVARCHAR and LONG NVARCHAR are described as DT_FIXCHAR, DT_VARCHAR, and DT_LONGVARCHAR, respectively, by default. If the db_change_nchar_charset function has been called, these data types are described as DT_NFIXCHAR, DT_NVARCHAR and DT_LONGNVARCHAR, respectively. See "db_change_nchar_charset function" [*SQL Anywhere Server - Programming*].

For more information about how NCHAR data types are described, see the documentation for the data type: "NCHAR data type" on page 80, "NVARCHAR data type" on page 82, and "LONG NVARCHAR data type" on page 79.

Permissions

None.

Side effects

None.

See also

- "ALLOCATE DESCRIPTOR statement [ESQL]" on page 357
- "DECLARE CURSOR statement [ESQL] [SP]" on page 552
- "OPEN statement [ESQL] [SP]" on page 683
- "PREPARE statement [ESQL]" on page 694

Standards and compatibility

- **SQL/2003** Core feature. Some clauses are vendor extensions.

Example

The following example shows how to use the DESCRIBE statement:

```
sqlda = alloc_sqlda( 3 );
EXEC SQL DESCRIBE OUTPUT
  FOR employee_statement
  INTO sqlda;
if( sqlda->sqld  >  sqlda->sqln ) {
  actual_size = sqlda->sqld;
  free_sqlda( sqlda );
  sqlda = alloc_sqlda( actual_size );
  EXEC SQL DESCRIBE OUTPUT
    FOR employee_statement
    INTO sqlda;
}
```

DESCRIBE statement [Interactive SQL]

The DESCRIBE statement returns information about a given database object.

Syntax 1 - Describing database objects

DESCRIBE [[**INDEX FOR**] **TABLE** | **PROCEDURE**] [*owner.*]*object-name*

object-name: *table*, *view*, *materialized view*, *procedure*, or *function*

Syntax 2 - Describing the current connection

DESCRIBE CONNECTION

Parameters

- **INDEX FOR clause** Indicates that you want to see the indexes for the specified *object-name*.
- **TABLE clause** Indicates that *object-name* to be described is a table or a view.
- **PROCEDURE clause** Indicates that *object-name* is a procedure or a function.

Remarks

Use DESCRIBE TABLE to list all the columns in the specified table or view. The DESCRIBE TABLE statement returns one row per table column, containing:

- **Column** The name of the column.
- **Type** The type of data in the column.
- **Nullable** Whether nulls are allowed (1=yes, 0=no).
- **Primary Key** Whether the column is in the primary key (1=yes, 0=no).

Use DESCRIBE INDEX FOR TABLE to list all the indexes for the specified table. The DESCRIBE TABLE statement returns one row per index, containing:

- **Index Name** The name of the index.
- **Columns** The columns in the index.
- **Unique** Whether the index is unique (1=yes, 0=no).
- **Type** The type of index. Possible values are: Clustered, Statistic, Hashed, and Other.

Use DESCRIBE PROCEDURE to list all the parameters used by the specified procedure or function. The DESCRIBE PROCEDURE statement returns one row for each parameter, containing:

- **Parameter** The name of the parameter.
- **Type** The data type of the parameter.
- **In/Out** Information about what is passed to, or returned from, the parameter. Possible values are:
  - **In** The parameter is passed to the procedure, but is not modified.
  - **Out** The procedure ignores the parameter's initial value and sets its value when the procedure returns.
  - **In/Out** The parameter is passed to the procedure and the procedure sets the parameter's value when the procedure returns.
  - **Result** The parameter returns a result set.
  - **Return** The parameter returns a declared return value.

If you do not specify either TABLE or PROCEDURE (for example, DESCRIBE *object-name*), Interactive SQL assumes the object is a table. However, if no such table exists, Interactive SQL attempts to describe the object as either a procedure or a function.

Use Syntax 2 to list information about the database or database server that Interactive SQL is connected to. The following properties are returned:

- **Database Product** The name and version number of the database product Interactive SQL is connected to (for example, SQL Anywhere 11.0.0.83).
- **Host Name** The network name of the computer the database server is running on.

- **Host TCP/IP Address** The IP address of the computer the database server is running on.
- **Host Operating System** The name and version number of the operating system used by the computer the database server is running on.
- **Server Name** The name of the database server.
- **Server TCP/IP Port** The port number used by the database server for the current connection.
- **Database Name** The name of the database that Interactive SQL is connected to.
- **Database Character Set** The character set used for CHAR columns in the database.
- **Connection String** The connection string that was used to connect to the database or database server. Three asterisks replace passwords.

Properties that do not apply to the current connection are omitted. For example, if you connect to a database server using shared memory, then the TCP/IP port is omitted.

Permissions

None

Side effects

None

See also

- "Using Interactive SQL" [*SQL Anywhere Server - Database Administration*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Examples

Describe the columns in the Departments table:

```
DESCRIBE TABLE Departments;
```

Here is an example of the result set for this statement:

Column	**Type**	**Nullable**	**Primary key**
DepartmentID	integer	0	1
DepartmentName	char(40)	0	0
DepartmentHeadID	integer	0	0

List the indexes for the Customers table:

```
DESCRIBE INDEX FOR TABLE Customers;
```

Here is an example of the results for this statement:

Index Name	Columns	Unique	Type
IX_customer_name	Surname,GivenName	0	Clustered

DETACH TRACING statement

Use this statement to end a diagnostic tracing session.

Syntax

DETACH TRACING { **WITH** | **WITHOUT** } **SAVE**

Parameters

- **WITH SAVE clause** Specify WITH SAVE to save data any unsaved diagnostic data in the diagnostic tables.
- **WITHOUT SAVE clause** Specify WITHOUT SAVE if you do not want to save any unsaved tracing data.

Remarks

Issue this statement from the database being profiled to stop sending diagnostic information to the diagnostic tables. If you specify the WITHOUT SAVE clause, you can still save the data later—assuming the tracing database is still running and another tracing session has not been started—by using the sa_save_trace_data system procedure. See "sa_save_trace_data system procedure" on page 932.

To see the current tracing levels set for a database, look in the sa_diagnostic_tracing_level table. See "sa_diagnostic_tracing_level table" on page 824.

Note
Tracing information is *not* unloaded as part of a database unload or reload operation. If you want to transfer tracing information from one database to another, you must do so manually by copying the contents of the sa_diagnostic_* tables; however, this is not recommended.

Permissions

Must have DBA authority.

Side effects

None.

See also

- "ATTACH TRACING statement" on page 406
- "REFRESH TRACING LEVEL statement" on page 708
- "Advanced application profiling using diagnostic tracing" [*SQL Anywhere Server - SQL Usage*]
- "sa_diagnostic_tracing_level table" on page 824
- "sa_save_trace_data system procedure" on page 932

Standards and compatibility

- **SQL/2003** Vendor extension.

DISCONNECT statement [ESQL] [Interactive SQL]

Use this statement to drop the current connection to a database.

Syntax

DISCONNECT [*connection-name* | **CURRENT** | **ALL**]

connection-name : *identifier*, *string*, or *hostvar*

Remarks

The DISCONNECT statement drops a connection with the database server and releases all resources used by it. If the connection to be dropped was named on the CONNECT statement, the name can be specified. Specifying ALL will drop all the application's connections to all database environments. CURRENT is the default, and will drop the current connection.

Before closing the database connection, Interactive SQL automatically executes a COMMIT statement if the commit_on_exit option is set to On. If this option is set to Off, Interactive SQL performs an implicit ROLLBACK. By default, the commit_on_exit option is set to On.

For information about dropping connections other than the current connection, see "DROP CONNECTION statement" on page 570.

This statement is not supported in procedures, triggers, events, or batches.

Permissions

None.

Side effects

None.

See also

- "CONNECT statement [ESQL] [Interactive SQL]" on page 429
- "SET CONNECTION statement [Interactive SQL] [ESQL]" on page 740
- "Using Interactive SQL" [*SQL Anywhere Server - Database Administration*]

Standards and compatibility

- **SQL/2003** SQL/foundation feature outside core SQL.

Example

The following statement shows how to use DISCONNECT in embedded SQL:

```
EXEC SQL DISCONNECT :conn_name
```

The following statement shows how to use DISCONNECT from Interactive SQL to disconnect all connections:

```
DISCONNECT ALL;
```

DROP CONNECTION statement

Use this statement to drop a user's connection to the database.

Syntax

DROP CONNECTION *connection-id*

Remarks

The DROP CONNECTION statement disconnects a user from the database by dropping the connection to the database.

The *connection-id* parameter is an integer constant. You can obtain the *connection-id* using the sa_conn_info system procedure.

This statement is not supported in procedures, triggers, events, or batches.

Permissions

Must have DBA authority.

Side effects

None.

See also

- "CONNECT statement [ESQL] [Interactive SQL]" on page 429
- "sa_conn_info system procedure" on page 852
- "Using exception handlers in procedures and triggers" [*SQL Anywhere Server - SQL Usage*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following procedure drops a connection identified by its connection number. Note that when executing the DROP CONNECTION statement from within a procedure, you should do so using the EXECUTE IMMEDIATE statement, as shown in this example:

```
CREATE PROCEDURE drop_connection_by_id( IN conn_number INTEGER )
 BEGIN
   EXECUTE IMMEDIATE 'DROP CONNECTION ' || conn_number;
 END;
```

The following statement drops the connection with ID number 4.

```
DROP CONNECTION 4;
```

DROP DATABASE statement

Use this statement to delete all database files associated with a database.

Syntax

DROP DATABASE *database-name* [**KEY** *key*]

Remarks

The DROP DATABASE statement physically deletes all associated database files from disk. If the database file does not exist, or is not in a suitable condition for the database to be started, an error is generated.

DROP DATABASE cannot be used in stored procedures, triggers, events, or batches.

Permissions

Required permissions are set using the database server -gu option. The default setting is to require DBA authority.

The database must not be in use to be dropped.

You must specify a key if you want to drop a strongly encrypted database

Not supported on Windows Mobile.

Side effects

In addition to deleting the database files from disk, any associated transaction log file or transaction log mirror file is deleted.

See also

- "CREATE DATABASE statement" on page 432
- "DatabaseKey connection parameter [DBKEY]" [*SQL Anywhere Server - Database Administration*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

Drop the database *temp.db*, in the *C:\temp* directory:

```
DROP DATABASE 'c:\temp\temp.db';
```

DROP DATATYPE statement

Use this statement to remove a datatype from the database.

Syntax

DROP DATATYPE *datatype-name*

Remarks

Use the IF EXISTS clause if you do not want an error returned when the DROP statement attempts to remove a database object that does not exist.

It is recommended that you use DROP DOMAIN rather than DROP DATATYPE, as DROP DOMAIN is the syntax used in the ANSI/ISO SQL3 draft. You cannot drop system-defined data types (such as MONEY or UNIQUEIDENTIFIERSTR) from a database.

Permissions

Any user who owns the object, or has DBA authority, can execute the DROP DATATYPE statement.

Side effects

Automatic commit. Clears the **Results** tab in the **Results** pane in Interactive SQL.

See also

- "CREATE DOMAIN statement" on page 443
- "ALTER DOMAIN statement" on page 364

Standards and compatibility

- **SQL/2003** Core feature.

Example

Drop MyDatatype from the database. If the datatype does not exist, an error is returned.

```
DROP DATATYPE MyDatatype;
```

DROP DBSPACE statement

Use this statement to remove a dbspace from the database.

Syntax

DROP DBSPACE *dbspace-name*

Remarks

You must drop all tables in the dbspace before dropping the dbspace. You cannot use the DROP DBSPACE statement to drop the pre-defined dbspaces system, temporary, temp, translog, or translogmirror. See "Pre-defined dbspaces" [*SQL Anywhere Server - Database Administration*].

Use the IF EXISTS clause if you do not want an error returned when the DROP DBSPACE statement attempts to remove a database object that does not exist.

DROP DBSPACE is prevented whenever the statement affects an object that is currently being used by another connection.

Permissions

You must own the object, or have DBA authority, and be the only connection to the database.

Side effects

Automatic commit. Clears the **Results** tab in the **Results** pane in Interactive SQL.

See also

- "CREATE DBSPACE statement" on page 439
- "ALTER DBSPACE statement" on page 362
- "Delete a dbspace" [*SQL Anywhere Server - Database Administration*]

Standards and compatibility

- **SQL/2003** Core feature.

Example

Drop MyDBSpace from the database. If the dbspace does not exist, an error is returned.

```
DROP DBSPACE MyDBSpace;
```

DROP DOMAIN statement

Use this statement to remove a domain from the database.

Syntax

DROP DOMAIN *domain-name*

Remarks

Use the IF EXISTS clause if you do not want an error returned when the DROP DOMAIN statement attempts to remove a database object that does not exist.

DROP DOMAIN is prevented if the data type is used in a table column, or in a procedure or function argument. You must change data types on all columns defined using the domain to drop the data type. It is recommended that you use DROP DOMAIN rather than DROP DATATYPE, as DROP DOMAIN is the syntax used in the ANSI/ISO SQL3 draft. You cannot drop system-defined data types (such as MONEY or UNIQUEIDENTIFIERSTR) from a database.

Permissions

Any user who owns the object, or has DBA authority, can execute the DROP DOMAIN statement.

Side effects

Automatic commit. Clears the **Results** tab in the **Results** pane in Interactive SQL.

See also

- "CREATE DOMAIN statement" on page 443
- "ALTER DOMAIN statement" on page 364

Standards and compatibility

- **SQL/2003** Core feature.

Example

Drop the domain MyDomain from the database. If the domain does not exist, an error is returned.

```
DROP DOMAIN MyDomain;
```

DROP EVENT statement

Use this statement to drop an event from the database.

Syntax

DROP EVENT [**IF EXISTS**] [*owner*.]*event-name*

Remarks

Use the IF EXISTS clause if you do not want an error returned when the DROP EVENT statement attempts to remove an event that does not exist.

Permissions

Any user who owns the object, or has DBA authority, can execute the DROP EVENT statement.

Side effects

Automatic commit. Clears the **Results** tab in the **Results** pane in Interactive SQL.

See also

- "CREATE EVENT statement" on page 450
- "ALTER EVENT statement" on page 365
- "TRIGGER EVENT statement" on page 767

Standards and compatibility

- **SQL/2003** Core feature.

Example

Drop MyEvent from the database. If the event does not exist, an error is returned.

```
DROP EVENT MyEvent;
```

DROP EXTERNLOGIN statement

Use this statement to drop an external login from the SQL Anywhere catalogs.

Syntax

DROP EXTERNLOGIN *login-name* **TO** *remote-server*

Parameters

- **DROP clause** Specifies the local user login name

- **TO clause** Specifies the name of the remote server. The local user's alternate login name and password for that server is the external login that is deleted.

Remarks

DROP EXTERNLOGIN deletes an external login from the SQL Anywhere catalogs.

Permissions

Must have DBA authority.

Side effects

Automatic commit.

See also

- "CREATE EXTERNLOGIN statement" on page 457

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

```
DROP EXTERNLOGIN DBA TO sybase1;
```

DROP FUNCTION statement

Use this statement to remove a function from the database.

Syntax

DROP FUNCTION [**IF EXISTS**] [*owner*.]*function-name*

Remarks

Use the IF EXISTS clause if you do not want an error returned when the DROP FUNCTION statement attempts to remove a function that does not exist.

DROP FUNCTION is prevented when the statement affects an object that is currently being used by another connection.

Permissions

Any user who owns the object, or has DBA authority, can execute the DROP FUNCTION statement.

Side effects

Automatic commit. Clears the **Results** tab in the **Results** pane in Interactive SQL.

See also

- "CREATE FUNCTION statement (web services)" on page 467
- "ALTER FUNCTION statement" on page 369

Standards and compatibility

- **SQL/2003** Core feature.

Example

Drop MyFunction from the database. If the function does not exist, an error is returned.

```
DROP FUNCTION MyFunction;
```

DROP INDEX statement

Use this statement to remove an index from the database.

Syntax

DROP INDEX { [[*owner*.]*table-name*.]*index-name* | [[*owner*.]*materialized-view-name*.]*index-name* }

Remarks

Use the IF EXISTS clause if you do not want an error returned when the DROP INDEX statement attempts to remove a database object that does not exist.

DROP INDEX is prevented when the statement affects an object that is currently being used by another connection.

Permissions

A user with REFERENCES permissions on the table can execute DROP INDEX.

The DROP INDEX statement cannot be executed when there are cursors opened with the WITH HOLD clause that use either statement or transaction snapshots. See "Snapshot isolation" [*SQL Anywhere Server - SQL Usage*].

Side effects

Automatic commit. Clears the **Results** tab in the **Results** pane in Interactive SQL. The DROP INDEX statement closes all cursors for the current connection.

If you use the DROP INDEX statement to drop an index on a local temporary table an `Index not found` error is returned. Use the DROP TABLE statement to drop a local temporary table. Indexes on local temporary tables are dropped automatically when the local temporary table goes out of scope.

See also

- "CREATE INDEX statement" on page 470
- "ALTER INDEX statement" on page 370

Standards and compatibility

- **SQL/2003** Core feature.

Example

Drop MyIndex from the database. If the index does not exist, an error is returned.

```
DROP INDEX MyIndex;
```

DROP LOGIN POLICY statement

Use this statement to drop a login policy.

Syntax

DROP LOGIN POLICY *policy-name*

Parameters

- **policy-name** The name of the login policy.

Remarks

The statement fails if you drop a policy that is assigned to a user. You cannot drop the root login policy. Use the ALTER USER statement to change a user's policy assignment. See "ALTER USER statement" on page 402.

Permissions

Must have DBA authority.

Side effects

None.

See also

- "ALTER LOGIN POLICY statement" on page 372
- "ALTER USER statement" on page 402
- "COMMENT statement" on page 425
- "CREATE LOGIN POLICY statement" on page 475
- "CREATE USER statement" on page 546
- "DROP USER statement" on page 591
- "Managing login policies overview" [*SQL Anywhere Server - Database Administration*]
- "Dropping a login policy" [*SQL Anywhere Server - Database Administration*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Examples

The following example creates a login policy, Test11, and then deletes it.

```
CREATE LOGIN POLICY Test11;
DROP LOGIN POLICY Test11;
```

DROP MATERIALIZED VIEW statement

Use this statement to remove a materialized view from the database.

Syntax

DROP MATERIALIZED VIEW [**IF EXISTS**] [*owner*.]*materialized-view-name*

Remarks

All data in the table is automatically deleted as part of the dropping process. All indexes and keys for the materialized view are dropped as well.

Use the IF EXISTS clause if you do not want an error returned when the DROP MATERIALIZED VIEW statement attempts to remove a materialized view that does not exist.

You cannot execute a DROP MATERIALIZED VIEW statement on an object that is currently being used by another connection.

Executing a DROP MATERIALIZED VIEW statement changes the status of all dependent regular views to INVALID. To determine view dependencies before dropping a materialized view, use the sa_dependent_views system procedure. See “sa_dependent_views system procedure” on page 863.

Permissions

Any user who owns the object, or has DBA authority, can execute the DROP MATERIALIZED VIEW statement.

Side effects

Automatic commit. Clears the **Results** tab in the **Results** pane in Interactive SQL. Closes all cursors for the current connection.

When a view is dropped, all procedures and triggers are unloaded from memory, so that any procedure or trigger that references the view reflects the fact that the view does not exist. The unloading and loading of procedures and triggers can affect performance if you are regularly dropping and creating views.

See also

- “CREATE MATERIALIZED VIEW statement” on page 477
- “ALTER MATERIALIZED VIEW statement” on page 373
- “REFRESH MATERIALIZED VIEW statement” on page 703
- “Materialized view statuses and properties” [*SQL Anywhere Server - SQL Usage*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

Drop MyMaterializedView from the database. If the materialized view does not exist, an error is returned.

```
DROP MATERIALIZED VIEW MyMaterializedView;
```

DROP MESSAGE statement

Use this statement to remove a message from the database.

Syntax

DROP MESSAGE *msgnum*

Remarks

None.

Permissions

Any user who owns the object, or has DBA authority, can execute the DROP MESSAGE statement.

Side effects

Automatic commit. Clears the **Results** tab in the **Results** pane in Interactive SQL.

See also

- "MESSAGE statement" on page 680

Standards and compatibility

- **SQL/2003** Core feature.

Example

Drop MyMessage from the database. If the message does not exist, an error is returned.

```
DROP MESSAGE MyMessage;
```

DROP PROCEDURE statement

Use this statement to remove a procedure from the database.

Syntax

DROP PROCEDURE [**IF EXISTS**] [*owner*.]*procedure-name*

Remarks

Use the IF EXISTS clause if you do not want an error returned when the DROP PROCEDURE statement attempts to remove a procedure that does not exist.

You cannot execute a DROP PROCEDURE statement when the statement affects an object that is currently being used by another connection.

Permissions

Any user who owns the object, or has DBA authority, can execute the DROP PROCEDURE statement.

Side effects

Automatic commit. Clears the **Results** tab in the **Results** pane in Interactive SQL.

See also

- "CREATE PROCEDURE statement (web services)" on page 495
- "ALTER PROCEDURE statement" on page 376

Standards and compatibility

- **SQL/2003** Core feature.

Example

Drop MyProcedure from the database. If the procedure does not exist, an error is returned.

```
DROP PROCEDURE MyProcedure;
```

DROP PUBLICATION statement [MobiLink] [SQL Remote]

Use this statement to drop a publication. In MobiLink, a publication identifies synchronized data in a SQL Anywhere remote database. In SQL Remote, publications identify replicated data in both consolidated and remote databases.

Syntax

DROP PUBLICATION [*owner*.]*publication-name*

owner, *publication-name* : *identifier*

Remarks

This statement is applicable only to MobiLink and SQL Remote.

Permissions

Must have DBA authority.

Side effects

Automatic commit. All subscriptions to the publication are dropped.

See also

- "ALTER PUBLICATION statement [MobiLink] [SQL Remote]" on page 377
- "CREATE PUBLICATION statement [MobiLink] [SQL Remote]" on page 500
- SQL Anywhere MobiLink clients: "Publishing data" [*MobiLink - Client Administration*]
- UltraLite MobiLink clients: "UltraLite DROP PUBLICATION statement" [*UltraLite - Database Management and Reference*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement drops the pub_contact publication.

```
DROP PUBLICATION pub_contact;
```

DROP REMOTE MESSAGE TYPE statement [SQL Remote]

Use this statement to delete a message type definition from a database.

Syntax

DROP REMOTE MESSAGE TYPE *message-system*

message-system:
FILE
| **FTP**
| **SMTP**

Remarks

The statement removes a message type from a database.

Permissions

Must have DBA authority. To be able to drop the type, there must be no user granted REMOTE or CONSOLIDATE permissions with this type.

Side effects

Automatic commit.

See also

- "CREATE REMOTE MESSAGE TYPE statement [SQL Remote]" on page 503
- "SQL Remote message systems" [*SQL Remote*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement drops the FILE message type from a database.

```
DROP REMOTE MESSAGE TYPE file;
```

DROP SERVER statement

Use this statement to drop a remote server from the SQL Anywhere catalog.

Syntax

DROP SERVER *server-name*

Remarks

DROP SERVER deletes a remote server from the SQL Anywhere catalogs. You must drop all the proxy tables that have been defined for the remote server before this statement will succeed.

Permissions

Only user DBA can delete a remote server.

Not supported on Windows Mobile.

Side effects

Automatic commit.

See also

- "CREATE SERVER statement" on page 506

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

```
DROP SERVER ase_prod;
```

DROP SERVICE statement

Use this statement to drop a web service.

Syntax

DROP SERVICE *service-name*

Remarks

This statement deletes a web service listed in the ISYSWEBSERVICE system table.

Permissions

Must have DBA authority.

Side effects

None.

See also

- "ALTER SERVICE statement" on page 382
- "CREATE SERVICE statement" on page 509
- "ISYSWEBSERVICE system table" on page 809

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

To drop a web service named tables, execute the following statement:

```
DROP SERVICE tables;
```

DROP STATEMENT statement [ESQL]

Use this statement to free statement resources.

Syntax

DROP STATEMENT [*owner*.]*statement-name*

statement-name :
identifier
| *hostvar*

Remarks

The DROP STATEMENT statement frees resources used by the named prepared statement. These resources are allocated by a successful PREPARE statement, and are normally not freed until the database connection is released.

Permissions

Must have prepared the statement.

Side effects

None.

See also

- "PREPARE statement [ESQL]" on page 694

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following are examples of DROP STATEMENT use:

```
EXEC SQL DROP STATEMENT S1;
EXEC SQL DROP STATEMENT :stmt;
```

DROP STATISTICS statement

Use this statement to erase all column statistics on the specified columns.

Syntax

DROP STATISTICS [**ON**] [*owner*.]*object-name* [**(** *column-list* **)**]

object-name :
table-name
| *materialized-view-name*
| *temp-table-name*

Remarks

The SQL Anywhere optimizer uses column statistics to determine the best strategy for executing each statement. SQL Anywhere automatically gathers and updates these statistics. Column statistics are stored permanently in the database in the ISYSCOLSTAT system table. Column statistics gathered while processing one statement are available when searching for efficient ways to execute subsequent statements.

Occasionally, the column statistics can become inaccurate or relevant statistics may be unavailable. This condition is most likely to arise when few queries have been executed since a large amount of data was added, updated, or deleted.

The DROP STATISTICS statement deletes all internal statistical data from the ISYSCOLSTAT system table for the specified columns. This drastic step leaves the optimizer with no access to essential statistical information. Without these statistics, the optimizer can generate inefficient data access plans, causing poor database performance.

The DROP STATISTICS statement requires an exclusive lock on the table against which it is being performed. This means that execution of the statement cannot proceed until all other connections that refer to the table have either committed or rolled back the referring transactions, or closed any open cursors that refer to the table.

This statement should be used only during problem determination or when reloading data into a database that differs substantially from the original data.

Permissions

Must have DBA authority.

Side effects

Automatic commit.

See also

- “CREATE STATISTICS statement” on page 516
- “Optimizer estimates and column statistics” [*SQL Anywhere Server - SQL Usage*]
- “ISYSCOLSTAT system table” on page 801

Standards and compatibility

- **SQL/2003** Vendor extension.

DROP SUBSCRIPTION statement [SQL Remote]

Use this statement to drop a subscription for a user from a publication.

Syntax

DROP SUBSCRIPTION TO *publication-name* [**(** *subscription-value* **)**]
 FOR *subscriber-id*, ...

subscription-value: *string*

subscriber-id: *string*

Parameters

- **publication-name** The name of the publication to which the user is being subscribed. This can include the owner of the publication.
- **subscription-value** A string that is compared to the subscription expression of the publication. This value is required because a user can have more than one subscription to a publication.
- **subscriber-id** The user ID of the subscriber to the publication.

Remarks

Drops a SQL Remote subscription for a user ID to a publication in the current database. The user ID will no longer receive updates when data in the publication is changed.

In SQL Remote, publications and subscriptions are two-way relationships. If you drop a subscription for a remote user to a publication on a consolidated database, you should also drop the subscription for the consolidated database on the remote database to prevent updates on the remote database being sent to the consolidated database.

Permissions

Must have DBA authority.

Side effects

Automatic commit.

See also

- "CREATE SUBSCRIPTION statement [SQL Remote]" on page 517
- "ISYSSUBSCRIPTION system table" on page 807

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement drops a subscription for the SamS user ID to the pub_contact publication.

```
DROP SUBSCRIPTION TO pub_contact
FOR SamS;
```

DROP SYNCHRONIZATION PROFILE statement [MobiLink]

Use this statement to delete a SQL Anywhere synchronization profile. Synchronization profiles define how a SQL Anywhere database synchronizes with the MobiLink server.

Syntax

DROP SYNCHRONIZATION PROFILE *name*

Parameters

- **name** The name of the synchronization profile to delete.

Remarks

None.

Permissions

Must have DBA authority.

Side effects

Automatic commit.

See also

- "CREATE SYNCHRONIZATION PROFILE statement [MobiLink]" on page 518
- "ALTER SYNCHRONIZATION PROFILE statement [MobiLink]" on page 384

Standards and compatibility

- **SQL/2003** Vendor extension.

DROP SYNCHRONIZATION SUBSCRIPTION statement [MobiLink]

Use this statement to drop a synchronization subscription in a MobiLink remote database.

Syntax

DROP SYNCHRONIZATION SUBSCRIPTION
TO *publication-name*
[**FOR** *ml_username*, ...]

Parameters

- **TO clause** Specify the name of a publication.
- **FOR clause** Specify one more MobiLink users.

  Omitting this clause drops the default settings for the publication.

Permissions

Must have DBA authority. Requires exclusive access to all tables referred to in the publication.

Side Effects

Automatic commit.

See also

- "ALTER SYNCHRONIZATION SUBSCRIPTION statement [MobiLink]" on page 386
- "CREATE SYNCHRONIZATION SUBSCRIPTION statement [MobiLink]" on page 519
- "ISYSSYNC system table" on page 807

Standards and compatibility

- **SQL/2003** Vendor extension.

Examples

The following example drops the subscription between the MobiLink user ml_user1 and the publication called sales_publication:

```
DROP SYNCHRONIZATION SUBSCRIPTION
   TO sales_publication
   FOR "ml_user1";
```

The following example omits the FOR clause, and so drops the default settings for the publication called sales_publication:

```
DROP SYNCHRONIZATION SUBSCRIPTION
   TO sales_publication;
```

DROP SYNCHRONIZATION USER statement [MobiLink]

Use this statement to drop one or more synchronization users from a SQL Anywhere remote database.

Syntax

DROP SYNCHRONIZATION USER *ml_username*, ...

ml_username: *identifier*

Remarks

Drop one or more synchronization users from a MobiLink remote database.

Permissions

Must have DBA authority. Requires exclusive access to all tables referred to in the publication.

Side Effects

All subscriptions associated with the user are also deleted.

See also

- “ALTER SYNCHRONIZATION USER statement [MobiLink]” on page 388
- “CREATE SYNCHRONIZATION USER statement [MobiLink]” on page 521
- “ISYSSYNC system table” on page 807

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

Remove MobiLink user ml_user1 from the database.

```
DROP SYNCHRONIZATION USER ml_user1;
```

DROP TABLE statement

Use this statement to remove a table from the database.

Syntax

DROP TABLE [**IF EXISTS**] [*owner*.]*table-name*

Remarks

When you remove a table, all data in the table is automatically deleted as part of the dropping process. All indexes and keys for the table are dropped as well.

Use the IF EXISTS clause if you do not want an error returned when the DROP TABLE statement attempts to remove a table that does not exist.

You cannot execute a DROP TABLE statement when the statement affects a table that is currently being used by another connection. Execution of a DROP TABLE statement is also prevented if there is a materialized view dependent on the table.

When you execute a DROP TABLE statement, the status of all dependent regular views change to INVALID. To determine view dependencies before dropping a table, use the sa_dependent_views system procedure. See “sa_dependent_views system procedure” on page 863.

Permissions

Any user who owns the object, or has DBA authority, can execute the DROP TABLE statement.

Global temporary tables cannot be dropped unless all users that have referenced the temporary table have disconnected.

Side effects

Automatic commit. Clears the **Results** tab in the **Results** pane in Interactive SQL. Executing a DROP TABLE statement closes all cursors for the current connection.

You can use the DROP TABLE statement to drop a local temporary table.

See also

- “CREATE TABLE statement” on page 523
- “ALTER TABLE statement” on page 389

Standards and compatibility

- **SQL/2003** Core feature.

Example

Drop MyTable from the database. If the table does not exist, an error is returned.

```
DROP TABLE MyTable;
```

Drop MyTable from the database if it exists. If the table does not exist, an error is not returned.

```
DROP TABLE IF EXISTS MyTable;
```

DROP TEXT CONFIGURATION statement

Drops a text configuration object.

Syntax

DROP TEXT CONFIGURATION [*owner*.]*text-config-name*

Remarks

Attempting to drop a text configuration object with dependent text indexes results in an error. You must drop the dependent text indexes before dropping the text configuration object.

Text configuration objects are stored in the ISYSTEXTCONFIG system table.

To determine the text indexes that refer to the text configuration object, see “View text indexes in the database” [*SQL Anywhere Server - SQL Usage*].

Permissions

Must be the owner of the text configuration object or have DBA authority.

Side effects

Automatic commit

See also

- “DROP TEXT INDEX statement” on page 590
- “Full text searching” [*SQL Anywhere Server - SQL Usage*]
- “Text configuration objects” [*SQL Anywhere Server - SQL Usage*]
- “SYSTEXTCONFIG system view” on page 1034
- “CREATE TEXT CONFIGURATION statement” on page 535
- “ALTER TEXT CONFIGURATION statement” on page 397

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statements create and drop the mytextconfig text configuration object:

```
CREATE TEXT CONFIGURATION mytextconfig FROM default_char;
DROP TEXT CONFIGURATION mytextconfig;
```

DROP TEXT INDEX statement

Removes a text index from the database.

Syntax

DROP TEXT INDEX *text-index-name*
ON [*owner.*]*table-name*

Parameters

- **ON clause** Use this clause to specify the table on which the text index was built.

Remarks

You must drop dependent text indexes before you can drop a text configuration object.

Permissions

Must be the owner of the underlying table, or have DBA authority, or have REFERENCES permission.

This statement cannot be executed when there are cursors opened with the WITH HOLD clause that use either statement or transaction snapshots. See "Snapshot isolation" [*SQL Anywhere Server - SQL Usage*].

Side effects

Automatic commit

See also

- "Full text searching" [*SQL Anywhere Server - SQL Usage*]
- "Text indexes" [*SQL Anywhere Server - SQL Usage*]
- "SYSTEXTCONFIG system view" on page 1034
- "CREATE TEXT INDEX statement" on page 536
- "ALTER TEXT INDEX statement" on page 399
- "DROP TEXT INDEX statement" on page 590
- "REFRESH TEXT INDEX statement" on page 706
- "TRUNCATE TEXT INDEX statement" on page 770

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statements create and drop the TextIdx text index:

```
CREATE TEXT INDEX TextIdx ON MarketingInformation ( Description )
DROP TEXT INDEX TextIdx ON MarketingInformation;
```

DROP TRIGGER statement

Use this statement to remove a trigger from the database.

Syntax

DROP TRIGGER [**IF EXISTS**] [*owner*.] [*table-name*.]*trigger-name*

Remarks

Use the IF EXISTS clause if you do not want an error returned when the DROP statement attempts to remove a database object that does not exist.

Permissions

A user with ALTER permissions on the table can execute DROP TRIGGER.

Side effects

Automatic commit. Clears the **Results** tab in the **Results** pane in Interactive SQL.

See also

- "CREATE TRIGGER statement" on page 538
- "ALTER TRIGGER statement" on page 401
- "ROLLBACK TRIGGER statement" on page 726

Standards and compatibility

- **SQL/2003** Core feature.

Example

Drop MyTrigger from the database. If the trigger does not exist, an error is returned.

```
DROP TRIGGER MyTrigger;
```

DROP USER statement

Use this statement to drop a user.

Syntax

DROP USER *user-name*

Parameters

- **user-name** The name of the user you are dropping.

Permissions

Must have DBA authority.

Remarks

None.

Side effects

None.

See also

- "ALTER LOGIN POLICY statement" on page 372
- "ALTER USER statement" on page 402
- "COMMENT statement" on page 425
- "CREATE LOGIN POLICY statement" on page 475
- "CREATE USER statement" on page 546
- "DROP LOGIN POLICY statement" on page 577
- "Managing login policies overview" [*SQL Anywhere Server - Database Administration*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following example drops the user SQLTester from a database.

```
DROP USER SQLTester;
```

DROP VARIABLE statement

Use this statement to eliminate a SQL variable.

Syntax

DROP VARIABLE [**IF EXISTS**] *identifier*

Remarks

The DROP VARIABLE statement eliminates a SQL variable that was previously created using the CREATE VARIABLE statement. Variables are automatically eliminated when the database connection is released. Variables are often used for large objects, so eliminating them after use or setting them to NULL can free up significant resources (primarily disk space).

Use the IF EXISTS clause if you do not want an error returned when the DROP statement attempts to remove a database object that does not exist.

Permissions

None.

Side effects

None.

See also

- “CREATE VARIABLE statement” on page 545
- “SET statement” on page 735

Standards and compatibility

- **SQL/2003** Vendor extension.

DROP VIEW statement

Use this statement to remove a view from the database.

Syntax

DROP VIEW [**IF EXISTS**] [*owner*.]*view-name*

Remarks

Use the IF EXISTS clause if you do not want an error returned when the DROP VIEW statement attempts to remove a view that does not exist.

When you execute the DROP VIEW statement, the status of all dependent regular views change to INVALID. To determine view dependencies before dropping a view, use the sa_dependent_views system procedure. See “sa_dependent_views system procedure” on page 863.

Permissions

Any user who owns the object, or has DBA authority, can execute the DROP VIEW statement.

Side effects

Automatic commit. Clears the **Results** tab in the **Results** pane in Interactive SQL. Executing a DROP VIEW statement closes all cursors for the current connection.

When a view is dropped, all procedures and triggers are unloaded from memory, so that any procedure or trigger that references the view reflects the fact that the view does not exist. The unloading and loading of procedures and triggers can affect performance if you are regularly dropping and creating views.

See also

- “CREATE VIEW statement” on page 547
- “ALTER VIEW statement” on page 403

Standards and compatibility

- **SQL/2003** Core feature.

Example

Drop MyView from the database. If the view does not exist, an error is returned.

```
DROP VIEW MyView;
```

SQL statements (E-O)

The following sections define the syntax information for SQL statements E-O.

See also

- "SQL statements (A-D)" on page 357
- "SQL statements (P-Z)" on page 692
- "Common elements in SQL syntax" on page 354
- "Syntax conventions" on page 355
- "Statement applicability indicators" on page 356

EXCEPT clause

Returns rows that exist in the result set for the query block preceding EXCEPT that do not exist in the result set of the query block following EXCEPT.

Syntax

```
[ WITH temporary-views ] main-query-block
  EXCEPT [ ALL | DISTINCT ] except-query-block
[ ORDER BY  [ integer | select-list-expression-name ] [ ASC | DESC ], ... ]
[ FOR XML xml-mode ]
[ OPTION( query-hint, ... ) ]

query-hint :
MATERIALIZED VIEW OPTIMIZATION option-value
| FORCE OPTIMIZATION
| option-name = option-value
```

main-query-block : A query block. See "Common elements in SQL syntax" on page 354.

except-query-block : A query block. See "Common elements in SQL syntax" on page 354.

option-name : *identifier*

option-value : *hostvar* (indicator allowed), *string*, *identifier*, or *number*

Parameters

- **main-query-block** A query block against which one or more query blocks is compared.
- **except-query-block** A query block to the right of an EXCEPT clause. The result of an *except-query-block* is compared with the result of *main-query-block* to identify rows that exist only in *main-query-block*.
- **EXCEPT clause** If there are duplicate rows in *main-query-block* that do not match rows in the *except-query-block*(s), the duplicates appear in the result if EXCEPT ALL is specified. To suppress duplicates in the result, specify EXCEPT or EXCEPT DISTINCT instead.

- **OPTION clause** Use this clause to specify hints for executing the statement. The following hints are supported:

  - MATERIALIZED VIEW OPTIMIZATION *option-value*
  - FORCE OPTIMIZATION
  - *option-name* = *option-value*

  For a description of these options, see the OPTIONS clause of the “SELECT statement” on page 728.

Remarks

Use the EXCEPT clause to compare the results of main-query-block to one or more except-query-blocks, and return only those rows that exist in main-query-block. Specify EXCEPT or EXCEPT DISTINCT if you do not want duplicates in main-query-block to appear as duplicates in the results. Otherwise, specify EXCEPT ALL.

EXCEPT is identical to EXCEPT DISTINCT.

The *query-block* must each have the same number of items in the select list.

The number of rows in the result set of EXCEPT ALL is exactly the difference between the number of rows in the result sets of the separate queries.

The results of EXCEPT are the same as EXCEPT ALL, except that when using EXCEPT, duplicate rows are eliminated before the difference between the result sets is computed.

If corresponding items in two select lists have different data types, SQL Anywhere chooses a data type for the corresponding column in the result and automatically convert the columns in each *query-block* appropriately. The first query specification of the UNION is used to determine the names to be matched with the ORDER BY clause.

The column names displayed are the same column names that are displayed for the first *query-block*. An alternative way of customizing result set column names is to use the WITH clause on the *query-block*.

Permissions

Must have SELECT permission for each *query-block*.

Side effects

None

See also

- “EXCEPT clause” on page 595
- “INTERSECT clause” on page 657
- “UNION clause” on page 771
- “SELECT statement” on page 728

Standards and compatibility

- **SQL/2003** EXCEPT DISTINCT is a core feature. EXCEPT ALL is feature F304.

Example

For examples of EXCEPT usage, see “Set operators and NULL” [*SQL Anywhere Server - SQL Usage*].

EXECUTE statement [ESQL]

Use this statement to execute a prepared SQL statement.

Syntax 1

EXECUTE *statement*
[**USING** { *hostvar-list* | **DESCRIPTOR** *sqlda-name* }]
[**INTO** { *into-hostvar-list* | **DESCRIPTOR** *into-sqlda-name* }]
[**ARRAY :***row-count*]

row-count : *integer* or *hostvar*

statement : *identifier* | *hostvar* | *string*

sqlda-name : *identifier*

into-sqlda-name : *identifier*

Syntax 2

EXECUTE IMMEDIATE *statement*

statement : *string* | *hostvar*

Parameters

- **USING clause** Results from a SELECT statement or a CALL statement are put into either the variables in the variable list or the program data areas described by the named SQLDA. The correspondence is one-to-one from the OUTPUT (selection list or parameters) to either the host variable list or the SQLDA descriptor array.
- **INTO clause** If EXECUTE INTO is used with an INSERT statement, the inserted row is returned in the second descriptor. For example, when using auto-increment primary keys or BEFORE INSERT triggers that generate primary key values, the EXECUTE statement provides a mechanism to re-fetch the row immediately and determine the primary key value that was assigned to the row. The same thing can be achieved by using @@identity with auto-increment keys.
- **ARRAY clause** The optional ARRAY clause can be used with prepared INSERT statements to allow wide inserts, which insert more than one row at a time and which can improve performance. The integer value is the number of rows to be inserted. The SQLDA must contain a variable for each entry (number of rows * number of columns). The first row is placed in SQLDA variables 0 to (columns per row)-1, and so on.

Remarks

The EXECUTE statement can be used for any SQL statement that can be prepared. Cursors are used for SELECT statements or CALL statements that return many rows from the database (see “Using cursors in embedded SQL” [*SQL Anywhere Server - Programming*]).

After successful execution of an INSERT, UPDATE or DELETE statement, the *sqlerrd*[*2*] field of the SQLCA (SQLCOUNT) is filled in with the number of rows affected by the operation.

Syntax 1 Execute the named dynamic statement, which was previously prepared. If the dynamic statement contains host variable place holders which supply information for the request (bind variables), either the *sqlda-name* must specify a C variable which is a pointer to a SQLDA containing enough descriptors for all the bind variables occurring in the statement, or the bind variables must be supplied in the *hostvar -list*.

Syntax 2 A short form to PREPARE and EXECUTE a statement that does not contain bind variables or output. The SQL statement contained in the string or host variable is immediately executed, and is dropped on completion.

Permissions

Permissions are checked on the statement being executed.

Side effects

None.

See also

- “EXECUTE IMMEDIATE statement [SP]” on page 600
- “PREPARE statement [ESQL]” on page 694
- “DECLARE CURSOR statement [ESQL] [SP]” on page 552

Standards and compatibility

- **SQL/2003** Feature outside core SQL.

Example

Execute a DELETE.

```
EXEC SQL EXECUTE IMMEDIATE
'DELETE FROM Employees WHERE EmployeeID = 105';
```

Execute a prepared DELETE statement.

```
EXEC SQL PREPARE del_stmt FROM
'DELETE FROM Employees WHERE EmployeeID = :a';
EXEC SQL EXECUTE del_stmt USING :employee_number;
```

Execute a prepared query.

```
EXEC SQL PREPARE sel1 FROM
'SELECT Surname FROM Employees WHERE EmployeeID = :a';
EXEC SQL EXECUTE sel1 USING :employee_number INTO :surname;
```

EXECUTE statement [T-SQL]

Use Syntax 1 to invoke a procedure, as an Adaptive Server Enterprise-compatible alternative to the CALL statement. Use Syntax 2 to execute a prepared SQL statement in Transact-SQL.

Syntax 1

EXECUTE [*@return_status* =] [*creator*.]*procedure_name* [*argument*, ...]

argument :
 [*@parameter-name* =] *expression*
| [*@parameter-name* =] *@variable* [*output*]

Syntax 2

EXECUTE (*string-expression* **)**

Remarks

Syntax 1 executes a stored procedure, optionally supplying procedure parameters and retrieving output values and return status information.

The EXECUTE statement is implemented for Transact-SQL compatibility, but can be used in either Transact-SQL or Watcom-SQL batches and procedures.

With Syntax 2, you can execute statements within Transact-SQL stored procedures and triggers. The EXECUTE statement extends the range of statements that can be executed from within procedures and triggers. It lets you execute dynamically prepared statements, such as statements that are constructed using the parameters passed in to a procedure. Literal strings in the statement must be enclosed in single quotes, and the statement must be on a single line.

The Transact-SQL EXECUTE statement does not have a way to signify that a result set is expected. One way to indicate that a Transact-SQL procedure returns a result set is to include something like the following:

```
IF 1 = 0 THEN
      SELECT 1 AS a
```

You can also execute statements within Transact-SQL stored procedures and triggers. See "EXECUTE IMMEDIATE statement [SP]" on page 600.

Permissions

Must be the owner of the procedure, have EXECUTE permission for the procedure, or have DBA authority.

Side effects

None.

See also

- "CALL statement" on page 419
- "EXECUTE statement [ESQL]" on page 597
- "EXECUTE IMMEDIATE statement [SP]" on page 600

Example

The following procedure illustrates Syntax 1.

```
CREATE PROCEDURE p1( @var INTEGER = 54 )
AS
PRINT 'on input @var = %1!', @var
DECLARE @intvar integer
SELECT @intvar=123
SELECT @var=@intvar
PRINT 'on exit @var = %1!', @var;
```

The following statement executes the procedure, supplying the input value of 23 for the parameter. If you are connected from an Open Client or JDBC application, the PRINT messages are displayed on the client window. If you are connected from an ODBC or embedded SQL application, the messages are displayed on the database server messages window.

```
EXECUTE p1 23;
```

The following is an alternative way of executing the procedure, which is useful if there are several parameters.

```
EXECUTE p1 @var = 23;
```

The following statement executes the procedure, using the default value for the parameter

```
EXECUTE p1;
```

The following statement executes the procedure, and stores the return value in a variable for checking return status.

```
EXECUTE @status = p1 23;
```

EXECUTE IMMEDIATE statement [SP]

Use this statement to enable dynamically-constructed statements to be executed from within a procedure.

Syntax 1

EXECUTE IMMEDIATE [*execute-option*] *string-expression*

execute-option:
  WITH QUOTES [**ON** | **OFF**]
| **WITH ESCAPES** { **ON** | **OFF** }
| **WITH RESULT SET** { **ON** | **OFF** }

Syntax 2

EXECUTE (*string-expression* **)**

Parameters

- **WITH QUOTES clause** When you specify WITH QUOTES or WITH QUOTES ON, any double quotes in the string expression are assumed to delimit an identifier. When you do not specify WITH QUOTES, or specify WITH QUOTES OFF, the treatment of double quotes in the string expression depends on the current setting of the quoted_identifier option.

  WITH QUOTES is useful when an object name that is passed into the stored procedure is used to construct the statement that is to be executed, but the name might require double quotes and the procedure might be called when the quoted_identifier option is set to Off. See "quoted_identifier option [compatibility]" *[SQL Anywhere Server - Database Administration]*.

- **WITH ESCAPES clause** WITH ESCAPES OFF causes any escape sequences (such as \n, \x, or \\) in the string expression to be ignored. For example, two consecutive backslashes remain as two backslashes, rather than being converted to a single backslash. The default setting is equivalent to WITH ESCAPES ON.

One use of WITH ESCAPES OFF is for easier execution of dynamically-constructed statements referencing file names that contain backslashes.

In some contexts, escape sequences in the *string-expression* are transformed before the EXECUTE IMMEDIATE statement is executed. For example, compound statements are parsed before being executed, and escape sequences are transformed during this parsing, regardless of the WITH ESCAPES setting. In these contexts, WITH ESCAPES OFF prevents further translations from occurring. For example:

```
BEGIN
   DECLARE String1 LONG VARCHAR;
   DECLARE String2 LONG VARCHAR;
   EXECUTE IMMEDIATE
        'SET String1 = ''One backslash: \\\\ ''';
     EXECUTE IMMEDIATE WITH ESCAPES OFF
        'SET String2 = ''Two backslashes: \\\\ ''';
   SELECT String1, String2
END
```

- **WITH RESULT SET clause** You can have an EXECUTE IMMEDIATE statement return a result set by specifying WITH RESULT SET ON. With this clause, the containing procedure is marked as returning a result set. If you do not include this clause, an error is reported when the procedure is called if the statement produces a result set.

Note
The default option is WITH RESULT SET OFF, meaning that no result set is produced when the statement is executed.

Remarks

The EXECUTE statement extends the range of statements that can be executed from within procedures and triggers. It lets you execute dynamically-prepared statements, such as statements that are constructed using the parameters passed in to a procedure.

Literal strings in the statement must be enclosed in single quotes, and the statement must be on a single line.

Only global variables can be referenced in a statement executed by EXECUTE IMMEDIATE.

Only syntax 2 can be used inside Transact-SQL stored procedures and triggers.

Permissions

None. The statement is executed with the permissions of the owner of the procedure, not with the permissions of the user who calls the procedure.

Side effects

None. However, if the statement is a data definition statement with an automatic commit as a side effect, that commit does take place.

For more information about using the EXECUTE IMMEDIATE statement in procedures, see "Using the EXECUTE IMMEDIATE statement in procedures" [*SQL Anywhere Server - SQL Usage*].

See also

- "CREATE PROCEDURE statement (web services)" on page 495
- "BEGIN statement" on page 413
- "EXECUTE statement [ESQL]" on page 597

Standards and compatibility

- **SQL/2003** SQL/foundation feature outside core SQL.

Examples

The following procedure creates a table, where the table name is supplied as a parameter to the procedure. The EXECUTE IMMEDIATE statement must all be on a single line.

```
CREATE PROCEDURE CreateTableProc(
                  IN tablename char(30)
                  )
BEGIN
   EXECUTE IMMEDIATE
   'CREATE TABLE ' || tablename ||
   ' ( column1 INT PRIMARY KEY)'
END;
```

To call the procedure and create a table called mytable:

```
CALL CreateTableProc( 'mytable' );
```

For an example of EXECUTE IMMEDIATE with a query that returns a result set, see "Using the EXECUTE IMMEDIATE statement in procedures" [*SQL Anywhere Server - SQL Usage*].

EXIT statement [Interactive SQL]

Use this statement to leave Interactive SQL.

Syntax

{ **EXIT** | **QUIT** | **BYE** } [*return-code*]

return-code: *number* | *connection-variable*

Remarks

This statement closes the Interactive SQL window if you are running Interactive SQL as a windowed program, or terminates Interactive SQL altogether when run in command-prompt (batch) mode. In both cases, the database connection is also closed. Before closing the database connection, Interactive SQL automatically executes a COMMIT statement if the commit_on_exit option is set to On. If this option is set to Off, Interactive SQL performs an implicit ROLLBACK. By default, the commit_on_exit option is set to On.

The optional return code can be used in batch files to indicate success or failure of the commands in an Interactive SQL command file. The default return code is 0.

Permissions

None.

Side effects

This statement automatically performs a commit if option commit_on_exit is set to On (the default); otherwise it performs an implicit rollback.

On Windows operating systems the optional return value is available as ERRORLEVEL.

See also

- "SET OPTION statement" on page 742
- "Using Interactive SQL" [*SQL Anywhere Server - Database Administration*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Examples

The following example sets the Interactive SQL return value to 1 if there are any rows in table T, or to 0 if T contains no rows.

```
CREATE VARIABLE rowCount INT;
CREATE VARIABLE retcode INT;
SELECT COUNT(*) INTO rowCount FROM T;
IF( rowCount > 0 ) THEN
    SET retcode = 1;
ELSE
    SET retcode = 0;
END IF;
EXIT retcode;
```

Note

You cannot write the following the statement because EXIT is an Interactive SQL statement (not a SQL statement), and you cannot include any Interactive SQL statement in other SQL block statements.

```
CREATE VARIABLE rowCount INT;
SELECT COUNT(*) INTO rowCount FROM T;
IF( rowCount > 0 ) THEN
    EXIT 1;    //  <-- not allowed
ELSE
    EXIT 0;    //  <-- not allowed
END IF;
```

EXPLAIN statement [ESQL]

Use this statement to retrieve a text specification of the optimization strategy used for a particular cursor.

Syntax

EXPLAIN PLAN FOR CURSOR *cursor-name*
{ **INTO** *hostvar* | **USING DESCRIPTOR** *sqlda-name* }

cursor-name : *identifier* or *hostvar*

sqlda-name : *identifier*

Remarks

The EXPLAIN statement retrieves a text representation of the optimization strategy for the named cursor. The cursor must be previously declared and opened.

The *hostvar* or *sqlda-name* variable must be of string type. The optimization string specifies in what order the tables are searched, and also which indexes are being used for the searches if any.

This string may be long, depending on the query, and has the following format:

```
table (index), table (index), ...
```

If a table has been given a correlation name, the correlation name will appear instead of the table name. The order that the table names appear in the list is the order in which they are accessed by the database server. After each table is a parenthesized index name. This is the index that is used to access the table. If no index is used (the table is scanned sequentially) the letters "seq" will appear for the index name. If a particular SQL SELECT statement involves subqueries, a colon (:) will separate each subquery's optimization string. These subquery sections will appear in the order that the database server executes the queries.

After successful execution of the EXPLAIN statement, the sqlerrd field of the SQLCA (SQLIOESTIMATE) is filled in with an estimate of the number of input/output operations required to fetch all rows of the query.

A discussion with quite a few examples of the optimization string can be found in "Improving database performance" [*SQL Anywhere Server - SQL Usage*].

Permissions

Must have opened the named cursor.

Side effects

None.

See also

- "DECLARE CURSOR statement [ESQL] [SP]" on page 552
- "PREPARE statement [ESQL]" on page 694
- "FETCH statement [ESQL] [SP]" on page 605
- "CLOSE statement [ESQL] [SP]" on page 423
- "OPEN statement [ESQL] [SP]" on page 683
- "Using cursors in embedded SQL" [*SQL Anywhere Server - Programming*]
- "The SQL Communication Area (SQLCA)" [*SQL Anywhere Server - Programming*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following example illustrates the use of EXPLAIN:

```
EXEC SQL BEGIN DECLARE SECTION;
char plan[300];
EXEC SQL END DECLARE SECTION;
EXEC SQL DECLARE employee_cursor CURSOR FOR
   SELECT EmployeeID, Surname
   FROM Employees
   WHERE Surname like :pattern;
EXEC SQL OPEN employee_cursor;
EXEC SQL EXPLAIN PLAN FOR CURSOR employee_cursor INTO :plan;
printf( "Optimization Strategy: '%s'.n", plan );
```

The plan variable contains the following string:

```
'Employees <seq>'
```

FETCH statement [ESQL] [SP]

Use this statement to reposition a cursor and then get data from it.

Syntax

FETCH *cursor-position cursor-name*
[**INTO** { *hostvar-list* | *variable-list* } | **USING DESCRIPTOR** *sqlda-name*]
[**PURGE**]
[**BLOCK** *n*]
[**FOR UPDATE**]
[**ARRAY** *fetch-count*]
INTO *variable-list* [**FOR UPDATE**]

cursor-position :
   NEXT | **PRIOR** | **FIRST** | **LAST**
| { **ABSOLUTE** | **RELATIVE** } *row-count*

row-count : *number* or *hostvar*

cursor-name : *identifier* or *hostvar*

hostvar-list : may contain indicator variables

variable-list : stored procedure variables

sqlda-name : *identifier*

fetch-count : *integer* or *hostvar*

Parameters

- **INTO clause**   The INTO clause is optional. If it is not specified, the FETCH statement positions the cursor only. The *hostvar-list* is for embedded SQL use only.
- **cursor position**   An optional positional parameter allows the cursor to be moved before a row is fetched. If the fetch includes a positioning parameter and the position is outside the allowable cursor positions, the SQLE_NOTFOUND warning is issued and the SQLCOUNT field indicates the offset from a valid position.

The OPEN statement initially positions the cursor before the first row.

- **NEXT clause** Next is the default positioning, and causes the cursor to be advanced one row before the row is fetched.
- **PRIOR clause** Causes the cursor to be backed up one row before fetching.
- **RELATIVE clause** RELATIVE positioning is used to move the cursor by a specified number of rows in either direction before fetching. A positive number indicates moving forward and a negative number indicates moving backward. So, a NEXT is equivalent to RELATIVE 1 and PRIOR is equivalent to RELATIVE -1. RELATIVE 0 retrieves the same row as the last fetch statement on this cursor.
- **ABSOLUTE clause** The ABSOLUTE positioning parameter is used to go to a particular row. A zero indicates the position before the first row (see "Using cursors in procedures and triggers" [*SQL Anywhere Server - SQL Usage*]).

  A one (1) indicates the first row, and so on. Negative numbers are used to specify an absolute position from the end of the cursor. A negative one (-1) indicates the last row of the cursor.
- **FIRST clause** A short form for ABSOLUTE 1.
- **LAST clause** A short form for ABSOLUTE -1.

Cursor positioning problems
Inserts and some updates to DYNAMIC SCROLL cursors can cause problems with cursor positioning. The database server does not put inserted rows at a predictable position within a cursor unless there is an ORDER BY clause on the SELECT statement. In some cases, the inserted row does not appear at all until the cursor is closed and opened again.

This occurs if a temporary table had to be created to open the cursor. For a description, see "Use work tables in query processing (use All-rows optimization goal)" [*SQL Anywhere Server - SQL Usage*].

The UPDATE statement may cause a row to move in the cursor. This will happen if the cursor has an ORDER BY that uses an existing index (a temporary table is not created).

- **BLOCK clause** Rows may be fetched by the client application more than one at a time. This is referred to as block fetching, prefetching, or multi-row fetching. The first fetch causes several rows to be sent back from the database server. The client buffers these rows, and subsequent fetches are retrieved from these buffers without a new request to the database server.

  The BLOCK clause is for use in embedded SQL only. It gives the client and server a hint about how many rows may be fetched by the application. The special value of 0 means the request is sent to the database server and a single row is returned (no row blocking). The BLOCK clause will reduce the number of rows included in the next prefetch to the BLOCK value. To increase the number of rows prefetched, use the PrefetchRows connection parameter.

  If you do not specify a BLOCK clause, the value specified on OPEN is used. See "OPEN statement [ESQL] [SP]" on page 683.

  FETCH RELATIVE 0 always re-fetches the row.

  If prefetch is disabled for the cursor, the BLOCK clause is ignored and rows are fetched one at a time. If ARRAY is also specified, then the number of rows specified by ARRAY are fetched.

- **PURGE clause** The PURGE clause is for use in embedded SQL only. It causes the client to flush its buffers of all rows, and then send the fetch request to the database server. Note that this fetch request may return a block of rows.
- **FOR UPDATE clause** The FOR UPDATE clause indicates that the fetched row will subsequently be updated with an UPDATE WHERE CURRENT OF CURSOR statement. This clause causes the database server to put an intent lock on the row. The lock is held until the end of the current transaction. See "How locking works" [*SQL Anywhere Server - SQL Usage*] and the FOR UPDATE clause of the "SELECT statement" on page 728.
- **ARRAY clause** The ARRAY clause is for use in embedded SQL only. It allows so-called wide fetches, which retrieve more than one row at a time, and which may improve performance.

  To use wide fetches in embedded SQL, include the fetch statement in your code as follows:

  ```
  EXEC SQL FETCH ... ARRAY nnn
  ```

  where ARRAY *nnn* is the last item of the FETCH statement. The fetch count *nnn* can be a host variable. The SQLDA must contain nnn * (columns per row) variables. The first row is placed in SQLDA variables 0 to (columns per row)-1, and so on.

  For a detailed example of using wide fetches, see "Fetching more than one row at a time" [*SQL Anywhere Server - Programming*].

Remarks

The FETCH statement retrieves one row from the named cursor. The cursor must have been previously opened.

Embedded SQL use A DECLARE CURSOR statement must appear before the FETCH statement in the C source code, and the OPEN statement must be executed before the FETCH statement. If a host variable is being used for the cursor name, the DECLARE statement actually generates code and must be executed before the FETCH statement.

The server returns in SQLCOUNT the number of records fetched, and always returns a SQLCOUNT greater than zero unless there is an error or warning.

If the SQLSTATE_NOTFOUND warning is returned on the fetch, the *sqlerrd*[*2*] field of the SQLCA (SQLCOUNT) contains the number of rows by which the attempted fetch exceeded the allowable cursor positions. The value is 0 if the row was not found but the position is valid; for example, executing FETCH RELATIVE 1 when positioned on the last row of a cursor. The value is positive if the attempted fetch was beyond the end of the cursor, and negative if the attempted fetch was before the beginning of the cursor. The cursor is positioned on the last row if the attempted fetch was beyond the end of the cursor, and on the first row if the attempted fetch was before the beginning of the cursor.

After successful execution of the fetch statement, the *sqlerrd*[*1*] field of the SQLCA (SQLIOCOUNT) is incremented by the number of input/output operations required to perform the fetch. This field is actually incremented on every database statement.

Single row fetch One row from the result of the SELECT statement is put into the variables in the variable list. The correspondence is one-to-one from the select list to the host variable list.

Multi-row fetch One or more rows from the result of the SELECT statement are put into either the variables in *variable-list* or the program data areas described by *sqlda-name*. In either case, the correspondence is one-to-one from the *select-list* to either the *hostvar-list* or the *sqlda-name* descriptor array.

Permissions

The cursor must be opened, and the user must have SELECT permission on the tables referenced in the declaration of the cursor.

Side effects

None.

See also

- "DECLARE CURSOR statement [ESQL] [SP]" on page 552
- "PREPARE statement [ESQL]" on page 694
- "OPEN statement [ESQL] [SP]" on page 683
- "Using cursors in embedded SQL" [*SQL Anywhere Server - Programming*]
- "Using cursors in procedures and triggers" [*SQL Anywhere Server - SQL Usage*]
- "FOR statement" on page 608
- "RESUME statement" on page 716

Standards and compatibility

- **SQL/2003** Core feature. Use in procedures is a Persistent Stored Module feature.

Example

The following is an embedded SQL example:

```
EXEC SQL DECLARE cur_employee CURSOR FOR
SELECT EmployeeID, Surname FROM Employees;
EXEC SQL OPEN cur_employee;
EXEC SQL FETCH cur_employee
INTO :emp_number, :emp_name:indicator;
```

The following is a procedure example:

```
BEGIN
   DECLARE cur_employee CURSOR FOR
      SELECT Surname
      FROM Employees;
   DECLARE name CHAR(40);
   OPEN cur_employee;
   lp: LOOP
      FETCH NEXT cur_employee into name;
      IF SQLCODE <> 0 THEN LEAVE lp END IF;
       ...
   END LOOP;
   CLOSE cur_employee;
END
```

FOR statement

Use this statement to repeat the execution of a statement list once for each row in a cursor.

Syntax

```
[ statement-label : ]
FOR for-loop-name AS cursor-name [ cursor-type ] CURSOR
 { FOR statement  [ FOR { UPDATE cursor-concurrency | FOR READ ONLY } ]
  | USING variable-name }
   DO statement-list
END FOR [ statement-label ]

cursor-type :
NO SCROLL
  | DYNAMIC SCROLL
  | SCROLL
  | INSENSITIVE
  | SENSITIVE


cursor-concurrency : BY { VALUES | TIMESTAMP | LOCK }

variable-name : identifier
```

Parameters

- **NO SCROLL clause** A cursor declared NO SCROLL is restricted to moving forward through the result set using FETCH NEXT and FETCH RELATIVE 0 seek operations.

  As rows cannot be returned to once the cursor leaves the row, there are no sensitivity restrictions on the cursor. When a NO SCROLL cursor is requested, SQL Anywhere supplies the most efficient kind of cursor, which is an asensitive cursor. See "Asensitive cursors" [*SQL Anywhere Server - Programming*].

- **DYNAMIC SCROLL clause** DYNAMIC SCROLL is the default cursor type. DYNAMIC SCROLL cursors can use all formats of the FETCH statement.

  When a DYNAMIC SCROLL cursor is requested, SQL Anywhere supplies an asensitive cursor. When using cursors there is always a trade-off between efficiency and consistency. Asensitive cursors provide efficient performance at the expense of consistency. See "Asensitive cursors" [*SQL Anywhere Server - Programming*].

- **SCROLL clause** A cursor declared SCROLL can use all formats of the FETCH statement. When a SCROLL cursor is requested, SQL Anywhere supplies a value-sensitive cursor. See "Value-sensitive cursors" [*SQL Anywhere Server - Programming*].

  SQL Anywhere must execute value-sensitive cursors in such a way that result set membership is guaranteed. DYNAMIC SCROLL cursors are more efficient and should be used unless the consistent behavior of SCROLL cursors is required.

- **INSENSITIVE clause** A cursor declared INSENSITIVE has its membership fixed when it is opened; a temporary table is created with a copy of all the original rows. FETCHING from an INSENSITIVE cursor does not see the effect of any other INSERT, UPDATE, or DELETE statement, or any other PUT, UPDATE WHERE CURRENT, DELETE WHERE CURRENT operations on a different cursor. It does see the effect of PUT, UPDATE WHERE CURRENT, DELETE WHERE CURRENT operations on the same cursor. See "Insensitive cursors" [*SQL Anywhere Server - Programming*].

- **SENSITIVE clause** A cursor declared SENSITIVE is sensitive to changes to membership or values of the result set. See "Sensitive cursors" [*SQL Anywhere Server - Programming*].
- **FOR UPDATE clause** FOR UPDATE is the default. Cursors default to FOR UPDATE for single-table queries without an ORDER BY clause, or if the ansi_update_constraints option is set to Off. When the ansi_update_constraints option is set to Cursors or Strict, then cursors over a query containing an ORDER BY clause default to READ ONLY. However, you can explicitly mark cursors as updatable using the FOR UPDATE clause.
- **READ ONLY clause** A cursor declared FOR READ ONLY cannot be used in an UPDATE (positioned) or a DELETE (positioned) operation. Because it is expensive to allow updates over cursors with an ORDER BY clause or a join, cursors over a query containing a join of two or more tables are READ ONLY and cannot be made updatable. In response to any request for a cursor that specifies FOR UPDATE, SQL Anywhere provides either a value-sensitive cursor or a sensitive cursor. Insensitive and asensitive cursors are not updatable.

Remarks

The FOR statement is a control statement that allows you to execute a list of SQL statements once for each row in a cursor. The FOR statement is equivalent to a compound statement with a DECLARE for the cursor and a DECLARE of a variable for each column in the result set of the cursor followed by a loop that fetches one row from the cursor into the local variables and executes *statement-list* once for each row in the cursor.

Valid cursor types include dynamic scroll (default), scroll, no scroll, sensitive, and insensitive.

The name and data type of each local variable is derived from the *statement* used in the cursor. With a SELECT statement, the data types are the data types of the expressions in the select list. The names are the select list item aliases, if they exist; otherwise, they are the names of the columns. Any select list item that is not a simple column reference must have an alias. With a CALL statement, the names and data types are taken from the RESULT clause in the procedure definition.

The LEAVE statement can be used to resume execution at the first statement after the END FOR. If the ending *statement-label* is specified, it must match the beginning *statement-label*.

Permissions

None.

Side effects

None.

See also

- "DECLARE CURSOR statement [ESQL] [SP]" on page 552
- "FETCH statement [ESQL] [SP]" on page 605
- "CONTINUE statement [T-SQL]" on page 431
- "LOOP statement" on page 672

Standards and compatibility

- **SQL/2003** Persistent Stored Module feature.

Example

The following fragment illustrates the use of the FOR loop.

```
FOR names AS curs INSENSITIVE CURSOR FOR
SELECT Surname
FROM Employees
DO
   CALL search_for_name( Surname );
END FOR;
```

This fragment also illustrates the use of the FOR loop.

```
BEGIN
    FOR names AS curs SCROLL CURSOR FOR
    SELECT EmployeeID, GivenName FROM Employees where EmployeeID < 130
    FOR UPDATE BY VALUES
    DO
        MESSAGE 'emp: ' || GivenName;
    END FOR;
END
```

The following example shows the FOR loop being using inside of a procedure called myproc, which returns the top 10 employees from the Employees table, depending on the sort order specified when calling the procedure (asc for ascending, and desc for descending).

```
CALL sa_make_object( 'procedure', 'myproc' ) ;
ALTER PROCEDURE myproc (
    IN @order_by VARCHAR(20) DEFAULT NULL
)
RESULT ( Surname person_name_t )
BEGIN
    DECLARE @sql LONG VARCHAR;
    DECLARE @msg LONG VARCHAR;
    DECLARE LOCAL TEMPORARY TABLE temp_names( surnames person_name_t );
    SET @sql = 'SELECT TOP(10) * FROM Employees AS t ' ;

    CASE @order_by
    WHEN 'asc' THEN
        SET @sql = @sql || 'ORDER BY t.Surname ASC';
        SET @msg = 'Sorted ascending by last name: ';
    WHEN 'desc' THEN
        SET @sql = @sql || 'ORDER BY t.Surname DESC';
        SET @msg = 'Sorted ascending by last name: ';
    END CASE;

    FOR loop_name AS SCROLL CURSOR USING @sql
    DO
        INSERT INTO temp_names( surnames ) VALUES( Surname );
        MESSAGE( @msg || Surname ) ;
    END FOR;
    SELECT * FROM temp_names;
END ;
```

Calling the myproc procedure and specifying asc (for example, `CALL myproc( 'asc' );`) returns the following results:

Surname
Ahmed

Surname
Barker
Barletta
Bertrand
Bigelow
Blaikie
Braun
Breault
Bucceri
Butterfield

FORWARD TO statement

Use this statement to send native syntax SQL statements to a remote server.

Syntax 1

FORWARD TO *server-name sql-statement*

Syntax 2

FORWARD TO [*server-name*]

Remarks

The FORWARD TO statement enables users to specify the server to which a passthrough connection is required. The statement can be used in two ways:

- **Syntax 1** Send a single statement to a remote server.
- **Syntax 2** Place SQL Anywhere into passthrough mode for sending a series of statements to a remote server. All subsequent statements are passed directly to the remote server. To turn passthrough mode off, issue FORWARD TO without a *server-name* specification.

  If you encounter an error from the remote server while in passthrough mode, you must still issue a FORWARD TO statement to turn passthrough off.

When establishing a connection to server-name on behalf of the user, the database server uses one of the following:

- A remote login alias set using CREATE EXTERNLOGIN

- If a remote login alias is not set up, the name and password used to communicate with SQL Anywhere

If the connection cannot be made to the server specified, the reason is contained in a message returned to the user.

After statements are passed to the requested server, any results are converted into a form that can be recognized by the client program.

server-name The name of the remote server.

SQL-statement A command in the native SQL syntax of the remote server. The command or group of commands is enclosed in curly brackets ({}) or single quotes.

Note
The FORWARD TO statement is a server directive and cannot be used in stored procedures, triggers, events, or batches.

Permissions

None

Side effects

The remote connection is set to AUTOCOMMIT (unchained) mode for the duration of the FORWARD TO session. Any work that was pending prior to the FORWARD TO statement is automatically committed.

Example

The following example sends a SQL statement to the remote server RemoteASE:

```
FORWARD TO RemoteASE { SELECT * FROM titles };
```

The following example shows a passthrough session with the remote server aseprod:

```
FORWARD TO aseprod
  SELECT * FROM titles
      SELECT * FROM authors
FORWARD TO;
```

Standards and compatibility

- **SQL/2003** Vendor extension.

FROM clause

Use this clause to specify the database tables or views involved in a DELETE, SELECT, or UPDATE statement. When used within a SELECT statement, the FROM clause can also be used in a MERGE or INSERT statement.

Syntax

FROM *table-expression*, ...

table-expression :
  table-name
| *view-name*
| *procedure-name*
| *derived-table*
| *lateral-derived-table*
| *join-expression*
| **(** *table-expression*, ... **)**
| *openstring-expression*
| *apply-expression*
| *contains-expression*

table-name :
[*userid***.**]*table-name*
[[**AS**] *correlation-name*]
[**WITH (** *hint* [...] **)**]

view-name :
[*userid***.**]*view-name* [[**AS**] *correlation-name*]
[**WITH (** *table-hint* **)**]

procedure-name :
[*owner*.]*procedure-name* **(** [*parameter*, ...] **)**
[**WITH (** *column-name data-type*, ... **)**]
[[**AS**] *correlation-name*]

derived-table :
(*select-statement* **)**
[**AS**] *correlation-name* [**(** *column-name*, ... **)**]

lateral-derived-table :
LATERAL (*select-statement* | *table-expression* **)**
[**AS**] *correlation-name* [**(** *column-name*, ... **)**]

join-expression :
table-expression join-operator table-expression
[**ON** *join-condition*]

join-operator :
[**KEY** | **NATURAL**] [*join-type*] **JOIN**
| **CROSS JOIN**

join-type :
  INNER
| **LEFT** [**OUTER**]
| **RIGHT** [**OUTER**]
| **FULL** [**OUTER**]

hint :
table-hint | *index-hint*

table-hint :
READPAST
| **UPDLOCK**
| **XLOCK**
| **FASTFIRSTROW**

| **HOLDLOCK**
| **NOLOCK**
| **READCOMMITTED**
| **READUNCOMMITTED**
| **REPEATABLEREAD**
| **SERIALIZABLE**

index-hint :
NO INDEX
| **INDEX (** *index-name* [, ...] **)** [**INDEX ONLY** { **ON** | **OFF** }]
| **FORCE INDEX (** *index-name* **)**

openstring-expression :
OPENSTRING ({ **FILE** | **VALUE** } *string-expression* **)**
WITH (*rowset-schema* **)**
 [**OPTION (** *scan-option* ... **)**]
 [**AS**] *correlation-name*

apply-expression :
table-expression { **CROSS** | **OUTER** } **APPLY** *table-expression*

contains-expression :
{*table-name* | *view-name* } **CONTAINS (** *column-name* [,...]**,** *contains-query* **)** [[**AS**] *score-correlation-name*]

rowset-schema :
column-schema-list
| **TABLE** [*owner*.]*table-name* [**(** *column-list* **)**]

column-schema-list :
{ *column-name user-or-base-type* | **filler( )** } [, ...]

column-list :
{ *column-name* | **filler( )** } [, ...]

scan-option :
BYTE ORDER MARK { **ON** | **OFF** }
| **COMMENTS INTRODUCED BY** *comment-prefix*
| **DELIMITED BY** *string*
| **ENCODING** *encoding*
| **ESCAPE CHARACTER** *character*
| **ESCAPES** { **ON** | **OFF** }
| **FORMAT** { **TEXT** | **BCP** }
| **HEXADECIMAL** { **ON** | **OFF** }
| **QUOTE** *string*
| **QUOTES** { **ON** | **OFF** }
| **ROW DELIMITED BY** *string*
| **SKIP** *integer*
| **STRIP** { **ON** | **OFF** | **LTRIM** | **RTRIM** | **BOTH** }

contains-query : string

Parameters

- **table-name** A base table or temporary table. Tables owned by a different user can be qualified by specifying the user ID. Tables owned by groups to which the current user belongs are found by default without specifying the user ID. See "Referring to tables owned by groups" [*SQL Anywhere Server - Database Administration*].

- **view-name** Specifies a view to include in the query. As with tables, views owned by a different user can be qualified by specifying the user ID. Views owned by groups to which the current user belongs are found by default without specifying the user ID.

  Although the syntax permits table hints on views, such hints have no effect.

- **procedure-name** A stored procedure that returns a result set. This clause applies to the FROM clause of SELECT statements only. The parentheses following the procedure name are required even if the procedure does not take parameters. If the stored procedure returns multiple result sets, only the first is used.

  The WITH clause provides a way of specifying column name aliases for the procedure result set. If a WITH clause is specified, the number of columns must match the number of columns in the procedure result set, and the data types must be compatible with those in the procedure result set. If no WITH clause is specified, the column names and types are those defined by the procedure definition. The following query illustrates the use of the WITH clause:

  ```
  SELECT sp.ident, sp.quantity, Products.name
  FROM ShowCustomerProducts( 149 ) WITH ( ident INT, description CHAR(20),
  quantity INT ) sp
     JOIN Products
  ON sp.ident = Products.ID;
  ```

- **derived-table** You can supply a SELECT statement instead of table or view name in the FROM clause. A SELECT statement used in this way is called a derived table, and it must be given an alias. For example, the following statement contains a derived table, MyDerivedTable, which ranks products in the Products table by UnitPrice.

  ```
  SELECT TOP 3 *
         FROM ( SELECT Description,
                       Quantity,
                       UnitPrice,
                       RANK() OVER ( ORDER BY UnitPrice ASC )
                   AS Rank
                   FROM Products ) AS MyDerivedTable
  ORDER BY Rank;
  ```

  For more information about derived tables, see "Querying derived tables" [*SQL Anywhere Server - SQL Usage*].

- **lateral-derived-table** A derived table, stored procedure, or joined table that may include references to objects in the parent statement (outer references). You must use a lateral derived table if you want to use an outer reference in the FROM clause.

  You can use outer references only to tables that precede the lateral derived table in the FROM clause. For example, you cannot use an outer reference to an item in the *select-list*.

  The table and the outer reference must be separated by a comma. For example, the following queries are valid:

  ```
  SELECT *
   FROM A, LATERAL( B LEFT OUTER JOIN C ON ( A.x = B.x ) ) LDT;

  SELECT *
   FROM A, LATERAL( SELECT * FROM B WHERE A.x = B.x ) LDT;
  ```

```
SELECT *
  FROM A, LATERAL( procedure-name( A.x ) ) LDT;
```

Specifying LATERAL (*table-expression*) is equivalent to specifying LATERAL (SELECT * FROM *table-expression*).

- **openstring-expression** Specify an OPENSTRING clause to query within a file or a BLOB, treating the content of these sources as a set of rows. When doing so, you also specify information about the schema of the file or BLOB for the result set to be generated, since you are not querying a defined structure such as a table or view. This clause applies to the FROM clause of a SELECT statement. It is not supported for UPDATE or DELETE statements.

  The ROWID function is supported over the result set of a table generated by an OPENSTRING expression.

  The following are the subclauses and parameters of the OPENSTRING clause, and how to use them to define and query data within files and BLOBs:

  - **FILE and VALUE clauses** Use the FILE clause to specify the file to query. Use the VALUE clause to specify the BLOB expression to query. The data type for the BLOB expression is assumed to be LONG BINARY. You can specify the READ_CLIENT_FILE function as a value to the VALUE clause.

    If neither the FILE nor VALUE keyword is specified, VALUE is assumed.

  - **WITH clause** Use this clause to specify the rowset schema (column names and data types) of the data being queried. You can specify the columns directly (for example, `WITH ( Surname CHAR(30), GivenName CHAR(30) )`). You can also use the TABLE subclause to reference a table to use to obtain schema information from (for example, `WITH TABLE dba.Employees ( Surname, GivenName )`). You must own, or have SELECT permissions on, the table you specify.

    When specifying columns, you can specify filler( ) for columns you want to skip in the input data (for example, `WITH ( filler( ), Surname CHAR(30), GivenName CHAR(30) )`). For more information about the use of filler( ), see "LOAD TABLE statement" on page 660.

  - **OPTION clause** Use the OPTION clause to specify parsing options to use for the input file, such as escape characters, delimiters, encoding, and so on. Supported options comprise those options for the LOAD TABLE statement that control the parsing of an input file. See "LOAD TABLE statement" on page 660.

- **scan-option** For a description of each scan option, see the load options described in "LOAD TABLE statement" on page 660.

- **apply-expression** Use this clause to specify a join condition where the right *table-expression* is evaluated for every row in the left *table-expression*. For example, you can use an apply expression to evaluate a function, procedure, or derived table for each row in a table expression. See "Joins resulting from apply expressions" [*SQL Anywhere Server - SQL Usage*].

- **contains-expression** Use the CONTAINS clause following a table name to filter the table and return only those rows matching the full text query specified with *contains-query*. Every matching row of the table is returned together with a score column that can be referred to using *score-correlation-name*, if it is specified. If *score-correlation-name* is not specified, then the score column can be referred to by the default correlation name, contains.

With the exception of the optional correlation name argument, the CONTAINS clause takes the same arguments as those of the CONTAINS search condition. See "CONTAINS search condition" on page 48.

There must be a text index on the columns listed in the CONTAINS clause. See "Text indexes" [*SQL Anywhere Server - SQL Usage*].

- **correlation-name** Use *correlation-name* to specify a substitute name for a table or view in the FROM clause. The substitute name can then be referenced from elsewhere in the statement. For example, emp and dep are correlation names for the Employees and Departments tables, respectively:

```
SELECT Surname, GivenName, DepartmentName
   FROM Employees emp, Departments dep,
   WHERE emp.DepartmentID=dep.DepartmentID;
```

- **WITH table-hint clause** The WITH *table-hint* clause allows you to specify the behavior to be used only for this table, and only for this statement. Use this clause to change the behavior without changing the isolation level or setting a database or connection option. Table hints can be used for base tables, temporary tables, and materialized views.

Caution
The WITH *table-hint* clause is an advanced feature that should be used only if needed, and only by experienced database administrators. In addition, the setting may not be respected in all situations.

- **Isolation level related table hints** The isolation level table hints are used to specify isolation level behavior when querying tables. They specify a locking method that is used only for the specified tables, and only for the current query. You cannot specify snapshot isolation levels as table hints.

Following is the list of supported isolation level related table hints:

Table hint	Description
HOLDLOCK	Sets the behavior to be equivalent to isolation level 3. This table hint is synonymous with SERIALIZABLE.
NOLOCK	Sets the behavior to be equivalent to isolation level 0. This table hint is synonymous with READUNCOMMITTED.
READCOM-MITTED	Sets the behavior to be equivalent to isolation level 1.
READPAST	Instructs the database server to ignore, instead of block on, write-locked rows. This table hint can only be used with isolation level 1. The READPAST hint is respected only when the correlation name in the FROM clause refers to a base or globally shared temporary table. In other situations (views, proxy tables, and table functions) the READPAST hint is ignored. Queries within views may utilize READPAST as long as the hint is specified for a correlation name that is a base table. The use of the READPAST table hint can lead to anomalies due to the interaction of locking and predicate evaluation within the server. In addition, you cannot use the READPAST hint against tables that are the targets of a DELETE, INSERT or UPDATE statement.

Table hint	Description
READUNCOMMITTED	Sets the behavior to be equivalent to isolation level 0. This table hint is synonymous with NOLOCK.
REPEATABLEREAD	Sets the behavior to be equivalent to isolation level 2.
SERIALIZABLE	Sets the behavior to be equivalent to isolation level 3. This table hint is synonymous with HOLDLOCK.
UPDLOCK	Indicates that rows processed by the statement from the hinted table are locked using intent locks. The affected rows remain locked until the end of the transaction. UPDLOCK works at all isolation levels and uses intent locks. See "Intent locks" [*SQL Anywhere Server - SQL Usage*].
XLOCK	Indicates that rows processed by the statement from the hinted table are to be locked exclusively. The affected rows remain locked until the end of the transaction. XLOCK works at all isolation levels and uses write locks. See "Write locks" [*SQL Anywhere Server - SQL Usage*].

For information about isolation levels, see "Isolation levels and consistency" [*SQL Anywhere Server - SQL Usage*].

Using READPAST with MobiLink synchronization
If you are writing queries for databases that participate in MobiLink synchronization, it is recommended that you do not use the READPAST table hint in your synchronization scripts.

For more information, see:

- "download_cursor table event" [*MobiLink - Server Administration*]
- "download_delete_cursor table event" [*MobiLink - Server Administration*]
- "upload_fetch table event" [*MobiLink - Server Administration*]

If you are considering READPAST because your application performs many updates that affect download performance, an alternative solution is to use snapshot isolation. See "MobiLink isolation levels" [*MobiLink - Server Administration*].

- **Optimization table hint (FASTFIRSTROW)** The FASTFIRSTROW table hint allows you to set the optimization goal for the query without setting the optimization_goal option to First-row. When you use FASTFIRSTROW, SQL Anywhere chooses an access plan that is intended to reduce the time to fetch the first row of the query's result. See "optimization_goal option [database]" [*SQL Anywhere Server - Database Administration*].
- **WITH (index-hint) clause** The WITH (*index-hint*) clause allows you to override the query optimizer plan selection algorithms, and tell the optimizer exactly how to access the table using indexes. Index hints can be used for base tables, temporary tables, and materialized views.

- **NO INDEX** Use this clause to force a sequential scan of the table (indexes are not used). Note that sequential scans may be very costly.
- **INDEX (index-name [,...])** Use this clause to specify up to four indexes that the optimizer must use to satisfy the query. If any of the specified indexes cannot be used, an error is returned.
- **INDEX ONLY { ON | OFF }** Use this clause to control whether an index-only retrieval of data is performed. If the INDEX (*index-name*...) clause is specified with INDEX ONLY ON, the database server attempts an index-only retrieval using the specified indexes. If any of the specified indexes cannot be used in satisfying an index-only retrieval, an error is returned (for example, if there are no indexes, or if the existing indexes cannot satisfy the query).

  Specify INDEX ONLY OFF to prevent an index-only retrieval.
- **FORCE INDEX (index-name)** Use this clause to specify the index that the optimizer must use to find rows in the table that satisfy the query. The FORCE INDEX (*index-name*) syntax is provided for compatibility, and does not support specifying more than one index.

  Caution
  Index hints override the query optimizer's decision making logic, and so should be used only by experienced users. Using index hints may lead to suboptimal access plans and poor performance.

Remarks

The SELECT, UPDATE, and DELETE statements require a table list to specify which tables are used by the statement.

Views and derived tables
Although the FROM clause description refers to tables, it also applies to views and derived tables unless otherwise noted.

The FROM clause creates a result set consisting of all the columns from all the tables specified. Initially, all combinations of rows in the component tables are in the result set, and the number of combinations is usually reduced by JOIN conditions and/or WHERE conditions.

You cannot use an ON phrase with CROSS JOIN.

For Syntax 2, if neither the FILE clause nor VALUE clause is specified, VALUE is assumed. That is, *string-expression* is assumed to be the value to be queried.

Permissions

The FILE clause of *openstring-expression* requires either DBA or READFILE authority.

The TABLE clause of *openstring-expression* requires the user to own, or have SELECT permissions on, the specified table.

Side effects

None.

See also

- “DELETE statement” on page 558
- “SELECT statement” on page 728
- “UPDATE statement” on page 777
- “INSERT statement” on page 649
- “MERGE statement” on page 673
- “Joins: Retrieving data from several tables” [*SQL Anywhere Server - SQL Usage*]
- “MultipleIndexScan method (MultIdx)” [*SQL Anywhere Server - SQL Usage*]

Standards and compatibility

- **SQL/2003** Core feature, except for the list below. The complexity of the FROM clause means that you should check individual clauses against the standard.
  - KEY JOIN, which is a vendor extension
  - FULL OUTER JOIN and NATURAL JOIN, which are SQL/foundation features outside core SQL
  - READPAST table hint, which is a vendor extension
  - LATERAL (*table-expression*), which is a vendor extension. Note that LATERAL (*select-statement*) is in the ANSI SQL standard as feature T491.
  - derived tables are feature F591
  - procedures in the FROM clause (table functions) are feature T326
  - common table expressions are feature T121
  - recursive table expressions are feature T131

Example

The following are valid FROM clauses:

```
...
FROM Employees
...
```

```
...
FROM Employees NATURAL JOIN Departments
...
```

```
...
FROM Customers
KEY JOIN SalesOrders
KEY JOIN SalesOrderItems
KEY JOIN Products
...
```

```
...
FROM Employees CONTAINS ( Street, ' Way ' )
...
```

The following query illustrates how to use derived tables in a query:

```
SELECT Surname, GivenName, number_of_orders
FROM Customers JOIN
     ( SELECT CustomerID, COUNT(*)
         FROM SalesOrders
          GROUP BY CustomerID )
       AS sales_order_counts( CustomerID,
                              number_of_orders )
ON ( Customers.ID = sales_order_counts.CustomerID )
WHERE number_of_orders > 3;
```

The following query illustrates how to select rows from stored procedure result sets:

```
SELECT t.ID, t.QuantityOrdered AS q, p.name
FROM ShowCustomerProducts( 149 ) t JOIN Products p
ON t.ID = p.ID;
```

The following example illustrates how to perform a query using the OPENSTRING clause to query a file. The CREATE TABLE statement creates a table called testtable with two columns, column1 and columns2. The UNLOAD statement creates a file called *testfile.dat* by unloading rows from the RowGenerator table. The SELECT statement uses the OPENSTRING clause in a FROM clause to query *testfile.dat* using the schema information from both the testtable and RowGenerator tables. The query returns one row with the value 49.

```
CREATE TABLE testtable( column1 CHAR(10), column2 INT );
UNLOAD SELECT * FROM RowGenerator TO 'testfile.dat';
SELECT A.column2
  FROM OPENSTRING( FILE 'testfile.dat' )
  WITH ( TABLE testtable( column2 ) ) A, RowGenerator B
  WHERE A.column2 = B.row_num
  AND A.column2 < 50
  AND B.row_num > 48;
```

The following example illustrates how to perform a query using the OPENSTRING clause to query a string value. The SELECT statement uses the OPENSTRING clause in a FROM clause to query a string value using the schema information provided in the WITH clause. The query returns two columns with three rows.

```
SELECT *
  FROM OPENSTRING( VALUE '1,"First"$2,"Second"$3,"Third"')
  WITH (c1 INT, c2 VARCHAR(30))
  OPTION ( DELIMITED BY ',' ROW DELIMITED BY '$')
  AS VALS
```

GET DATA statement [ESQL]

Use this statement to get string or binary data for one column of the current row of a cursor. GET DATA is usually used to fetch LONG BINARY or LONG VARCHAR fields. See "SET statement" on page 735.

Syntax

GET DATA *cursor-name*
COLUMN *column-num*
OFFSET *start-offset*
[**WITH TEXTPTR**]
USING DESCRIPTOR *sqlda-name* | **INTO** *hostvar*, ...

cursor-name : *identifier*, or *hostvar*

column-num : *integer* or *hostvar*

start-offset : *integer* or *hostvar*

sqlda-name : *identifier*

Parameters

- **COLUMN clause** The value of *column-num* starts at one, and identifies the column whose data is to be fetched. That column must be of a string or binary type.
- **OFFSET clause** The *start-offset* indicates the number of bytes to skip over in the field value. Normally, this would be the number of bytes previously fetched. The number of bytes fetched on this GET DATA statement is determined by the length of the target host variable.

  The indicator value for the target host variable is a short integer, so it cannot always contain the number of bytes truncated. Instead, it contains a negative value if the field contains the NULL value, a positive value (NOT necessarily the number of bytes truncated) if the value is truncated, and zero if a non-NULL value is not truncated.

  Similarly, if a LONG VARCHAR or a LONG VARCHAR host variable is used with an offset greater than zero, the untrunc_len field does not accurately indicate the size before truncation.
- **WITH TEXTPTR clause** If the WITH TEXTPTR clause is given, a text pointer is retrieved into a second host variable or into the second field in the SQLDA. This text pointer can be used with the Transact-SQL READ TEXT and WRITE TEXT statements. The text pointer is a 16-bit binary value, and can be declared as follows:

  ```
  DECL_BINARY( 16 ) textptr_var;
  ```

  The WITH TEXTPTR clause can only be used with long data types (LONG BINARY, LONG VARCHAR, TEXT, IMAGE). If you attempt to use it with another data type, the error INVALID_TEXTPTR_VALUE is returned.

  The total length of the data is returned in the SQLCOUNT field of the SQLCA structure.

Remarks

Get a piece of one column value from the row at the current cursor position. The cursor must be opened and positioned on a row, using FETCH.

Permissions

None.

Side effects

None.

See also

- "FETCH statement [ESQL] [SP]" on page 605
- "READTEXT statement [T-SQL]" on page 702

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following example uses GET DATA to fetch a binary large object (also called a BLOB).

```
EXEC SQL BEGIN DECLARE SECTION;
DECL_BINARY(1000) piece;
short ind;

EXEC SQL END DECLARE SECTION;
int size;
/* Open a cursor on a long varchar field */
EXEC SQL DECLARE big_cursor CURSOR FOR
SELECT long_data FROM some_table
WHERE key_id = 2;
EXEC SQL OPEN big_cursor;
EXEC SQL FETCH big_cursor INTO :piece;
for( offset = 0; ; offset += piece.len ) {
   EXEC SQL GET DATA big_cursor COLUMN 1
   OFFSET :offset INTO :piece:ind;
   /* Done if the NULL value */
   if( ind < 0 ) break;
   write_out_piece( piece );
   /* Done when the piece was not truncated */
   if( ind == 0 ) break;
}
EXEC SQL CLOSE big_cursor;
```

GET DESCRIPTOR statement [ESQL]

Use this statement to retrieve information about a variable within a descriptor area, or retrieves its value.

Syntax

GET DESCRIPTOR *descriptor-name*
{ *hostvar* = **COUNT** | **VALUE** { *integer* | *hostvar* } *assignment*, ... }

assignment :
 hostvar =
TYPE
| **LENGTH**
| **PRECISION**
| **SCALE**
| **DATA**
| **INDICATOR**
| **NAME**
| **NULLABLE**
| **RETURNED_LENGTH**

descriptor-name : identifier

Remarks

The GET DESCRIPTOR statement is used to retrieve information about a variable within a descriptor area, or to retrieve its value.

The value { *integer* | *hostvar* } specifies the variable in the descriptor area about which the information is retrieved. Type checking is performed when doing GET ... DATA to ensure that the host variable and the descriptor variable have the same data type. LONG VARCHAR and LONG BINARY are not supported by GET DESCRIPTOR ... DATA.

If an error occurs, it is returned in the SQLCA.

Permissions

None.

Side effects

None.

See also

- "ALLOCATE DESCRIPTOR statement [ESQL]" on page 357
- "DEALLOCATE DESCRIPTOR statement [ESQL]" on page 550
- "SET DESCRIPTOR statement [ESQL]" on page 741
- "The SQL descriptor area (SQLDA)" [*SQL Anywhere Server - Programming*]

Standards and compatibility

- **SQL/2003** Core feature.

Example

The following example returns the type of the column with position col_num in sqlda.

```
int get_type( SQLDA *sqlda, int col_num )
{
    EXEC SQL BEGIN DECLARE SECTION;
    int ret_type;
    int col = col_num;
    EXEC SQL END DECLARE SECTION;
EXEC SQL GET DESCRIPTOR sqlda VALUE :col :ret_type = TYPE;
    return( ret_type );
}
```

For a longer example, see "ALLOCATE DESCRIPTOR statement [ESQL]" on page 357.

GET OPTION statement [ESQL]

You can use this statement to get the current setting of an option. It is recommended that you use the CONNECTION_PROPERTY function instead.

Syntax

GET OPTION [*userid*.]*option-name*
{ **INTO** *hostvar* | **USING DESCRIPTOR** *sqlda-name* }

userid : *identifier*, *string*, or *hostvar*

option-name : *identifier*, *string*, or *hostvar*

hostvar : indicator variable allowed

sqlda-name : *identifier*

Remarks

The GET OPTION statement is provided for compatibility with older versions of the software. The recommended way to get the values of options is to use the CONNECTION_PROPERTY system function.

The GET OPTION statement gets the option setting of the option *option-name* for the user *userid* or for the connected user if *userid* is not specified. This is either the user's personal setting or the PUBLIC setting if there is no setting for the connected user. If the option specified is a database option and the user has a temporary setting for that option, then the temporary setting is retrieved.

If *option-name* does not exist, GET OPTION returns the warning SQLE_NOTFOUND.

Permissions

None required.

Side effects

None.

See also

- "SET OPTION statement" on page 742
- "Alphabetical list of system procedures" on page 836
- "CONNECTION_PROPERTY function [System]" on page 157

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement illustrates use of GET OPTION.

```
EXEC SQL GET OPTION 'date_format' INTO :datefmt;
```

GOTO statement [T-SQL]

Use this statement to branch to a labeled statement.

Syntax

label : **GOTO** *label*

Remarks

Any statement in a Transact-SQL procedure, trigger, or batch can be labeled. The label name is a valid identifier followed by a colon. In the GOTO statement, the colon is not used.

Permissions

None.

Side effects

None.

Standards and compatibility

- **SQL/2003** Persistent Stored Module feature.

Example

The following Transact-SQL batch prints the message "yes" on the database server messages window four times:

```
DECLARE @count SMALLINT
SELECT @count = 1
restart:
   PRINT 'yes'
   SELECT @count = @count + 1
   WHILE @count <=4
    GOTO restart
```

GRANT statement

Use this statement to create new user IDs, to grant or change permissions for the specified users, and to create or change passwords.

You can grant permissions on disabled objects. Permissions to disabled objects are stored in the database and become effective when the object is enabled.

Syntax 1 - Grant authorities

GRANT *authority*, ...
 TO *userid*, ...

authority :
BACKUP
| **DBA**
| **PROFILE**
| **READCLIENTFILE**
| **READFILE**
| [**RESOURCE** | **ALL**]
| **VALIDATE**
| **WRITECLIENTFILE**

Syntax 2 - Grant group status or membership in a group

GRANT { **GROUP** | **MEMBERSHIP IN GROUP** *userid*, ... }
 TO *userid*, ...

Syntax 3 - Grant database object permissions

GRANT *permission*, ...
 ON [*owner.*]*object-name*
 TO *userid*, ...
[**WITH GRANT OPTION**]
[**FROM** *userid*]

permission :
ALL [**PRIVILEGES**]
| **ALTER**
| **DELETE**
| **INSERT**
| **REFERENCES** [**(** *column-name*, ... **)**]
| **SELECT** [**(** *column-name*, ... **)**]
| **UPDATE** [**(** *column-name*, ... **)**]

Syntax 4 - Grant execute permission

GRANT EXECUTE ON [*owner*.]{ *procedure-name* | *user-defined-function* }
TO *userid*, ...

Syntax 5 - Grant integrated login

GRANT INTEGRATED LOGIN TO *user-profile-name*, ...
AS USER *userid*

Syntax 6 - Grant Kerberos login

GRANT KERBEROS LOGIN TO *client-Kerberos-principal*, ...
AS USER *userid*

Syntax 7 - Grant connect permissions

GRANT CONNECT TO *userid*, ...
[**AT** *starting-id*]
[**IDENTIFIED BY** *password*, ...]

Syntax 8 - Grant creation permission on a dbspace

GRANT CREATE ON *dbspace-name*
TO *userid*, ...

Parameters

- **AT *starting-id* clause** This clause is not for general purpose use. The clause specifies the internal numeric value to be used for the first user ID in the list.

  The AT starting-id clause is used by the Unload utility.

- **GRANT authority clause** Use this clause to grant one of the authorities listed below:

  - **BACKUP authority** This authority grants the user the ability to back up the database. See "BACKUP authority" [*SQL Anywhere Server - Database Administration*].

  - **DBA authority** This authority grants the user the ability to perform all tasks. This is usually reserved for the person in the organization who is looking after the database. See "DBA authority" [*SQL Anywhere Server - Database Administration*].

  - **PROFILE authority** This authority grants the user the ability to perform profiling and diagnostic operations. See "PROFILE authority" [*SQL Anywhere Server - Database Administration*].

  - **READCLIENTFILE authority** This authority grants the user the ability to read from a file on the client computer, for example when loading data. See "READCLIENTFILE authority" [*SQL Anywhere Server - Database Administration*].
  - **READFILE authority** This authority grants the user the ability to execute a SELECT statement against a file using the OPENSTRING clause. See "READFILE authority" [*SQL Anywhere Server - Database Administration*].
  - **RESOURCE or ALL authority** This authority grants the user the ability to create tables and views. ALL is a synonym for RESOURCE that is compatible with Sybase Adaptive Server Enterprise. See "RESOURCE authority" [*SQL Anywhere Server - Database Administration*].
  - **VALIDATE authority** This authority grants the user the ability to perform the validation operations supported by the various VALIDATE statements, such as validating the database, validating tables and indexes, and validating checksums. It also allows the user to use the Validation utility (dbvalid), and the Validate Database wizard in Sybase Central. See "VALIDATE authority" [*SQL Anywhere Server - Database Administration*].
  - **WRITECLIENTFILE authority** This authority grants the user the ability to write to a file on the client computer, for example when unloading data. See "WRITECLIENTFILE authority" [*SQL Anywhere Server - Database Administration*].
- **GROUP clause** This permission allows the user(s) to have members. See "Managing groups" [*SQL Anywhere Server - Database Administration*].
- **MEMBERSHIP IN GROUP clause** This permission grants a user membership in a group. The user inherits the inheritable permissions and authorities set at the group level. See "Managing groups" [*SQL Anywhere Server - Database Administration*].
- **GRANT permission clause** The GRANT *permission* clause allows you to grant permission on individual tables or views. The table permissions can be specified individually, or you can use ALL to grant all permissions at once. The following is a list of grantable permissions:
  - **ALL permission** This permission grants ALTER, DELETE, INSERT, REFERENCES, SELECT, and UPDATE permissions. ALL is a synonym for RESOURCE.
  - **ALTER permission** This permission allows the user to alter the named table with the ALTER TABLE statement. This permission is not allowed for views.
  - **DELETE permission** This permission allows the user to delete rows from the named table or view.
  - **INSERT permission** This permission allows the user to insert rows into the named table or view.
  - **REFERENCES permission** This permission allows the user to create indexes on the named table, and foreign keys that reference the named tables. If column names are specified, the user can reference only those columns. REFERENCES permissions on columns cannot be granted for views, only for tables. INDEX is a synonym for REFERENCES.
  - **SELECT permission** This permission allows the user to view information in the view or table. If column names are specified, the users are allowed to view only those columns. SELECT permissions on columns cannot be granted for views, only for tables.
  - **UPDATE permission** This permission allows the user to update rows in the view or table. If column names are specified, the user can update only those columns.

- **FROM clause** If FROM *userid* is specified, the *userid* is recorded as a grantor user ID in the system tables. This clause is for use by the Unload utility (dbunload). Do not use or modify this option directly.
- **CONNECT TO clause**

  Note
  It is recommended that you use the CREATE USER statement to create users. See "CREATE USER statement" on page 546.

  Creates a new user. GRANT CONNECT can also be used by any user to change their own password. To create a user with an empty string as the password, use:

  ```
  GRANT CONNECT TO userid IDENTIFIED BY "";
  ```

  To create a user with no password, use:

  ```
  GRANT CONNECT TO userid;
  ```

  A user with no password cannot connect to the database. This is useful if you are creating a group and do not want anyone to connect to the database using the group user ID. A user ID must be a valid identifier.

  User IDs and passwords cannot:

  ○ begin with white space, single quotes, or double quotes
  ○ end with white space
  ○ contain semicolons

  A password can be either a valid identifier, or a string (maximum 255 bytes) placed in single quotes. For information about specifying a valid password, see "Setting a password" [*SQL Anywhere Server - Database Administration*].

  The verify_password_function option can be used to specify a function to implement password rules (for example, passwords must include at least one digit). If a password verification function is used, you cannot specify more than one user ID and password in the GRANT CONNECT statement. See "verify_password_function option [database]" [*SQL Anywhere Server - Database Administration*].
- **CREATE ON clause** Allows users to create database objects in the specified dbspace. The CREATE permission can be inherited through group membership. Before a user can create objects, they must also have RESOURCE authority. See "RESOURCE authority" [*SQL Anywhere Server - Database Administration*].

Remarks

The GRANT statement is used to grant database permissions and authorities to individual user IDs and groups. It is also used to create users and groups.

If WITH GRANT OPTION is specified, then the named user ID is also given permission to GRANT the same permissions to other user IDs. Members of groups do not inherit the WITH GRANT OPTION if it is granted to a group.

Syntax 4 of the GRANT statement is used to grant permission to execute a procedure.

Syntax 5 of the GRANT statement creates an explicit integrated login mapping between one or more Windows user or group profiles and an existing database user ID, allowing users who successfully log in to their local computer to connect to a database without having to provide a user ID or password. The *user-profile-name* can be of the form *domain\user-name*. The database user ID the integrated login is mapped to must have a password. See "Using integrated logins" [*SQL Anywhere Server - Database Administration*].

Syntax 6 of the GRANT statement creates a Kerberos authenticated login mapping from one or more Kerberos principals to an existing database user ID. This allows users who have successfully logged in to Kerberos (users who have a valid Kerberos ticket-granting ticket) to connect to a database without having to provide a user ID or password. The database user ID the Kerberos login is mapped to must have a password. The *client-Kerberos-principal* must have the format *user*/*instance*@*REALM*, where /*instance* is optional. The full principal, including the realm, must be specified, and principals that differ only in the instance or realm are treated as different.

Principals are case sensitive and must be specified in the correct case. Mappings for multiple principals that differ only in case are not supported (for example, you cannot have mappings for both jjordan@MYREALM.COM and JJordan@MYREALM.COM).

If no explicit mapping is made for a Kerberos principal, and the Guest database user ID exists and has a password, then the Kerberos principal connects using the Guest database user ID (the same Guest database user ID as for integrated logins).

For more information about Kerberos authentication, see "Kerberos authentication" [*SQL Anywhere Server - Database Administration*].

Permissions

Syntax 3 If the FROM clause is specified, you must have DBA authority. Otherwise, you must either own the table, or have been granted permissions on the table WITH GRANT OPTION.

Syntax 4 You must either own the procedure, or have DBA authority.

Syntax 5 and 6 You must have DBA authority.

Syntax 7 You must either be changing your own password using GRANT CONNECT, or have DBA authority. If you are changing another user's password (with DBA authority), the other user must not be connected to the database.

Side effects

Automatic commit.

See also

- "REVOKE statement" on page 718
- "Database permissions and authorities overview" [*SQL Anywhere Server - Database Administration*]
- "CREATE USER statement" on page 546

Standards and compatibility

- **SQL/2003** Syntax 3 is a core feature. Syntax 4 is a Persistent Stored Module feature. Other syntaxes are vendor extensions.

Example

Create a new database user.

```
GRANT CONNECT TO SQLTester
IDENTIFIED BY welcome
```

Grant permissions on the Employees table to user Laurel.

```
GRANT
SELECT, UPDATE ( Street )
ON Employees
TO Laurel;
```

More than one permission can be granted in a single statement. Separate the permissions with commas.

Allow the user Hardy to execute the Calculate_Report procedure.

```
GRANT EXECUTE ON Calculate_Report
TO Hardy;
```

GRANT CONSOLIDATE statement [SQL Remote]

Use this statement to identify the database immediately above the current database in a SQL Remote hierarchy, who will receive messages from the current database.

Syntax

GRANT CONSOLIDATE
TO *userid*
TYPE *message-system*, ...
ADDRESS *address-string*, ...
[**SEND** { **EVERY** | **AT** } *hh*:*mm*:*ss*]

message-system:
FILE | **FTP** | **SMTP**

address: *string*

Parameters

- **userid** The user ID for the user to be granted the permission.
- **message-system** One of the message systems supported by SQL Remote.
- **address** The address for the specified message system.

Remarks

In a SQL Remote installation, the database immediately above the current database in a SQL Remote hierarchy must be granted CONSOLIDATE permissions. GRANT CONSOLIDATE is issued at a remote database to identify its consolidated database. Each database can have only one user ID with CONSOLIDATE permissions: you cannot have a database that is a remote database for more than one consolidated database.

The consolidated user is identified by a message system, identifying the method by which messages are sent to and received from the consolidated user. The address-name must be a valid address for the message-system, enclosed in single quotes. There can be only one consolidated user per remote database.

For the FILE message type, the address is a subdirectory of the directory pointed to by the SQLREMOTE environment variable.

The GRANT CONSOLIDATE statement is required for the consolidated database to receive messages, but does not by itself subscribe the consolidated database to any data. To subscribe to data, a subscription must be created for the consolidated user ID to one of the publications in the current database. Running the database extraction utility at a consolidated database creates a remote database with the proper GRANT CONSOLIDATE statement already issued.

The optional SEND EVERY and SEND AT clauses specify a frequency at which messages are sent. The string contains a time that is a length of time between messages (for SEND EVERY) or a time of day at which messages are sent (for SEND AT). With SEND AT, messages are sent once per day.

If a user has been granted remote permissions without a SEND EVERY or SEND AT clause, the Message Agent processes messages, and then stops. To run the Message Agent continuously, you must ensure that every user with REMOTE permission has either a SEND AT or SEND EVERY frequency specified.

It is anticipated that at many remote databases, the Message Agent is run periodically, and that the consolidated database will have no SEND clause specified.

Permissions

Must have DBA authority.

Side effects

Automatic commit.

See also

- "CONSOLIDATE permission" [*SQL Remote*]
- "GRANT PUBLISH statement [SQL Remote]" on page 633
- "GRANT REMOTE statement [SQL Remote]" on page 634
- "REVOKE CONSOLIDATE statement [SQL Remote]" on page 720

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

```
GRANT CONSOLIDATE TO con_db
TYPE SMTP
ADDRESS 'Singer, Samuel';
```

GRANT PUBLISH statement [SQL Remote]

Use this statement to identify the publisher of the current database.

Syntax

GRANT PUBLISH TO *userid*

Remarks

Each database in a SQL Remote installation is identified in outgoing messages by a user ID, called the publisher. The GRANT PUBLISH statement identifies the publisher user ID associated with these outgoing messages.

Only one user ID can have PUBLISH authority. The user ID with PUBLISH authority is identified by the special constant CURRENT PUBLISHER. The following query identifies the current publisher:

```
SELECT CURRENT PUBLISHER;
```

If there is no publisher, the special constant is NULL.

The current publisher special constant can be used as a default setting for columns. It is often useful to have a CURRENT PUBLISHER column as part of the primary key for replicating tables, as this helps prevent primary key conflicts due to updates at more than one site.

To change the publisher, you must first drop the current publisher using the REVOKE PUBLISH statement, and then create a new publisher using the GRANT PUBLISH statement.

Permissions

Must have DBA authority.

Side effects

Automatic commit.

See also

- "PUBLISH permission" [*SQL Remote*]
- "GRANT PUBLISH statement [SQL Remote]" on page 633
- "GRANT CONSOLIDATE statement [SQL Remote]" on page 632
- "REVOKE PUBLISH statement [SQL Remote]" on page 721
- "CREATE SUBSCRIPTION statement [SQL Remote]" on page 517

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

```
GRANT PUBLISH TO publisher_ID;
```

GRANT REMOTE statement [SQL Remote]

Use this statement to identify a database immediately below the current database in a SQL Remote hierarchy, who will receive messages from the current database. These are called remote users.

Syntax

GRANT REMOTE TO *userid*, ...
TYPE *message-system*, ...
ADDRESS *address-string*, ...
[**SEND** { **EVERY** | **AT** } *send-time*]

Parameters

- **userid** The user ID for the user to be granted the permission
- **message-system** One of the message systems supported by SQL Remote. It must be one of the following values:
  - FILE
  - FTP
  - SMTP
- **address-string** A string containing a valid address for the specified message system.
- **send-time** A string containing a time specification in the form *hh*:*mm*:*ss*.

Remarks

In a SQL Remote installation, each database receiving messages from the current database must be granted REMOTE permissions.

The single exception is the database immediately above the current database in a SQL Remote hierarchy, which must be granted CONSOLIDATE permissions.

The remote user is identified by a message system, identifying the method by which messages are sent to and received from the consolidated user. The address-name must be a valid address for the message-system, enclosed in single quotes.

For the FILE message type, the address is a subdirectory of the directory pointed to by the SQLREMOTE environment variable.

The GRANT REMOTE statement is required for the remote database to receive messages, but does not by itself subscribe the remote user to any data. To subscribe to data, a subscription must be created for the user ID to one of the publications in the current database, using the database extraction utility or the CREATE SUBSCRIPTION statement.

The optional SEND EVERY and SEND AT clauses specify a frequency at which messages are sent. The string contains a time that is a length of time between messages (for SEND EVERY) or a time of day at which messages are sent (for SEND AT). With SEND AT, messages are sent once per day.

If a user has been granted remote permissions without a SEND EVERY or SEND AT clause, the Message Agent processes messages, and then stops. To run the Message Agent continuously, you must ensure that every user with REMOTE permission has either a SEND AT or SEND EVERY frequency specified.

It is anticipated that at many consolidated databases, the Message Agent is run continuously, so that all remote databases would have a SEND clause specified. A typical setup may involve sending messages to laptop users daily (SEND AT) and to remote servers every hour or two (SEND EVERY). You should use as few different times as possible, for efficiency.

Permissions

Must have DBA authority.

Side effects

Automatic commit.

See also

- "REMOTE permission" [*SQL Remote*]
- "GRANT PUBLISH statement [SQL Remote]" on page 633
- "REVOKE REMOTE statement [SQL Remote]" on page 722
- "GRANT CONSOLIDATE statement [SQL Remote]" on page 632

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement grants remote permissions to user SamS, using an SMTP email system, sending messages to the address Singer, Samuel once every two hours:

```
GRANT REMOTE TO SamS
TYPE SMTP
ADDRESS 'Singer, Samuel'
SEND EVERY '02:00';
```

GRANT REMOTE DBA statement [MobiLink] [SQL Remote]

Use this statement to grant remote DBA privileges to a user ID.

Syntax

GRANT REMOTE DBA
TO *userid*, ...
[**IDENTIFIED BY** *password*]

Parameters

- **IDENTIFIED BY** The IDENTIFIED BY clause is optional for this statement. If included, the password for the user is changed.

Remarks

A user ID with REMOTE DBA authority has full DBA authority only:

- In MobiLink, when the connection is made from the SQL Anywhere synchronization client (dbmlsync) utility. The REMOTE DBA authority enables dbmlsync to have full access to the database to make any changes contained in the messages. Any other connection using the same user ID is granted no special authority. See "Permissions for dbmlsync" [*MobiLink - Client Administration*].

- In SQL Remote, when the connection is made from the Message Agent. The REMOTE DBA authority enables the Message Agent to have full access to the database to make any changes contained in the messages. Any other connection using the same user ID is granted no special authority.

The REMOTE DBA authority avoids having to grant full DBA authority to the user ID, thereby avoiding security problems associated with distributing DBA user IDs and passwords.

For example, a SQL Remote user ID with REMOTE DBA authority has no extra permissions on any connection apart from the Message Agent. Even if the user ID and password for a REMOTE DBA user is widely distributed, there is no security problem. As long as the user ID has no permissions beyond CONNECT granted on the database, no one can use this user ID to access data in the database.

Permissions

Must have DBA authority.

Side effects

Automatic commit.

See also

- MobiLink: "Initiating synchronization" [*MobiLink - Client Administration*]
- SQL Remote: "Grant REMOTE DBA authority" [*SQL Remote*]
- "REVOKE REMOTE DBA statement [SQL Remote]" on page 723

Standards and compatibility

- **SQL/2003** Vendor extension.

Examples

You can grant REMOTE DBA authority to a user ID named **dbremote** as follows:

```
GRANT REMOTE DBA
TO dbremote
IDENTIFIED BY dbremote;
```

GROUP BY clause

Use this clause to group columns, alias names, and functions as part of the SELECT statement.

Syntax

GROUP BY
| *group-by-term*, ...]
| *simple-group-by-term*, ... **WITH ROLLUP**
| *simple-group-by-term*, ... **WITH CUBE**
|**GROUPING SETS (** *group-by-term*, ... **)**

group-by-term :
simple-group-by-term
| **(** *simple-group-by-term*, ... **)**

| **ROLLUP (** *simple-group-by-term*, ... **)**
| **CUBE (** *simple-group-by-term*, ... **)**

simple-group-by-term :
expression
| **(** *expression* **)**
| **( )**

Parameters

- **GROUPING SETS clause** The GROUPING SETS clause allows you to perform aggregate operations on multiple groupings from a single query specification. Each set specified in a GROUPING SET clause is equivalent to a GROUP BY clause.

  For example, the following two queries are equivalent:

  ```
  SELECT a, b, SUM( c ) FROM t
  GROUP BY GROUPING SETS ( ( a, b ), ( a ), ( b ), ( ) );

  SELECT a, b, SUM( c ) FROM t
    GROUP BY a, b
  UNION ALL
  SELECT a, NULL, SUM( c ) FROM t
    GROUP BY a
  UNION ALL
  SELECT NULL, b, SUM( c ) FROM t
    GROUP BY b
  UNION ALL
  SELECT NULL, NULL, SUM( c ) FROM t;
  ```

  An grouping expression may be reflected in the result set as a NULL value, depending on the grouping in which the result row belongs. This may cause confusion over whether the NULL is the result of another grouping, or whether the NULL is the result of an actual NULL value in the underlying data. To distinguish between NULL values present in the input data and NULL values inserted by the grouping operator, use the GROUPING function. See "GROUPING function [Aggregate]" on page 213.

  Specifying an empty set of parentheses ( ) in the GROUPING SETS clause returns a single row containing the overall aggregate.

  For more information about using empty parentheses in GROUPING sets, including an example, see "Specifying an empty grouping specification" [*SQL Anywhere Server - SQL Usage*].

- **ROLLUP clause** The ROLLUP clause is similar to the GROUPING SETS clause in that it can be used to specify multiple grouping specifications within a single query specification. A ROLLUP clause of *n* *simple-group-by-terms* generates *n*+1 grouping sets, formed by starting with the empty parentheses, and then appending successive *group-by-terms* from left to right.

  For example, the following two statements are equivalent:

  ```
  SELECT a, b, SUM( c ) FROM t
  GROUP BY ROLLUP ( a, b );

  SELECT a, b, SUM( c ) FROM t
  GROUP BY GROUPING SETS ( ( a, b ), a, ( ) );
  ```

  You can use a ROLLUP clause within a GROUPING SETS clause.

For more information about ROLLUP operations, see "Using ROLLUP" [*SQL Anywhere Server - SQL Usage*].

- **CUBE clause** The CUBE clause is similar to the ROLLUP and GROUPING SETS clauses in that it can be used to specify multiple grouping specifications within a single query specification. The CUBE clause is used to represent all possible combinations that can be made from the expressions listed in the CUBE clause.

  For example, the following two statements are equivalent:

  ```
  SELECT a, b, SUM( c ) FROM t
  GROUP BY CUBE ( a, b, c );

  SELECT a, b, SUM( c ) FROM t
  GROUP BY GROUPING SETS ( ( a, b, c ), ( a, b ), ( a, c ),
   ( b, c ), a, b, c, () );
  ```

  You can use a CUBE clause within a GROUPING SETS clause.

  For more information about ROLLUP operations, see "Using CUBE" [*SQL Anywhere Server - SQL Usage*].

- **WITH ROLLUP clause** This is an alternative syntax to the ROLLUP clause, and is provided for T-SQL compatibility.

- **WITH CUBE clause** This is an alternate syntax to the CUBE clause, and is provided for T-SQL compatibility.

Remarks

When using the GROUP BY clause, you can group by columns, alias names, or functions. The result of the query contains one row for each distinct value (or set of values) of each grouping set.

See also

- "SELECT statement" on page 728
- "GROUP BY clause extensions" [*SQL Anywhere Server - SQL Usage*]

Standards and compatibility

- **SQL/2003** While the GROUP BY clause is a core feature, GROUPING SETS, ROLLUP, and CUBE are features outside core SQL. For example, the ROLLUP clause is part of feature T431. WITH ROLLUP and WITH CUBE are vendor extensions.

Examples

The following example returns a result set showing the total number of orders, and then provides subtotals for the number of orders in each year (2000 and 2001).

```
SELECT year ( OrderDate ) Year, Quarter ( OrderDate ) Quarter, count(*)
Orders
FROM SalesOrders
GROUP BY ROLLUP ( Year, Quarter )
ORDER BY Year, Quarter;
```

Like the preceding ROLLUP operation example, the following CUBE query example returns a result set showing the total number of orders and provides subtotals for the number of orders in each year (2000 and

2001). Unlike ROLLUP, this query also gives subtotals for the number of orders in each quarter (1, 2, 3, and 4).

```
SELECT year (OrderDate) Year, Quarter ( OrderDate ) Quarter, count(*) Orders
FROM SalesOrders
GROUP BY CUBE ( Year, Quarter )
ORDER BY Year, Quarter;
```

The following example returns a result set that gives subtotals for the number of orders in the years 2000 and 2001. The GROUPING SETS operation lets you select the columns to be subtotaled instead of returning all combinations of subtotals like the CUBE operation.

```
SELECT year (OrderDate) Year, Quarter ( OrderDate ) Quarter, count(*) Orders
FROM SalesOrders
GROUP BY GROUPING SETS ( ( Year, Quarter ), ( Year ) )
ORDER BY Year, Quarter;
```

HELP statement [Interactive SQL]

Use this statement to receive help in the Interactive SQL environment.

Syntax

HELP [**'***topic***'**]

Remarks

The HELP statement is used to access SQL Anywhere documentation.

The *topic* for help can be optionally specified. You must enclose *topic* in single quotes. In some help formats, the topic cannot be specified; in this case, a link to a general help page for Interactive SQL appears.

You can specify the following *topic* values:

- SQL Anywhere error codes
- SQL statement keywords (such as INSERT, UPDATE, SELECT, CREATE DATABASE)

Permissions

None.

Side effects

None.

See also

- "Using Interactive SQL" [*SQL Anywhere Server - Database Administration*]

Standards and compatibility

- **SQL/2003** Vendor extension.

IF statement

Use this statement to control conditional execution of SQL statements.

Syntax

IF *search-condition* **THEN** *statement-list*
[**ELSEIF** { *search-condition* | *operation-type* } **THEN** *statement-list*] ...
[**ELSE** *statement-list*]
{ **END IF** | **ENDIF** }

Remarks

The IF statement is a control statement that allows you to conditionally execute the first list of SQL statements whose *search-condition* evaluates to TRUE. If no *search-condition* evaluates to TRUE, and an ELSE clause exists, the *statement-list* in the ELSE clause is executed.

Execution resumes at the first statement after the END IF.

IF statement is different from IF expression
Do not confuse the syntax of the IF statement with that of the IF expression.

For information about the IF expression, see "IF expressions" on page 18.

Permissions

None.

Side effects

None.

See also

- "BEGIN statement" on page 413
- "Using procedures, triggers, and batches" [*SQL Anywhere Server - SQL Usage*]
- "Search conditions" on page 35

Standards and compatibility

- **SQL/2003** Persistent Stored Module feature.

Example

The following procedure illustrates the use of the IF statement:

```
CREATE PROCEDURE TopCustomer2 (OUT TopCompany CHAR(35), OUT TopValue INT)
BEGIN
   DECLARE err_notfound EXCEPTION
   FOR SQLSTATE '02000';
   DECLARE curThisCust CURSOR FOR
   SELECT CompanyName, CAST(    sum(SalesOrderItems.Quantity *
   Products.UnitPrice) AS INTEGER) VALUE
   FROM Customers
   LEFT OUTER JOIN SalesOrders
   LEFT OUTER JOIN SalesOrderItems
   LEFT OUTER JOIN Products
```

```
      GROUP BY CompanyName;
      DECLARE ThisValue INT;
      DECLARE ThisCompany CHAR(35);
      SET TopValue = 0;
      OPEN curThisCust;
      CustomerLoop:
      LOOP
         FETCH NEXT curThisCust
         INTO ThisCompany, ThisValue;
         IF SQLSTATE = err_notfound THEN
            LEAVE CustomerLoop;
         END IF;
         IF ThisValue > TopValue THEN
            SET TopValue = ThisValue;
            SET TopCompany = ThisCompany;
         END IF;
      END LOOP CustomerLoop;
      CLOSE curThisCust;
   END;
```

IF statement [T-SQL]

Use this statement to control conditional execution of a SQL statement, as an alternative to the Watcom-SQL IF statement.

Syntax

IF *expression statement*
[**ELSE** [**IF** *expression*] *statement*]

Remarks

The Transact-SQL IF conditional and the ELSE conditional each control the execution of only a single SQL statement or compound statement (between the keywords BEGIN and END).

In comparison to the Watcom-SQL IF statement, there is no THEN in the Transact-SQL IF statement. The Transact-SQL version also has no ELSEIF or END IF keywords.

Permissions

None.

Side effects

None.

Standards and compatibility

- **SQL/2003** Transact-SQL extension.

Example

The following example illustrates the use of the Transact-SQL IF statement:

```
IF (SELECT max(ID) FROM sysobjects) < 100
   RETURN
ELSE
         BEGIN
```

```
      PRINT 'These are the user-created objects'
      SELECT name, type, ID
      FROM sysobjects
      WHERE ID < 100
   END
```

The following two statement blocks illustrate Transact-SQL and Watcom-SQL compatibility:

```
/* Transact-SQL IF statement */
IF @v1 = 0
   PRINT '0'
ELSE IF @v1 = 1
   PRINT '1'
ELSE
   PRINT 'other'
/* Watcom-SQL IF statement */
IF v1 = 0 THEN
   PRINT '0'
ELSEIF v1 = 1 THEN
   PRINT '1'
ELSE
   PRINT 'other'
END IF
```

INCLUDE statement [ESQL]

Use this statement to include a file into a source program to be scanned by the SQL preprocessor.

Syntax

INCLUDE *filename*

filename : **SQLDA** | **SQLCA** | *string*

Remarks

The INCLUDE statement is very much like the C preprocessor #include directive. The SQL preprocessor reads an embedded SQL source file and replaces all the embedded SQL statements with C-language source code. If a file contains information that the SQL preprocessor requires, include it with the embedded SQL INCLUDE statement.

Two file names are specially recognized: SQLCA and SQLDA. The following statement must appear before any embedded SQL statements in all embedded SQL source files.

```
EXEC SQL INCLUDE SQLCA;
```

This statement must appear at a position in the C program where static variable declarations are allowed. Many embedded SQL statements require variables (invisible to the programmer), which are declared by the SQL preprocessor at the position of the SQLCA include statement. The SQLDA file must be included if any SQLDAs are used.

Permissions

None.

Side effects

None.

Standards and compatibility

- **SQL/2003** Core feature.

INPUT statement [Interactive SQL]

Use this statement to import data into a database table from an external file or from the keyboard, or import from a generic ODBC data source

Syntax 1 - Import from an external file or from the keyboard

INPUT INTO [*owner*.]*table-name input-options*

input-options :
[**(** *column-name*, ... **)**]
[**BY** { **ORDER** | **NAME**]
[**BYTE ORDER MARK** { **ON** | **OFF** }]
[**COLUMN WIDTHS (** *integer*, ...**)**]
[**DELIMITED BY** *string*]
[**ENCODING** *encoding*]
[**ESCAPE CHARACTER** *character*]
[**ESCAPES** { **ON** | **OFF** }
[**FORMAT** *input-format*]
[**FROM** *filename* | **PROMPT**]
[**NOSTRIP**]

input-format :
TEXT
| **FIXED**

encoding : *identifier* or *string*

Syntax 2 - Import from an ODBC data source

INPUT
USING *connection-string*
FROM *source-table-name*
INTO *destination-table-name*
[**CREATE TABLE** { **ON** | **OFF** }]

connection-string :
{ **DRIVER=***odbc_driver_name*
| **DSN=***odbc_data_source* } [**;** { *connection_parameter* **=** *value* }]

Parameters

- **BY clause** The BY clause allows the user to specify whether the columns from the input file should be matched up with the table columns based on their ordinal position in the list (ORDER, the default) or by their names (NAME). Not all input formats have column name information in the file. NAME is allowed only for those formats that do.

- **BYTE ORDER MARK clause** Use this clause to specify whether to process a byte order mark (BOM) in the data.

  The BYTE ORDER MARK clause is relevant only when reading from TEXT formatted files. Attempts to use the BYTE ORDER MARK clause with FORMAT clauses other than TEXT returns an error.

  The BYTE ORDER MARK clause is used only when reading or writing files encoded with UTF-8 or UTF-16 (and their variants). Attempts to use the BYTE ORDER MARK clause with any other encoding returns an error.

  If the ENCODING clause is specified:

  ○ If the BYTE ORDER MARK option is ON and you specify a UTF-16 encoding with an endian such as UTF-16BE or UTF-16LE, the Interactive SQL searches for a BOM at the beginning of the data. If a BOM is present, it is used to verify the endianness of the data. If you specify the wrong endian, an error is returned.
  ○ If the BYTE ORDER MARK option is ON and you specify a UTF-16 encoding without an explicit endian, Interactive SQL searches for a BOM at the beginning of the data. If a BOM is present, it is used to determine the endianness of the data. Otherwise, the operating system endianness is assumed.
  ○ If the BYTE ORDER MARK option is ON and you specify a UTF-8 encoding, Interactive SQL searches for a BOM at the beginning of the data. If a BOM is present it is ignored.

  If the ENCODING clause is not specified:

  ○ If you do not specify an ENCODING clause and the BYTE ORDER MARK option is ON, Interactive SQL looks for a BOM at the beginning of the input data. If a BOM is located, the source encoding is automatically selected based on the encoding of the BOM (UTF-16BE, UTF-16LE, or UTF-8) and the BOM is not considered to be part of the data to be loaded.
  ○ If you do not specify an ENCODING clause and the BYTE ORDER MARK option is OFF, or a BOM is not found at the beginning of the input data, the database CHAR encoding is used.

- **COLUMN WIDTHS clause** COLUMN WIDTHS can be specified for FIXED format only. It specifies the widths of the columns in the input file. If COLUMN WIDTHS is not specified, the widths are determined by the database column types.

- **CREATE TABLE clause** Use the CREATE TABLE clause to specify whether to create the destination table if it does not exist. The default is ON.

- **DELIMITED BY clause** The DELIMITED BY clause allows you to specify a string to be used as the delimiter in TEXT input format. The default delimiter is a comma.

- **ENCODING clause** The *encoding* argument allows you to specify the encoding that is used to read the file. The ENCODING clause can only be used with the TEXT format.

  For more information about how to obtain the list of SQL Anywhere supported encodings, see "Supported character sets" [*SQL Anywhere Server - Database Administration*].

  For Interactive SQL, if *encoding* is not specified, the encoding that is used to read the file is determined in the following order:

  ○ the encoding specified with the default_isql_encoding option (if this option is set)

  ○ the default encoding of the operating system character set on the computer that Interactive SQL is running on

If the input file was created using the OUTPUT statement and an encoding was specified, then the same ENCODING clause should be specified on the INPUT statement.

For more information about Interactive SQL and encodings, see "default_isql_encoding option [Interactive SQL]" [*SQL Anywhere Server - Database Administration*].

Specify the BYTE ORDER clause to include a byte order mark in the data.

- **ESCAPE CHARACTER clause** The default escape character for hexadecimal codes and symbols is a backslash (\). For example, \x0A is the linefeed character.

  The newline characters can be specified as \n. Other characters can be specified using hexadecimal ASCII codes, such as \x09 for the tab character. A sequence of two backslash characters (\\ ) is interpreted as a single backslash. A backslash followed by any character other than n, x, X or \ is interpreted as two separate characters. For example, \q is interpreted as a backslash and the letter q.

  The escape character can be changed, using the ESCAPE CHARACTER clause. For example, to use the exclamation mark as the escape character, specify:

  ```
  ... ESCAPE CHARACTER '!'
  ```

- **ESCAPES clause** With ESCAPES turned on (the default), characters following the escape character are interpreted as special characters by the database server. With ESCAPES turned off, the characters are read exactly as they appear in the source.

- **FORMAT clause** The FORMAT clause allows you to specify the file format for the output.

  If the FORMAT clause is not specified, then each set of values must be in the format set by the Interactive SQL SET OPTION *input_format* statement.

  Input from a command file is terminated by a line containing END. Input from a file is terminated at the end of the file.

  Allowable input formats are:

  - **TEXT** Input lines are assumed to be characters, one row per line, with column values separated by delimiters. Alphabetic strings may be enclosed in apostrophes (single quotes) or quotation marks (double quotes). Strings containing delimiters must be enclosed in either single or double quotes. If the string itself contains single or double quotes, double the quote character to use it within the string. You can use the DELIMITED BY clause to specify a different delimiter string than the default, which is a comma.

    Three other special sequences are also recognized. The two characters \n represent a newline character, \\ represents a single (\), and the sequence \xDD represents the character with hexadecimal code DD.

    If the file has entries indicating that a value might be null, it is treated as NULL. If the value in that position cannot be NULL, a zero is inserted in numeric columns and an empty string in character columns.

  - **FIXED** Input lines are in fixed format. The width of the columns can be specified using the COLUMN WIDTHS clause. If they are not specified, column widths in the file must be the same as the maximum number of characters required by any value of the corresponding database column's type.

The FIXED format cannot be used with binary columns that contain embedded newline and End-of-File character sequences.

If you want to use other formats such as, DBASEII, DBASE III, FoxPro, Lotus 123, or Excel 97, you need to use INPUT USING.

- **FROM filename clause** The *filename* can be quoted or unquoted. If the string is quoted, it is subject to the same formatting requirements as other SQL strings.

  To indicate directory paths, the backslash character (\) must be represented by two backslashes. The statement to load data from the file *c:\temp\input.dat* into the Employees table is:

  ```
  INPUT INTO Employees
  FROM 'c:\\temp\\input.dat';
  ```

  The location of a relative *filename* is determined as follows:

  - If the INPUT statement is executed directly in Interactive SQL, the path to *filename* is resolved relative to the directory in which Interactive SQL is running. For example, suppose you open Interactive SQL from the directory *c:\work* and execute the following statement:

    ```
    INPUT INTO Employees
     FROM 'inputs\\inputfile.dat';
    ```

    Interactive SQL looks for *c:\work\inputs\inputfile.dat*.

  - If the INPUT statement resides in a *.sql* file, Interactive SQL first attempts to resolve the path to *filename* relative to the location of the file. If unsuccessful, Interactive SQL looks for *filename* in a path relative to the directory in which Interactive SQL is running.

    For example, suppose you had a file, *c:\homework\inputs.sql*, that contained the following statement:

    ```
    INPUT INTO Employees
     FROM 'inputs\\inputfile.dat';
    ```

    Interactive SQL would first look for *inputfile.dat* in *c:\homework\inputs*. If Interactive SQL does not find *inputfile.dat* in that location, Interactive SQL looks in the directory in which Interactive SQL is running.

- **FROM source-table-name clause** The *source-table-name* parameter is a quoted string containing the name of the table in the source database. The name can be in the form *database-name.owner.table-name*, *owner.table-name*, or simply *table-name*. Use a period to separate the components, even if that is not the native separator in the source database. If the source database requires a database name, but not an owner name, the format of *source-table-name* must be *database..table* (in this case the owner name is empty). Do not quote any of the names in the parameter (for example, do not use `'dba."my-table"'`; use `'dba.`*`my-table`*`'` instead.)

- **INTO clause** The name of the table to input the data into.

- **PROMPT clause** The PROMPT clause allows the user to enter values for each column in a row. When running in windowed mode, a window is displayed, allowing the user to enter the values for the new row. If you are running Interactive SQL from the command line, Interactive SQL prompts you to type the value for each column on the command line.

- **NOSTRIP clause** Normally, for TEXT input format, trailing blanks are stripped from unquoted strings before the value is inserted. NOSTRIP can be used to suppress trailing blank stripping. Trailing blanks are not stripped from quoted strings, regardless of whether the option is used. Leading blanks are stripped from unquoted strings, regardless of the NOSTRIP option setting.

- **USING clause** The USING clause inputs data from an ODBC data source. You can either specify the ODBC data source name with the DSN option, or the ODBC driver name and connection parameters with the DRIVER option. *Connection-parameter* is an optional list of database-specific connection parameters.

  odbc-data-source is the name of a user or ODBC data source name. For example, *odbc-data-source* for the SQL Anywhere demo database is SQL Anywhere 11 Demo.

  Odbc-driver-name is the ODBC driver name. For a SQL Anywhere 11 database, the *odbc-driver-name* is SQL Anywhere 11; for an UltraLite database, *odbc-driver-name* is UltraLite 11.

Remarks

The INPUT statement allows efficient mass insertion into a named database table. Lines of input are read either from the user via an input window (if PROMPT is specified) or from a file (if FROM *filename* is specified). If neither is specified, the input is read from the command file that contains the INPUT statement —in Interactive SQL, this can even be directly from the SQL Statements pane.

When the input is read directly from the SQL Statements pane, you must specify a semicolon before the values for the records to be inserted at the end of the INPUT statement. For example:

```
INPUT INTO Owner.TableName;
value1, value2, value3
value1, value2, value3
value1, value2, value3
value1, value2, value3
END;
```

The END statement terminates data for INPUT statements that do not name a file and do not include the PROMPT keyword.

If a column list is specified, the data is inserted into the specified columns of the named table. By default, the INPUT statement assumes that column values in the input file appear in the same order as they appear in the database table definition. If the input file's column order is different, you must list the input file's actual column order at the end of the INPUT statement.

For example, if you create a table with the following statement:

```
CREATE TABLE inventory (
Quantity INTEGER,
item VARCHAR(60)
);
```

and you want to import TEXT data from the input file *stock.txt* that contains the name value before the quantity value,

```
'Shirts', 100
'Shorts', 60
```

then you must list the input file's actual column order at the end of the INPUT statement for the data to be inserted correctly:

```
INPUT INTO inventory
FROM stock.txt
```

```
FORMAT TEXT
(item, Quantity);
```

By default, the INPUT statement stops when it attempts to insert a row that causes an error. Errors can be treated in different ways by setting the on_error and conversion_error options (see SET OPTION). Interactive SQL prints a warning on the Messages tab if any string values are truncated on INPUT. Missing values for NOT NULL columns are set to zero for numeric types and to the empty string for non-numeric types. If INPUT attempts to insert a NULL row, the input file contains an empty row.

Because the INPUT statement is an Interactive SQL command, it cannot be used in any compound statement (such as IF) or in a stored procedure.

See "Statements allowed in procedures, triggers, events, and batches" [*SQL Anywhere Server - SQL Usage*].

Permissions

Must have INSERT permission on the table or view.

Side effects

None.

See also

- "OUTPUT statement [Interactive SQL]" on page 686
- "INSERT statement" on page 649
- "SET OPTION statement [Interactive SQL]" on page 745
- "LOAD TABLE statement" on page 660
- "Importing data" [*SQL Anywhere Server - SQL Usage*]
- "Using Interactive SQL" [*SQL Anywhere Server - Database Administration*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following is an example of an INPUT statement from a TEXT file.

```
INPUT INTO Employees
FROM new_emp.inp
FORMAT TEXT;
```

The following fictitious example copies the table, ulTest, into a table called saTest. ulTest is a table in an UltraLite database in the file *C:\test\myULDatabase.udb*, and saTest is a table created in *demo.db*:

```
INPUT USING 'driver=Ultralite 11;dbf=C:\\test\\myULDatabase.udb'
            FROM "ulTest" INTO "saTest";
```

INSERT statement

Use this statement to insert a single row (syntax 1) or a selection of rows from elsewhere in the database (syntax 2) into a table.

Syntax 1

INSERT [**INTO**] [*owner*.]*table-name* [**(** *column-name*, ... **)**]
[**ON EXISTING** {
  ERROR
  | **SKIP**
  | **UPDATE** [**DEFAULTS** { **ON** | **OFF** }]
  }]
{ **DEFAULT VALUES**
  | **VALUES (** [*expression* | **DEFAULT**, ...**)**] }
[**OPTION(** *query-hint*, ... **)**]

Syntax 2

INSERT [**INTO**] [*owner*.]*table-name* [**(** *column-name*, ... **)**]
[**ON EXISTING** {
  ERROR
  | **SKIP**
  | **UPDATE** [**DEFAULTS** { **ON** | **OFF** }]
  }]
[**WITH AUTO NAME**]
select-statement
[**OPTION(** *query-hint*, ... **)**]

query-hint :
MATERIALIZED VIEW OPTIMIZATION *option-value*
| **FORCE OPTIMIZATION**
| **FORCE NO OPTIMIZATION**
| *option-name* = *option-value*

option-name : *identifier*

option-value : *hostvar* (indicator allowed), *string*, *identifier*, or *number*

Parameters

- **VALUES clause** Use the VALUES clause to specify the values to insert. If you want to insert the default values defined for the columns, specify DEFAULT VALUES. You can also specify VALUES (), which is equivalent to DEFAULT VALUES.
- **WITH AUTO NAME clause** WITH AUTO NAME applies only to syntax 2. If you specify WITH AUTO NAME, the names of the items in the SELECT statement determine which column the data belongs in. The SELECT statement items should be either column references or aliased expressions. Destination columns not defined in the SELECT statement are assigned their default value. This is useful when the number of columns in the destination table is very large.
- **ON EXISTING clause** The ON EXISTING clause of the INSERT statement applies to both syntaxes. It updates existing rows in a table, based on primary key lookup, with new column values. This clause can only be used on tables that have a primary key. Attempting to use this clause on tables without primary keys generates a syntax error. You cannot insert values into a proxy table with the ON EXISTING clause.

Note
If you anticipate many rows qualifying for the ON EXISTING condition, consider using the MERGE statement instead. The MERGE statement provides more control over the actions you can take for matching rows. It also provides a more sophisticated syntax for defining what constitutes a match. See "MERGE statement" on page 673.

If you specify the ON EXISTING clause, the database server performs a primary key lookup for each input row. If the corresponding row does not already exist in the table, it inserts the new row. For rows that already exist in the table, you can choose to silently ignore the input row (SKIP), generate an error message for duplicate key values (ERROR), or update the old values using the values from the input row (UPDATE). By default, if you do not specify the ON EXISTING clause, attempting to insert rows into a table where the row already exists results in a duplicate key value error, and is equivalent to specifying the ON EXISTING ERROR clause.

When using the ON EXISTING UPDATE clause, the input row is compared to the stored row. Any column values explicitly stated in the input row replace the corresponding column values in the stored row. Likewise, column values not explicitly stated in the input row result in no change to the corresponding column values in the stored row—with the exception of columns with defaults. When using the ON EXISTING UPDATE clause with columns that have defaults (including DEFAULT AUTOINCREMENT columns), you can further specify whether to update the column value with the default values by specifying ON EXISTING UPDATE DEFAULTS ON, or leave the column value as it is by specifying ON EXISTING UPDATE DEFAULTS OFF. If nothing is specified, the default behavior is ON EXISTING UPDATE DEFAULTS OFF.

Note
DEFAULTS ON and DEFAULTS OFF parameters do not affect values in DEFAULT TIMESTAMP, DEFAULT UTC TIMESTAMP, or DEFAULT LAST USER. For these columns, the value in the stored row is always updated during the UPDATE.

When using the ON EXISTING SKIP and ON EXISTING ERROR clauses, if the table contains default columns, the server computes the default values even for rows that already exist. As a result, default values such as AUTOINCREMENT cause side effects even for skipped rows. In this case of AUTOINCREMENT, this results in skipped values in the AUTOINCREMENT sequence. The following example illustrates this:

```
CREATE TABLE t1( c1 INT PRIMARY KEY, c2 INT DEFAULT AUTOINCREMENT );
INSERT INTO t1( c1 ) ON EXISTING SKIP VALUES( 20 );
INSERT INTO t1( c1 ) ON EXISTING SKIP VALUES( 20 );
INSERT INTO t1( c1 ) ON EXISTING SKIP VALUES( 30 );
```

The row defined in the first INSERT statement is inserted, and c2 is set to 1. The row defined in the second INSERT statement is skipped because it matches the existing row. However, the autoincrement counter still increments to 2 (but does not impact the existing row). The row defined in the third INSERT statement is inserted, and the value of c2 is set to 3. So, the values inserted for the example above are:

```
20,1
30,3
```

> **Caution**
> If you are using SQL Remote, do not replicate DEFAULT LAST USER columns. When the column is replicated the column value is set to the SQL Remote user, not the replicated value.

- **OPTION clause** Use this clause to specify hints for executing the statement. The following hints are supported:
  - MATERIALIZED VIEW OPTIMIZATION *option-value*
  - FORCE OPTIMIZATION
  - FORCE NO OPTIMIZATION
  - *option-name* = *option-value*

  For a description of these options, see the OPTIONS clause of the "SELECT statement" on page 728.

Remarks

The INSERT statement is used to add new rows to a database table.

Since text indexes and materialized views are impacted by changes to the underlying table data, consider truncating dependent text indexes or materialized views before bulk loading (LOAD TABLE, INSERT, MERGE) data into their underlying tables. See "TRUNCATE statement" on page 768, and "TRUNCATE TEXT INDEX statement" on page 770.

Syntax 1 Insert a single row with the specified expression column values. The keyword DEFAULT can be used to cause the default value for the column to be inserted. If the optional list of column names is given, values are inserted one for one into the specified columns. If the list of column names is not specified, the values are inserted into the table columns in the order they were created (the same order as retrieved with SELECT *). The row is inserted into the table at an arbitrary position. (In relational databases, tables are not ordered.)

Syntax 2 Carry out mass insertion into a table with the results of a fully general SELECT statement. Insertions are done in an arbitrary order unless the SELECT statement contains an ORDER BY clause. See "SELECT statement" on page 728.

If you specify column names, the columns from the select list are matched ordinally with the columns specified in the column list, or sequentially in the order in which the columns were created.

Inserts can be done into views if the query specification defining the view is updatable. For more information about identifying views that are inherently *non-updatable*, see "Working with regular views" [*SQL Anywhere Server - SQL Usage*].

Character strings inserted into tables are always stored in the same case as they are entered, regardless of whether the database is case sensitive or not. So, a string Value inserted into a table is always held in the database with an uppercase V and the remainder of the letters lowercase. SELECT statements return the string as Value. If the database is not case sensitive, however, all comparisons make Value the same as value, VALUE, and so on. Further, if a single-column primary key already contains an entry Value, an INSERT of value is rejected, as it would make the primary key not unique.

Inserting a significant amount of data using the INSERT statement will also update column statistics.

Performance tips
To insert many rows into a table, it is more efficient to declare a cursor and insert the rows through the cursor, where possible, than to carry out many separate INSERT statements. Before inserting data, you can specify the percentage of each table page that should be left free for later updates. See "ALTER TABLE statement" on page 389.

Permissions

Must have INSERT permission on the table.

If the ON EXISTING UPDATE clause is specified, UPDATE permissions on the table are required as well.

Side effects

None.

See also

- "Importing data" [*SQL Anywhere Server - SQL Usage*]
- "MERGE statement" on page 673
- "INPUT statement [Interactive SQL]" on page 644
- "LOAD TABLE statement" on page 660
- "UPDATE statement" on page 777
- "DELETE statement" on page 558
- "PUT statement [ESQL]" on page 697
- "Accessing data on client computers" [*SQL Anywhere Server - SQL Usage*]

Standards and compatibility

- **SQL/2003** Core feature. INSERT ... ON EXISTING is a vendor extension.

Examples

Add an Eastern Sales department to the database.

```
INSERT
INTO Departments ( DepartmentID, DepartmentName )
VALUES ( 230, 'Eastern Sales' );
```

Create the table DepartmentHead and fill it with the names of department heads and their departments using the WITH AUTO NAME syntax.

```
CREATE TABLE DepartmentHead(
      pk INT PRIMARY KEY DEFAULT AUTOINCREMENT,
      DepartmentName VARCHAR(128),
      ManagerName VARCHAR(128) );
INSERT
INTO DepartmentHead WITH AUTO NAME
SELECT GivenName || ' ' || Surname AS ManagerName,
      DepartmentName
FROM Employees JOIN Departments
ON EmployeeID = DepartmentHeadID;
```

Create the table MyTable5 and populate it using the WITH AUTO NAME syntax.

```
CREATE TABLE MyTable5(
      pk INT PRIMARY KEY DEFAULT AUTOINCREMENT,
      TableName CHAR(128),
      TableNameLen INT );
INSERT INTO MyTable5 WITH AUTO NAME
SELECT
      length(t.table_name) AS TableNameLen,
      t.table_name AS TableName
FROM SYS.SYSTAB t
WHERE table_id <= 10;
```

Insert a new department, executing the statement at isolation level 3, rather than using the current isolation level setting of the database.

```
INSERT INTO Departments
   (DepartmentID, DepartmentName, DepartmentHeadID)
   VALUES(600, 'Foreign Sales', 129)
   OPTION( isolation_level = 3 );
```

INSTALL EXTERNAL OBJECT statement

Use this statement to install an object that can be run in an external environment.

Syntax

INSTALL EXTERNAL OBJECT *object-name*
[*update-mode*]
FROM { **FILE** *file-path* | **VALUE** *expression* }
ENVIRONMENT *environment-name*

environment-name :
PERL
| **PHP**

update-mode :
NEW
| **UPDATE**

Parameters

- **object-name** The name by which the installed object will be identified within the database.
- **update-mode** The update mode for the object. If the update mode is omitted, then NEW is assumed.
- **file-path** The location on the server computer from where the object is being installed.
- **environment-name** The name of the external environment in which the external object is run.

Remarks

For more information about external environments, see "Overview of external environments" [*SQL Anywhere Server - Programming*].

Permissions

Must have DBA authority.

Side effects

None

See also

- "Overview of external environments" [*SQL Anywhere Server - Programming*]
- "ALTER EXTERNAL ENVIRONMENT statement" on page 367
- "REMOVE EXTERNAL OBJECT statement" on page 710
- "START EXTERNAL ENVIRONMENT statement" on page 754
- "STOP EXTERNAL ENVIRONMENT statement" on page 761
- "SYSEXTERNENV system view" on page 1000
- "SYSEXTERNENVOBJECT system view" on page 1000

Standards and compatibility

- **SQL/2003** Vendor extension.

Examples

In this example, you install a Perl script that is located in a file into the database.

```
INSTALL EXTERNAL OBJECT 'PerlScript'
NEW
FROM FILE 'perlfile.pl'
ENVIRONMENT PERL;
```

Perl code also can be built and installed from an expression, as follows:

```
INSTALL EXTERNAL OBJECT 'PerlConsoleExample'
NEW
FROM VALUE 'sub WriteToServerConsole { print $sa_output_handle $_[0]; }'
ENVIRONMENT PERL;
```

Perl code also can be built and installed from a variable, as follows:

```
CREATE VARIABLE PerlVariable LONG VARCHAR;
SET PerlVariable =
  'sub WriteToServerConsole { print $sa_output_handle $_[0]; }';

INSTALL EXTERNAL OBJECT 'PerlConsoleExample'
NEW
FROM VALUE PerlVariable
ENVIRONMENT PERL;
```

INSTALL JAVA statement

Use this statement to make Java classes available for use within a database.

Syntax

INSTALL JAVA
[**NEW** | **UPDATE**]
[**JAR** *jar-name*]
FROM { **FILE** *filename* | *expression* }

Parameters

- **NEW and UPDATE keyword clauses** If you specify an install mode of NEW, the referenced Java classes must be new classes, rather than updates of currently installed classes. An error occurs if a class with the same name exists in the database and the NEW install mode is used.

If you specify UPDATE, the referenced Java classes may include replacements for Java classes that are already installed in the given database.

If *install-mode* is omitted, the default is NEW.

- **JAR clause** If this is specified, then the *filename* must designate a jar file. JAR files typically have extensions of *.jar* or *.zip*.

  Installed jar and zip files can be compressed or uncompressed.

  If the JAR option is specified, the jar is retained as a jar after the classes that it contains have been installed. That jar is the associated jar of each of those classes. The jars installed in a database with the JAR option are called the retained jars of the database.

  The *jar-name* is a character string value, of up to 255 bytes long. The *jar-name* is used to identify the retained jar in subsequent INSTALL JAVA, UPDATE, and REMOVE JAVA statements.

- **FROM FILE clause** Specifies the location of the Java class(es) to be installed.

  The formats supported for *filename* include fully qualified file names, such as '*c:\libs\jarname.jar*' and '*/usr/u/libs/jarname.jar*', and relative file names, which are relative to the current working directory of the database server.

  The *filename* must identify either a class file, or a jar file.

- **FROM clause** Expressions must evaluate to a binary type whose value contains a valid class file or jar file.

Remarks

The class definition for each class is loaded by each connection's VM the first time that class is used. When you INSTALL a class, the VM on your connection is implicitly restarted. Therefore, you have immediate access to the new class, whether the INSTALL has an *install-mode* of NEW or UPDATE. Because the VM is restarted, any values stored in Java static variables are lost, and any SQL variables with Java class types are dropped.

For other connections, the new class is loaded the next time a VM accesses the class for the first time. If the class is already loaded by a VM, that connection does not see the new class until the VM is restarted for that connection.

Permissions

DBA permissions are required to execute the INSTALL JAVA statement.

All installed classes can be referenced in any way by any user.

Not supported on Windows Mobile.

See also

- "REMOVE JAVA statement" on page 711

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement installs the user-created Java class named Demo, by providing the file name and location of the class.

```
INSTALL JAVA NEW
FROM FILE 'D:\JavaClass\Demo.class';
```

The following statement installs all the classes contained in a zip file, and associates them within the database with a JAR file name.

```
INSTALL JAVA
JAR 'Widgets'
FROM FILE 'C:\Jars\Widget.zip';
```

Again, the location of the zip file is not retained and classes must be referenced using the fully qualified class name (package name and class name).

INTERSECT clause

Computes the intersection between the result sets of two or more queries.

Syntax

[**WITH** *temporary-views*] *query-block*
  INTERSECT [**ALL** | **DISTINCT**] *query-block*
[**ORDER BY** [*integer* | *select-list-expression-name*] [**ASC** | **DESC**], ...]
[**FOR XML** *xml-mode*]
[**OPTION(** *query-hint*, ... **)**]

query-block : see “Common elements in SQL syntax” on page 354

query-hint :
MATERIALIZED VIEW OPTIMIZATION *option-value*
| **FORCE OPTIMIZATION**
| *option-name* **=** *option-value*

option-name : *identifier*

option-value : *hostvar* (indicator allowed), *string*, *identifier*, or *number*

Parameters

- **OPTION clause** Use this clause to specify hints for executing the statement. The following hints are supported:
  - MATERIALIZED VIEW OPTIMIZATION *option-value*
  - FORCE OPTIMIZATION
  - *option-name* = *option-value*

  For a description of these options, see the OPTIONS clause of the “SELECT statement” on page 728.

Remarks

The intersection between the result sets of several query blocks can be obtained as a single result using INTERSECT or INTERSECT ALL. INTERSECT DISTINCT is identical to INTERSECT.

The query blocks must each have the same number of items in the select list.

The results of INTERSECT are the same as INTERSECT ALL, except that when using INTERSECT, duplicate rows are eliminated before the intersection between the result sets is computed.

If corresponding items in two select lists have different data types, SQL Anywhere chooses a data type for the corresponding column in the result and automatically convert the columns in each *query-block* appropriately. The first *query-block* of the UNION is used to determine the names to be matched with the ORDER BY clause.

The column names displayed are the same column names that are displayed for the first *query-block*. An alternative way of customizing result set column names is to use the WITH clause on the *query-block*.

Permissions

Must have SELECT permission for each *query-block*.

Side effects

None.

See also

- "EXCEPT clause" on page 595
- "INTERSECT clause" on page 657
- "UNION clause" on page 771
- "SELECT statement" on page 728

Standards and compatibility

- **SQL/2003** Feature F302.

Example

For examples of INTERSECT usage, see "Set operators and NULL" [*SQL Anywhere Server - SQL Usage*].

LEAVE statement

Use this statement to leave a compound statement or loop.

Syntax

LEAVE *statement-label*

Remarks

The LEAVE statement is a control statement that allows you to leave a labeled compound statement or a labeled loop. Execution resumes at the first statement after the compound statement or loop.

The compound statement that is the body of a procedure or trigger has an implicit label that is the same as the name of the procedure or trigger.

Permissions

None.

Side effects

None.

See also

- "LOOP statement" on page 672
- "FOR statement" on page 608
- "BEGIN statement" on page 413
- "Using procedures, triggers, and batches" [*SQL Anywhere Server - SQL Usage*]

Standards and compatibility

- **SQL/2003** Persistent Stored Module feature.

Example

The following fragment shows how the LEAVE statement is used to leave a loop.

```
SET i = 1;
lbl:
LOOP
   INSERT
   INTO Counters ( number )
   VALUES ( i );
   IF i >= 10 THEN
      LEAVE lbl;
   END IF;
   SET i = i + 1
END LOOP lbl
```

The following example fragment uses LEAVE in a nested loop.

```
outer_loop:
LOOP
   SET i = 1;
   inner_loop:
   LOOP
      ...
      SET i = i + 1;
      IF i >= 10 THEN
         LEAVE outer_loop
      END IF
   END LOOP inner_loop
END LOOP outer_loop
```

LOAD STATISTICS statement

For internal use only. This statement loads statistics into the ISYSCOLSTAT system table. It is used by the dbunload utility to unload column statistics from the old database. It should not be used manually.

Syntax

LOAD STATISTICS [[*owner*.]*table-name*.]*column-name*
format-id, *density*, *max-steps*, *actual-steps*, *step-values*, *frequencies*

Parameters

- **format-id** Internal field used to determine the format of the rest of the row in the ISYSCOLSTAT system table.
- **density** An estimate of the weighted average selectivity of a single value for the column, not counting the selectivity of large single value selectivities stored in the row.
- **max-steps** The maximum number of steps allowed in the histogram.
- **actual-steps** The number of steps actually used at this time.
- **step-values** Boundary values of the histogram steps.
- **frequencies** Selectivities of histogram steps.

Permissions

Must have DBA authority.

Side effects

None.

See also

- "ISYSCOLSTAT system table" on page 801
- "Unload utility (dbunload)" [*SQL Anywhere Server - Database Administration*]

Standards and compatibility

- **SQL/2003** Vendor extension.

LOAD TABLE statement

Use this statement to import bulk data into a database table from an external file.

Syntax

LOAD [**INTO**] **TABLE** [*owner*.]*table-name*
[**(** *column-name*, ... **)**]
load-source
[*load-option* ...]
[*statistics-limitation-option*]

load-source :
{ **FROM** *filename-expression*
  | **USING FILE** *filename-expression*
  | **USING CLIENT FILE** *client-filename-expression*
  | **USING VALUE** *value-expression*
  | **USING COLUMN** *column-expression* }

filename-expression : string | variable

client-filename-expression : string | variable

value-expression : expression

column-expression :
column-name
  FROM *table-name*
  ORDER BY *column-list*

load-option :
BYTE ORDER MARK { **ON** | **OFF** }
| **CHECK CONSTRAINTS** { **ON** | **OFF** }
| { **COMPRESSED** | **AUTO COMPRESSED** | **NOT COMPRESSED** }
| { **ENCRYPTED KEY '***key***'** | **NOT ENCRYPTED** }
| **COMMENTS INTRODUCED BY** *comment-prefix*
| **COMPUTES** { **ON** | **OFF** }
| **DEFAULTS** { **ON** | **OFF** }
| **DELIMITED BY** *string*
| **ENCODING** *encoding*
| **ESCAPE CHARACTER** *character*
| **ESCAPES** { **ON** | **OFF** }
| **FORMAT** { **TEXT** | **BCP** }
| **HEXADECIMAL** { **ON** | **OFF** }
| **ORDER** { **ON** | **OFF** }
| **PCTFREE** *percent-free-space*
| **QUOTE** *string*
| **QUOTES** { **ON** | **OFF** }
| **ROW DELIMITED BY** *string*
| **SKIP** *integer*
| **STRIP** { **ON** | **OFF** | **LTRIM** | **RTRIM** | **BOTH** }
| **WITH CHECKPOINT** { **ON** | **OFF** }
| **WITH** { **FILE NAME** | **ROW** | **CONTENT** } **LOGGING**

statistics-limitation-option :
STATISTICS {
  ON [**ALL COLUMNS**]
  | **ON KEY COLUMNS**
  | **ON (** *column-list* **)**
  | **OFF**
  }

comment-prefix : string

encoding : string

Parameters

- **column-name** Use this clause to specify one or more columns to load data into. Any columns not present in the column list become NULL if DEFAULTS is OFF. If DEFAULTS is ON and the column has a default value, that value is used. If DEFAULTS is OFF and a non-nullable column is omitted from the column list, the database server attempts to convert the empty string to the column's type.

  When a column list is specified, it lists the columns that are expected to exist in the file and the order in which they are expected to appear. Column names cannot be repeated. Column names that do not appear in the list are set to NULL/zero/empty or DEFAULT (depending on column nullability, data type, and the DEFAULTS setting). Columns that exist in the input file that are to be ignored by LOAD TABLE can be specified using **filler()** as a column name.

- **load-source** Use this clause to specify the data source to load data from. There are several sources of data from which data can be loaded. The following list gives the supported load sources:
- **FROM clause** Use this to specify a file. The *filename-expression* is passed to the database server as a string. The string is therefore subject to the same database formatting requirements as other SQL strings. In particular:
  - To indicate directory paths, the backslash character (\) must be represented by two backslashes. The statement to load data from the file *c:\temp\input.dat* into the Employees table is:

    ```
    LOAD TABLE Employees
    FROM 'c:\\temp\\input.dat' ...
    ```

  - The path name is relative to the database server, not to the client application.
  - You can use UNC path names to load data from files on computers other than the database server.
- **USING FILE clause** Use this clause to load data from a file. This is synonymous with specifying the FROM *filename* clause.
- **USING CLIENT FILE clause** Use this clause to load data from a file on a client computer. When the database server retrieves data from *client-filename-expression*, the data is not materialized in the server's memory, so the database server limit on the size of BLOB expressions does not apply to the file. Therefore, the client file can be of an arbitrary size.
- **USING VALUE clause** Use this clause to load data from any expression of CHAR, NCHAR, BINARY, or LONG BINARY type, or BLOB string. The following are examples of how this clause can be used:
  - The following syntax uses the xp_read_file system procedure to get the values to load from the target file:

    ```
    ... USING VALUE xp_read_file( 'filename' )...
    ```

  - The following syntax specifies the value directly, inserting two rows with values of 4 and 5, respectively;

    ```
    ... USING VALUE '4\n5'...
    ```

  - The following syntax uses the results of the READ_CLIENT_FILE function as the value:

    ```
    ... USING VALUE READ_CLIENT_FILE( client-filename-expression )
    ```

    In this case, you can also specify `USING CLIENT FILE` *`client-filename-expression`* since they are semantically equivalent.

  If the ENCODING clause is not specified in the LOAD TABLE statement, then encoding for values is assumed to be in the database character set (db_charset) if the values are of type CHAR or BINARY, and NCHAR database character set (nchar_charset) if the values are of type NCHAR.
- **USING COLUMN clause** Use this clause to load data from a single column in another table. This clause is used by the database server when it replays the transaction log during recovery by replaying the LOAD TABLE ... WITH CONTENT LOGGING statements. Transaction log records for LOAD

TABLE ... WITH CONTENT LOGGING statements comprise chunks of concatenated rows. When the database server encounters these chunks in the transaction log during recovery, it loads the chunks into a temporary table and then loads all the data from the original load operation.

The following clauses are supported in the USING COLUMN clause:

- **table-name** The name of the base or temporary table that contains the column to load data from. When used by the database server during recovery from the transaction log, this is the table that holds the chunks of rows to be parsed and loaded.
- **column-name** The name of the column in *table-name* that holds the chunks of rows to be loaded.
- **column-list** One or more columns in the destination table used to sort the rows prior to loading the data.
- **load-option clause** There are several load options you can specify to control how data is loaded. The following list gives the supported load options:
  - **BYTE ORDER MARK clause** Use this clause to specify whether a byte order mark (BOM) is present in the encoding. By default, this option is ON, which enables the server to search for and interpret a byte order mark (BOM) at the beginning of the data. If BYTE ORDER MARK is OFF, the server does not search for a BOM.

    If the ENCODING clause is specified:

    - If the BYTE ORDER MARK option is ON and you specify a UTF-16 encoding with an endian such as UTF-16BE or UTF-16LE, the database server searches for a BOM at the beginning of the data. If a BOM is present, it is used to verify the endianness of the data. If you specify the wrong endian, an error is returned.
    - If the BYTE ORDER MARK option is ON and you specify a UTF-16 encoding without an explicit endian, the database server searches for a BOM at the beginning of the data. If a BOM is present, it is used to determine the endianness of the data. Otherwise, the operating system endianness is assumed.
    - If the BYTE ORDER MARK option is ON and you specify a UTF-8 encoding, the database server searches for a BOM at the beginning of the data. If a BOM is present it is ignored.

    If the ENCODING clause is not specified:

    - If you do not specify an ENCODING clause and the BYTE ORDER MARK option is ON, the server looks for a BOM at the beginning of the input data. If a BOM is located, the source encoding is automatically selected based on the encoding of the BOM (UTF-16BE, UTF-16LE, or UTF-8) and the BOM is not considered to be part of the data to be loaded.
    - If you do not specify an ENCODING clause and the BYTE ORDER MARK option is OFF, or a BOM is not found at the beginning of the input data, the database CHAR encoding is used.
  - **CHECK CONSTRAINTS clause** Use this clause to control whether constraints are checked during loading. CHECK CONSTRAINTS is ON by default, but the Unload utility (dbunload) writes out LOAD TABLE statements with CHECK CONSTRAINTS set to OFF. Setting CHECK CONSTRAINTS to OFF disables check constraints, which can be useful, for example, during database rebuilding. If a table has check constraints that call user-defined functions that are not yet created, the rebuild fails unless CHECK CONSTRAINTS is set to OFF.

- **COMMENTS INTRODUCED BY clause** Use this clause to specify the string used in the data file to introduce a comment. When used, LOAD TABLE ignores any line that begins with the string *comment-prefix*. For example, in the following statement, lines in *input.dat* that start with // are ignored.

  ```
  LOAD TABLE Employees FROM 'c:\\temp\\input.dat' COMMENTS INTRODUCED
  BY '//' ...
  ```

  Comments are only allowed at the beginning of a new line.

  If COMMENTS INTRODUCED BY is omitted, the data file must not contain comments because they are interpreted as data.

- **COMPRESSED clause** Specify COMPRESSED if the data being loaded is compressed in the input file. The database server decompresses the data before loading it. If you specify COMPRESSED and the data is not compressed, the LOAD fails and returns an error.

  Specify AUTO COMPRESSED to allow the database server determine whether the data in the input file is compressed. If so, the database server decompresses the data before loading it.

  Specify NOT COMPRESSED to indicate that the data in the input file is not compressed. You can also specify NOT COMPRESSED if the data is compressed, but you don't want the database server to decompress it. In this case, the data remains compressed in the database. However, if a file is both encrypted and compressed, you cannot use NOT ENCRYPTED without also using NOT COMPRESSED.

- **COMPUTES clause** By default, this option is ON, which enables recalculation of computed columns. Setting COMPUTES to OFF disables computed column recalculations. COMPUTES OFF is useful, for example, if you are rebuilding a database, and a table has a computed column that calls a user-defined function that is not yet created. The rebuild would fail unless this option was set to OFF.

  The Unload utility (dbunload) writes out LOAD TABLE statements with the COMPUTES set to OFF.

- **DEFAULTS clause** By default, DEFAULTS is set to OFF. If DEFAULTS is OFF, any column not present in the list of columns is assigned NULL. If DEFAULTS is set to OFF and a non-nullable column is omitted from the list, the database server attempts to convert the empty string to the column's type. If DEFAULTS is set to ON and the column has a default value, that value is used.

- **DELIMITED BY clause** Use this clause to specify the column delimiter string. The default column delimiter string is a comma; however, it can be any string up to 255 bytes in length (for example, `... DELIMITED BY '###' ...`). The delimiter you specify is a string and should be quoted. If you want to specify tab-delimited values, you could specify the hexadecimal escape sequence for the tab character (9), `... DELIMITED BY '\x09' ...`.

- **ENCODING clause** Use this clause to specify the character encoding used for the data being loaded into the database. The ENCODING clause can only be used with the TEXT format.

  If a translation error occurs during the load operation, it is reported based on the setting of the on_charset_conversion_failure option. See "on_charset_conversion_failure option [database]" [*SQL Anywhere Server - Database Administration*].

For more information about how to obtain the list of supported SQL Anywhere encodings, see "Supported character sets" [*SQL Anywhere Server - Database Administration*].

Specify the BYTE ORDER clause to include a byte order mark in the data.

- **ENCRYPTED clause** Use this clause to specify encryption settings. When loading encrypted data, specify ENCRYPTED KEY followed by the key used to encrypt the data in the input file.

  Specify NOT ENCRYPTED to indicate that the data in the input file is not encrypted. You can also specify NOT ENCRYPTED if the data is encrypted, but you don't want the database server to decompress it. In this case, the data remains compressed in the database. However, if a file is both encrypted and compressed, you cannot use NOT ENCRYPTED without also using NOT COMPRESSED.

- **ESCAPE CHARACTER clause** Use this clause to specify the escape character used in the data. The default escape character for characters stored as hexadecimal codes and symbols is a backslash (\), so \x0A is the linefeed character, for example. This can be changed using the ESCAPE CHARACTER clause. For example, to use the exclamation mark as the escape character, you would enter:

  ```
  ESCAPE CHARACTER '!'
  ```

  Only one single-byte character can be used as an escape character.

- **ESCAPES clause** Use this clause to control whether to recognize escape characters. With ESCAPES turned ON (the default), characters following the backslash character are recognized and interpreted as special characters by the database server. Newline characters can be included as the combination \n, and other characters can be included in data as hexadecimal ASCII codes, such as \x09 for the tab character. A sequence of two backslash characters (\\ ) is interpreted as a single backslash. A backslash followed by any character other than n, x, X, or \ is interpreted as two separate characters. For example, \q inserts a backslash and the letter q.

- **FORMAT clause** Use this clause to specify the format of the data source you are loading data from. If you choose TEXT, input lines are assumed to be characters (as defined by the ENCODING option), one row per line, with values separated by the column delimiter string. Choosing BCP allows the import of Adaptive Server Enterprise-generated BCP out files containing BLOBs.

- **HEXADECIMAL clause** Use this clause to specify whether to read binary values as hexadecimals. By default, HEXADECIMAL is ON. With HEXADECIMAL ON, binary column values are read as **0x***nnnnnn*..., where 0x is a zero followed by an x, and each *n* is a hexadecimal digit. It is important to use HEXADECIMAL ON when dealing with multibyte character sets.

  The HEXADECIMAL clause can be used only with the FORMAT TEXT clause.

- **ORDER clause** Use this clause to specify the order to sort the data into when loading. The default for ORDER is ON. If ORDER is ON, and a clustered index has been declared, then LOAD TABLE sorts the input data according to the clustered index and inserts rows in the same order. If the data you are loading is already sorted, you should set ORDER to OFF. See "Using clustered indexes" [*SQL Anywhere Server - SQL Usage*].

- **PCTFREE clause** Use this clause to specify the percentage of free space you want to reserve for each table page. This setting overrides any permanent setting for the table, but only for the duration of the load, and only for the data being loaded. The value *percent-free-space* is an integer between 0 and 100. A value of 0 specifies that no free space is to be left on each page—each page is to be fully packed. A high value causes each row to be inserted into a page by itself. For more information about PCTFREE, see "CREATE TABLE statement" on page 523.

- **QUOTE clause** The QUOTE clause is for TEXT data only; the *string* is placed around string values. The default is a single quote (apostrophe).
- **QUOTES clause** Use this clause to specify whether strings are enclosed in quotes. When QUOTES is set to ON (the default), the LOAD TABLE statement expects strings to be enclosed in quote characters. The quote character is either an apostrophe (single quote) or a quotation mark (double quote). The first such character encountered in a string is treated as the quote character for the string. Strings must be terminated by a matching quote.

  When QUOTES is set to ON, column delimiter strings can be included in column values. Also, quote characters are assumed not to be part of the value. Therefore, the following line is treated as two values, not three, despite the presence of the comma in the address. Also, the quotes surrounding the address are not inserted into the database.

  ```
  '123 High Street, Anytown',(715)398-2354
  ```

  To include a quote character in a value, when QUOTES is set to ON, you must use two quotes. The following line includes a value in the third column that is a single quote character:

  ```
  '123 High Street, Anytown','(715)398-2354',''''
  ```

- **ROW DELIMITED BY clause** Use this clause to specify the string that indicates the end of an input record. The default delimiter string is a newline (\n); however, it can be any string up to 255 bytes in length (for example, `ROW DELIMITED BY '###'`). The same formatting requirements apply to other SQL strings. If you wanted to specify tab-delimited values, you could specify the hexadecimal escape sequence for the tab character (9), `ROW DELIMITED BY '\x09'`. If your delimiter string contains a \n, it matches either \r\n or \n.
- **SKIP clause** Use this clause to specify whether to ignore lines at the beginning of a file. The *integer* argument specifies the number of lines to skip. You can use this clause to skip over a line containing column headings, for example. If the row delimiter is not the default (newline), then skipping may not work correctly if the data contains the row delimiter embedded within a quoted string.
- **STRIP clause** Use this clause to specify whether unquoted values should have leading or trailing blanks stripped off before they are inserted. The STRIP option accepts the following options:
  - **STRIP OFF** Do not strip off leading or trailing blanks.
  - **STRIP LTRIM** Strip leading blanks.
  - **STRIP RTRIM** Strip trailing blanks.
  - **STRIP BOTH** Strip both leading and trailing blanks.
  - **STRIP ON** Deprecated. Equivalent to STRIP RTRIM.
- **WITH CHECKPOINT clause** Use this clause to specify whether to perform a checkpoint. The default setting is OFF. If this clause is set to ON, a checkpoint is issued after successfully completing and logging the statement. If this clause is set to ON, and the database requires

automatic recovery before a CHECKPOINT is issued, the data file used to load the table must be present for the recovery to complete successfully. If WITH CHECKPOINT ON is specified, and recovery is subsequently required, recovery begins after the checkpoint, and the data file need not be present.

The data files are required, regardless of what is specified for this clause, if the database becomes corrupt and you need to use a backup and apply the current log file.

Caution
If you set the database option conversion_error to Off, you may load bad data into your table without any error being reported. If you do not specify WITH CHECKPOINT ON, and the database needs to be recovered, the recovery may fail as conversion_error is On (the default value) during recovery. It is recommended that you do not load tables when conversion_error is set to Off and WITH CHECKPOINT ON is not specified.

For more information about the conversion_error option, see "conversion_error option [compatibility]" [*SQL Anywhere Server - Database Administration*].

- **WITH { FILE NAME | ROW | CONTENT } LOGGING** Use this clause to control the level of detail logged in the transaction log during a load operation. The levels of logging are as follows:
  - **WITH FILE NAME LOGGING clause** The WITH FILE NAME LOGGING clause causes only the LOAD TABLE statement to be recorded in the transaction log. To guarantee consistent results when the transaction log is used during recovery, the file used for the original load operation must be present in its original location, and must contain the original data. This level of logging does not impact performance; however, you should not use it if your database is involved in mirroring or synchronization. Also, this level can not be used when loading from an expression or a client file.

    When you do not specify a logging level in the LOAD TABLE statement, WITH FILE NAME LOGGING is the default level when specifying:

    - FROM *filename-expression*
    - USING FILE *filename-expression*
  - **WITH ROW LOGGING clause** The WITH ROW LOGGING clause causes each row that is loaded to be recorded in the transaction log as an INSERT statement. This level of logging is recommended for databases involved in synchronization, and is supported in database mirroring. However, when loading large amounts of data, this logging type can impact performance, and results in a much longer transaction log.

    This level is also ideal for databases where the table being loaded into contains non-deterministic values, such as computed columns, or CURRENT TIMESTAMP defaults.
  - **WITH CONTENT LOGGING clause** The WITH CONTENT LOGGING clause causes the database server to chunk together the content of the rows that are being loaded. These chunks can be reconstituted into rows later, for example during recovery from the transaction log. When loading large amounts of data, this logging type has a very low impact on performance, and offers increased data protection, but it does result in a longer transaction log. This level of logging is recommended for databases involved in mirroring, or where it is desirable to not maintain the original data files for later recovery.

The WITH CONTENT LOGGING clause cannot be used if the database is involved in synchronization.

When you do not specify a logging level in the LOAD TABLE statement, WITH CONTENT LOGGING is the default level when specifying:

- USING CLIENT FILE *client-filename-expression*
- USING VALUE *value-expression*
- USING COLUMN *column-expression*

- **statistics-limitation-option** Allows you to limit the columns for which statistics are generated during the execution of LOAD TABLE. Otherwise, statistics are generated for all columns. You should only use this clause if you are certain that statistics will not be used on some columns. You can specify ON ALL COLUMNS (the default), OFF, ON KEY COLUMNS, or a list of columns for which statistics should be generated.

Remarks

LOAD TABLE allows efficient mass insertion into a database table from a file. LOAD TABLE is more efficient than the Interactive SQL statement INPUT.

LOAD TABLE places a write lock on the whole table. For base tables and global temporary tables, a commit is performed. For local temporary tables, a commit is not performed

If you attempt to use LOAD TABLE on a table on which an immediate text index is built, or that is referenced by an immediate view, the load fails. This does not occur for non-immediate text indexes or materialized views; however, it is strongly recommended that you truncate the data in dependent indexes and materialized views before executing the LOAD TABLE statement, and then refresh the indexes and views after. See “TRUNCATE statement” on page 768, and “TRUNCATE TEXT INDEX statement” on page 770.

Do not use the LOAD TABLE statement on a temporary table for which ON COMMIT DELETE ROWS was specified, either explicitly or by default, at creation time. However, you *can* use LOAD TABLE if ON COMMIT PRESERVE ROWS or NOT TRANSACTIONAL was specified.

With FORMAT TEXT, a NULL value is indicated by specifying no value. For example, if three values are expected and the file contains `1,,'Fred',`, then the values inserted are 1, NULL, and Fred. If the file contains `1,2,`, then the values 1, 2, and NULL are inserted. Values that consist only of spaces are also considered NULL values. For example, if the file contains `1,  ,'Fred',`, then values 1, NULL, and Fred are inserted. All other values are considered not NULL. For example, '' (single-quote single-quote) is an empty string. 'NULL' is a string containing four letters.

If a column being loaded by LOAD TABLE does not allow NULL values and the file value is NULL, then numeric columns are given the value 0 (zero), character columns are given an empty string (''). If a column being loaded by LOAD TABLE allows NULL values and the file value is NULL, then the column value is NULL (for all types).

If the table contains columns a, b, and c, and the input data contains a, b, and c, but the LOAD statement only specifies only a and b as columns to load data into, the following values are inserted into column c:

- if DEFAULTS ON is specified, and column c has a default value, the default value is used.
- if column c does not have a default defined for it and it allows NULLs, then a NULL is used.

- if column c does not have a default defined for it and it does not allow NULLs, then either a zero (0) or an empty string ('') is used, or an error is returned, depending on the data type of the column.

LOAD TABLE and column statistics To create histograms on table columns, LOAD TABLE captures column statistics when it loads data. The histograms are used by the optimizer. For more information about how column statistics are used by the optimizer, see "Optimizer estimates and column statistics" [*SQL Anywhere Server - SQL Usage*].

Following are additional tips about loading and column statistics:

- LOAD TABLE saves statistics on base tables for future use. It does not save statistics on global temporary tables.
- If you are loading into an empty table that may have previously contained data, it may be beneficial to drop the statistics for the column before executing the LOAD TABLE statement. See "DROP STATISTICS statement" on page 583.
- If column statistics exist when LOAD TABLE is performed on a column, statistics for the column are *not* recalculated. Instead, statistics for the new data are inserted into the existing statistics. This means that if the existing column statistics are out-of-date, they will still be out of date after loading new data into the column. If you suspect that the column statistics are out of date, you should consider updating them either before, or after, executing the LOAD TABLE statement. See "Updating column statistics to improve optimizer performance" [*SQL Anywhere Server - SQL Usage*].
- LOAD TABLE adds statistics only if the table has five or more rows. If the table has at least five rows, histograms are modified as follows:

Data already in table?	Histogram present?	Action taken
Yes	Yes	Integrate changes into the existing histograms
Yes	No	Do not build histograms
No	Yes	Integrate changes into the existing histograms
No	No	Build new histograms

- LOAD TABLE does not generate statistics for columns that contain NULL values for more than 90% of the rows being loaded.

Using dynamically constructed file names You can execute a LOAD TABLE statement with a dynamically constructed file name by assigning the file name to a variable and using the variable name in the LOAD TABLE statement.

Permissions

The permissions required to execute a LOAD TABLE statement depend on the database server -gl option, as follows:

- If the -gl option is set to ALL, you must be the owner of the table or have DBA authority or have ALTER privileges.

- If the -gl option is set to DBA, you must have DBA authority.
- If the -gl option is set to NONE, LOAD TABLE is not permitted.

See “-gl server option” [*SQL Anywhere Server - Database Administration*].

Requires an exclusive lock on the table.

When reading from a file on a client computer:

- READCLIENTFILE authority is required. See “READCLIENTFILE authority” [*SQL Anywhere Server - Database Administration*].
- Read permissions are required on the directory being read from.
- The allow_read_client_file database option must be enabled. See “allow_read_client_file option [database]” [*SQL Anywhere Server - Database Administration*].
- The read_client_file secured feature must be enabled. See “-sf server option” [*SQL Anywhere Server - Database Administration*].

Side effects

Automatic commit.

Inserts are not recorded in the log file unless WITH ROW LOGGING clause is specified. So, the inserted rows may not be recovered in the event of a failure depending upon the logging type. In addition, the LOAD TABLE statement without the WITH ROW LOGGING clause should never be used in databases used as MobiLink clients, or in a database involved in SQL Remote replication, because these technologies replicate changes through analysis of the log file.

The LOAD TABLE statement does not fire any triggers associated with the table.

A checkpoint is carried out at the beginning of the operation. A second checkpoint is performed at the end if WITH CHECKPOINT ON is specified.

Column statistics are updated if a significant amount of data is loaded.

See also

- “Importing data” [*SQL Anywhere Server - SQL Usage*]
- “UNLOAD statement” on page 772
- “INSERT statement” on page 649
- “INPUT statement [Interactive SQL]” on page 644
- “Accessing data on client computers” [*SQL Anywhere Server - SQL Usage*]
- “Importing and exporting data” [*SQL Anywhere Server - SQL Usage*]

Standards and compatibility

- **SQL/2003**   Vendor extension.

Example

Following is an example of LOAD TABLE. First, you create a table, and then load data into it using a file called *input.txt*.

```
CREATE TABLE t( a CHAR(100), let_me_default INT DEFAULT 1, c CHAR(100) );
```

Following is the content of a file called *input.txt*:

```
ignore_me, this_is_for_column_c, this_is_for_column_a
```

The following LOAD statement loads the file called *input.txt*:

```
LOAD TABLE T ( filler(), c, a ) FROM 'input.txt' FORMAT TEXT DEFAULTS ON;
```

The command `SELECT * FROM t` yields the result set:

a	let_me_default	c
this_is_for_column_a	1	this_is_for_column_c

The following example executes the LOAD TABLE statement with a dynamically-constructed file name, via the EXECUTE IMMEDIATE statement:

```
CREATE PROCEDURE LoadData( IN from_file LONG VARCHAR )
BEGIN
    DECLARE path LONG VARCHAR;
    SET path = 'd:\\data\\' || from_file;
    LOAD MyTable FROM path;
END;
```

The following example loads UTF-8-encoded table data into mytable:

```
LOAD TABLE mytable FROM 'mytable_data_in_utf8.dat' ENCODING 'UTF-8';
```

LOCK TABLE statement

Use this statement to prevent other concurrent transactions from accessing or modifying a table.

Syntax

LOCK TABLE *table-name*
[**WITH HOLD**]
IN { **SHARE** | **EXCLUSIVE** } **MODE**

Parameters

- **table-name** The name of the table. The table must be a base table, not a view. As temporary table data is local to the current connection, locking global or local temporary tables has no effect.
- **WITH HOLD clause** Specify this clause to lock the table until the end of the connection. If the clause is not specified, the lock is released when the current transaction is committed or rolled back.
- **SHARE MODE clause** Specify this clause to obtain a shared table lock on the table, preventing other transactions from modifying the table but allowing them read access. If a transaction puts a shared lock on a table, it can change data in the table provided no other transaction holds a lock of any kind on the row(s) being modified.
- **EXCLUSIVE MODE clause** Specify this clause to obtain an exclusive table lock on the table, preventing other transactions from accessing the table. No other transaction can execute queries, updates,

or any other action against the table. If a table is locked exclusively with a statement such as `LOCK TABLE...IN EXCLUSIVE MODE`, the default behavior is to not acquire row locks for the table. This behavior can be disabled by setting the subsume_row_locks option to Off.

Remarks

The LOCK TABLE statement allows direct control over concurrency at a table level, independent of the current isolation level.

While the isolation level of a transaction generally governs the kinds of locks that are set when the current transaction executes a request, the LOCK TABLE statement allows more explicit control locking of the rows in a table.

You cannot execute the LOCK TABLE statement against a view. However, if you execute the LOCK TABLE statement against a base table, a shared schema lock is created, which locks dependent views. A shared schema lock persists until the transaction is committed or rolled back.

Permissions

To lock a table in SHARE mode, SELECT privileges are required.

To lock a table in EXCLUSIVE mode, you must be the table owner or have DBA authority.

Side effects

Other transactions that require access to the locked table may be delayed or blocked.

See also

- "Table locks" [*SQL Anywhere Server - SQL Usage*]
- "SELECT statement" on page 728
- "sa_locks system procedure" on page 893

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement prevents other transactions from modifying the Customers table for the duration of the current transaction:

```
LOCK TABLE Customers
IN SHARE MODE;
```

LOOP statement

Use this statement to repeat the execution of a statement list.

Syntax

[*statement-label* :]
[**WHILE** *search-condition*] **LOOP**

statement-list
END LOOP [*statement-label*]

Remarks

The WHILE and LOOP statements are control statements that allow you to execute a list of SQL statements repeatedly while a *search-condition* evaluates to TRUE. The LEAVE statement can be used to resume execution at the first statement after the END LOOP.

If the ending *statement-label* is specified, it must match the beginning *statement-label*.

Permissions

None.

Side effects

None.

See also

- "FOR statement" on page 608
- "CONTINUE statement [T-SQL]" on page 431

Standards and compatibility

- **SQL/2003** Persistent Stored Module feature.

Example

A While loop in a procedure.

```
...
SET i = 1;
WHILE i <= 10 LOOP
   INSERT INTO Counters( number ) VALUES ( i );
   SET i = i + 1;
END LOOP;
...
```

A labeled loop in a procedure.

```
SET i = 1;
lbl:
LOOP
   INSERT
   INTO Counters( number )
   VALUES ( i );
   IF i >= 10 THEN
      LEAVE lbl;
   END IF;
   SET i = i + 1;
END LOOP lbl
```

MERGE statement

Use this statement to merge tables, views, and system procedure results into a table or view.

Syntax

MERGE
INTO *target-object* [*into-column-list*]
USING [**WITH AUTO NAME**] *source-object*
  ON *merge-search-condition*
merge-operation […]
 [**OPTION (** *query-hint***,** ... **)**]

target-object:
[*userid.*]*target-table-name* [[**AS**] *target-correlation-name*]
| [*userid.*]*target-view-name* [[**AS**] *target-correlation-name*]
| **(** *select-statement* **)** [**AS**] *target-correlation-name*

source-object :
[*userid.*]*source-table-name* [[**AS**] *source-correlation-name*] [**WITH (** *table-hints* **)**]
| [*userid.*]*source-view-name* [[**AS**] *source-correlation-name*]
| [*userid.*]*source-mat-view-name* [[**AS**] *source-correlation-name*]
| **(** *select-statement* **)** [**AS**] *source-correlation-name* [*using-column-list*]
| *procedure*

procedure :
[*owner.*]*procedure-name* **(** *procedure-syntax* **)**
  [**WITH (** *column-name data-type***,** ... **)**]
  [[**AS**] *source-correlation-name*]

merge-search-condition :
search-condition
| **PRIMARY KEY**

merge-operation :
WHEN MATCHED [**AND** *search-condition*] **THEN** *match-action*
| **WHEN NOT MATCHED** [**AND** *search-condition*] **THEN** *not-match-action*

match-action :
DELETE
| **RAISERROR** [*error-number*]
| **SKIP**
| **UPDATE SET** *set-item***,** ...
| **UPDATE** [**DEFAULTS** { **ON** | **OFF** }]

not-match-action :
INSERT
| **INSERT** [*insert-column-list*] **VALUES (** *value***,** ... **)**
| **RAISERROR** [*error-number*]
| **SKIP**

set-item :
[*target-correlation-name.*]*column-name* **=** { *expression* | **DEFAULT** }
| [*owner-name.*]*target-table-name***.***column-name* **=** { *expression* | **DEFAULT** }

insert-column-list :
(*column-name***,** ... **)**

query-hint :
MATERIALIZED VIEW OPTIMIZATION *option-value*

| **FORCE OPTIMIZATION**
| *option-name* **=** *option-value*

into-column-list :
(*column-name***,** ... **)**

using-column-list :
(*column-name***,** ... **)**

error-number : positive integer or variable greater than 17000

option-name : identifier

option-value : hostvar (indicator allowed), string, identifier, or number

table-hints : see "FROM clause" on page 613

search-condition : see "Search conditions" on page 35

set-clause-list : see "SET statement" on page 735

Parameters

- **INTO clause** Use this clause to define the target object for the MERGE statement. *target-object* can be the name of a base table, regular view, or derived table; it cannot be a materialized view. The derived table or view must represent an updatable query block. For example, if the view or derived table definition contains UNION, INTERSECT, EXCEPT, or GROUP BY, then it cannot be used as a target object for the MERGE statement.

  When *target-object* is a derived table, the optional *into-column-list* can be used to provide alternate names for the columns of the derived table. When used in this manner, the size of the *into-column-list* must match the column list for the derived table, and the ordering of the two lists must be the same.

  When *target-object* is a base table or view, *into-column-list* can be used to specify a subset of the table or view columns as relevant for the rest of the MERGE statement.

  The database server uses *into-column-list* to resolve:

  - UPDATE without a SET clause in WHEN MATCHED clause
  - INSERT without a VALUES clause in a WHEN NOT MATCHED clause
  - PRIMARY KEY search condition in the ON clause
  - WITH AUTO NAME clause in the USING clause

  If you do not specify *into-column-list*, then *into-column-list* is assumed to contain all the columns of the *target-object*.

- **USING clause** Use this clause to define the source of the data you are merging from. *source-object* can be a base table (including table hints), a view, a materialized view, a derived table, or a procedure. If *source-object* is a derived table, you can specify *using-column-list*. All columns of *source-object* are used if you do not specify *using-column-list*.

- **WITH AUTO NAME clause** Use this clause to get the server to automatically use column names to match columns in the *into-column-list* columns in *target-object* for the merge operation. The following examples are equivalent and demonstrate that the order of the columns in *into-column-list* changes to match the names of the columns in the *source-object* when WITH AUTO Name is specified:

```
... INTO T ( Name, ID, Description )
   USING WITH AUTO NAME ( SELECT Description, Name, ID FROM PRODUCTS WHERE
Description LIKE '%cap%')
... INTO T ( Description, Name, ID )
   USING ( SELECT Description, Name, ID FROM PRODUCTS WHERE Description
LIKE '%cap%' )
```

- **ON clause** Use this clause to specify the condition to match a row in *source-object* with rows in *target-object*.

  For more information about search condition syntax, see "Search conditions" on page 35.

  You can specify ON PRIMARY KEY to match *source-object* rows based on the *target-object* primary key definition. *source-object* does not need a primary key. However, *target-object* must have a primary key. When specifying ON PRIMARY KEY:

  - An error is returned if *target-object* is not a base table, or if it does not have a primary key.
  - An error is returned if one or more primary key columns are not included in *into-column-list*.
  - The number of columns in *into-column-list* and *using-column-list* can be different as long as every primary key column in *into-column-list* has a corresponding matching column in *using-column-list*. For example, if *into-column-list* is (I1, I2, I3), *using-column-list* is (U1, U2), and the primary key columns are (I2, I3), an error is returned because column (I3) of the *target-object* primary key does not have a match in the *using-column-list*.
  - Regardless of the definition of the primary key, matching of primary key columns in *into-column-list* to expressions in *using-column-list* is based on the position of the primary key columns in *into-column-list*. For example, suppose the primary key on *target-object* is defined as (B, C), and the *into-column-list* is (E, C, F, A, D, B). When ON PRIMARY KEY is specified, *target-object* column B is compared to the sixth element of *using-column-list* because column B is in the sixth position in the *into-column-list*. Likewise, *target-object* column C is compared to the second element of *using-column-list*.

    ON PRIMARY KEY is syntactic shorthand for a corresponding ON condition. For example, assume that *into-column-list* is (I1, I2, .. I*n*), and that the corresponding matched *using-column-list* is (U1, U2, .. U*m*). Also assume that the primary key columns of *target-object* are I1, I2, I3 and all the primary key columns are contained in *into-column-list*. In this case, *merge-search-condition* is defined as the conjunct `"I1=U1 AND I2=U2 AND I3=U3"`.

- **WHEN MATCHED and WHEN NOT MATCHED clauses** Use the WHEN MATCHED and WHEN NOT MATCHED clauses to define an action to take when a row from *source-object* matches or does not match a row in *target-object*. You specify the action after the THEN keyword. You can control the actions to take for subsets of matching or non-matching rows by specifying an additional AND clause.

  The ON clause determines how rows from *source-object* are separated into matching and non-matching rows. A row in *source-object* is considered a matching row when the ON clause is TRUE for at least one row in *target-object*. A row from *source-object* is considered a non-matching row when the ON clause is not TRUE for any rows in *target-object*. Use multiple WHEN MATCHED and WHEN NOT MATCHED clauses to partition sets of matching and non-matching rows into disjoint subsets. Each subset is processed by a WHEN clause. WHEN MATCHED and WHEN NOT MATCHED clauses are processed in the order they appear in the MERGE statement.

The search condition specified in the AND clause of a WHEN MATCHED or WHEN NOT MATCHED clause determines if a candidate row is processed by the specific clause. When you specify a WHEN MATCHED or WHEN NOT MATCHED clause without the AND clause the search condition in the AND clause is assumed to be TRUE. If a row satisfies the AND condition for more than one clause, the row is processed by the clause that appears first in the MERGE statement.

An error is returned when any of the WHEN MATCHED clauses process the same *target-object* row more than once. A *target-object* row can belong to the same subset of the same WHEN MATCHED clause more than once if it matches two different input rows from the *source-object*.

In the following example an error is returned because the row with ID 300 from the *target-object* Products matches 111 rows from the *source-object* SalesOrderItems. All the matches belong to the same subset corresponding to the WHEN MATCHED THEN UPDATE clause.

```
MERGE INTO Products
   USING SalesOrderItems S
   ON S.ProductID = Products.ID
   WHEN MATCHED THEN UPDATE SET Products.Quantity = 20;
```

WHEN MATCHED: For a matching row, you can specify one of the following actions for *match-action*:

- **DELETE** Specify DELETE to delete the row from *target-object*.
- **RAISERROR** Specify RAISERROR to terminate the merge operation, roll back any changes, and return an error. By default, when you specify RAISERROR, the database server returns SQLSTATE 23510 and SQLCODE -1254. Optionally, you can customize the SQLCODE that is returned by specifying the *error-number* parameter after the RAISERROR keyword. The custom SQLCODE must be a positive integer greater than 17000, and can be specified either as a number or a variable. When you specify a custom SQLCODE, the number returned is a negative number.

  For example, if you specify `WHEN MATCHED AND` *`search-condition`* `THEN RAISERROR 17001`, then, when a row is found that satisfies the conditions of the WHEN clause, the merge operation fails, changes are rolled back, and the error returned has SQLSTATE 23510 and SQLCODE -17001. See "Using the RAISERROR action" [*SQL Anywhere Server - SQL Usage*].
- **SKIP** Specify SKIP to skip the row; no action is taken.
- **UPDATE** Specify UPDATE SET to update the row using the *set-item* values. *set-item* is a simple assignment expression where a column is set to the value of *expression*. There are no restrictions on the *expression*. You can also specify DEFAULT to set the column to the default defined for the column.

  For example, `UPDATE SET target-column1=DEFAULT, target-column2=source-column2` sets target-column1 to its default value and sets target-column2 to be the same as the modify row from source-column2 in *source-object*.

  If you do not specify the SET clause, the SET clause is defined by *into-column-list* and *using-column-list*. For example, if *into-column-list* is (I1, I2, .. I*n*), and *using-column-list* is (U1, U2, .. U*n*) the SET clause is assumed to be `"SET I1=U1 , I2=U2 , .. In=Un"`.

WHEN NOT MATCHED: For a non-matching row, you can specify one of the following actions for *non-match-action*:

- **INSERT** Specify INSERT ... VALUES to insert the row using the specified values. When you specify the INSERT clause without a VALUES clause, the VALUES clause is defined by *into-column-list* and *using-column-list*. For example, if *into-column-list* is (I1, I2, .. I*n*), and *using-column-list* is (U1, U2, .. U*n*), the INSERT without a VALUES clause is equivalent to `INSERT (I1, I2, .. In) VALUES (U1, U2, .. Un)`.
- **RAISERROR** Specify RAISERROR to terminate the merge operation, roll back any changes, and return an error. When you specify RAISERROR, the database server returns SQLSTATE 23510 and SQLCODE -1254 by default. Optionally, you can customize the SQLCODE that is returned by specifying the *error-number* parameter after the RAISERROR keyword. The custom SQLCODE must be a positive integer greater than 17000, and can be specified either as a number or a variable. When you specify a custom SQLCODE, the number returned is a negative number.

  For example, if you specify `WHEN NOT MATCHED AND search-condition THEN RAISERROR 17001`, then, when a row is found that satisfies the conditions of the WHEN clause, the merge operation fails, changes are rolled back, and the error returned has SQLSTATE 23510 and SQLCODE -17001. See "Using the RAISERROR action" [*SQL Anywhere Server - SQL Usage*].
- **SKIP** Specify SKIP to skip the row; no action is taken.

- **OPTION clause** Use this clause to specify hints for executing the statement. The following hints are supported:

  - MATERIALIZED VIEW OPTIMIZATION *option-value*
  - FORCE OPTIMIZATION
  - *option-name* = *option-value*

  For a description of these options, see the OPTIONS clause of the "SELECT statement" on page 728.

Remarks

Rows in *source-object* are compared to rows in *target-object* and found to be matching or non-matching depending on whether they satisfy the conditions of the ON clause. Rows in *source-object* are considered a match if there exists at least one row in *target-table* such that *merge-search-condition* evaluates to true. Matching rows and non-matching rows are then grouped by the actions defined for them in the WHEN MATCHED and WHEN NOT MATCHED clauses according to the search conditions specified by the AND clauses. The process of grouping rows by matched and non-matched actions is referred to as **branching**, and each group is referred to as a **branch**.

Once branching is complete, the database begins executing the action defined for the rows of the branch. Branches are processed in the order in which they occur, which matches the order in which the WHEN clauses occur in the statement. If, during branching, more than one row in *source-object* has an action defined for the same row in *target-object*, the merge operation fails and an error is returned. This prevents the merge operation from performing more than one action on any given row in *target-object*.

As branches are processed, the insert, update, and delete actions are recorded in the transaction log as their respective INSERT, UPDATE, and DELETE statements.

For information about how triggers can impact the merge operation, see "Import data with the MERGE statement" [*SQL Anywhere Server - SQL Usage*].

Permissions

DBA authority, or:

- INSERT, UPDATE, and DELETE permissions on *target-object* if the INSERT, UPDATE or DELETE action is specified in the MERGE statement.
- SELECT permission is required on any objects referenced in the MERGE statement.
- EXECUTE permission is required on any procedure referenced in the MERGE statement.

Side effects

Any triggers defined for *target-object* are fired.

See also

- "Import data with the MERGE statement" [*SQL Anywhere Server - SQL Usage*]
- "UPDATE statement" on page 777
- "INSERT statement" on page 649
- "DELETE statement" on page 558
- "SELECT statement" on page 728
- "Search conditions" on page 35

Standards and compatibility

- **SQL/2003** The MERGE statement is feature F312 of the SQL/2003 standard. The MERGE statement in SQL Anywhere is compliant with the MERGE statement specification in the SQL/2003 standard, with additional extensions. Some of these extensions to the MERGE statement are compliant with the forthcoming SQL/2008 standard. These include:
  - multiple WHEN MATCHED and WHEN NOT MATCHED clauses
  - [AND *search-condition*] with WHEN [NOT] MATCHED clauses
  - DELETE in a WHEN MATCHED clause
  - RAISERROR in a WHEN [NOT] MATCHED clause
  - SKIP in a WHEN [NOT] MATCHED clause
  - OPTION clause
  - PRIMARY KEY clause
  - DEFAULTS clause
  - INSERT clause without a VALUES clause
  - WITH AUTO NAME clause
  - UPDATE clause without the SET clause

Examples

The following example merges a row from a derived table into the Products table, effectively adding a new tee shirt with the same attributes as an existing tee shirt, but with a new color, quantity, and product identifier. In this example if the product with identification number 304 already exists in the Products table then the row is not inserted:

```
MERGE INTO Products ( ID, Name, Description, Size, Color, Quantity,
UnitPrice, Photo )
   USING WITH AUTO NAME (
      SELECT 304 AS ID,
             'Purple' AS Color,
             100 AS Quantity,
             Name,
             Description,
             Size,
             UnitPrice,
             Photo
          FROM Products WHERE Products.ID = 300 ) AS DT
   ON PRIMARY KEY
   WHEN NOT MATCHED THEN INSERT;
```

The following example is equivalent to the previous, but does not use syntactic shorthand:

```
MERGE INTO Products ( ID, Name, Description, Size, Color, Quantity,
UnitPrice, Photo )
   USING (
      SELECT 304 AS ID,
             'Purple' AS Color,
             100 AS Quantity,
             Name,
             Description,
             Size,
             UnitPrice,
             Photo
          FROM Products WHERE Products.ID = 300 )
          AS DT ( ID, Name, Description, Size, Color, Quantity, UnitPrice,
Photo )
   ON ( Products.ID = DT.ID )
   WHEN NOT MATCHED
      THEN INSERT ( ID, Name, Description, Size, Color, Quantity, UnitPrice,
Photo )
     VALUES ( DT.ID, DT.Name, DT.Description, DT.Size, DT.Color, DT.Quantity,
DT.UnitPrice, DT.Photo );
```

For a more detailed examples of the MERGE statement, see "Import data with the MERGE statement" [*SQL Anywhere Server - SQL Usage*].

MESSAGE statement

Use this statement to display a message.

Syntax

MESSAGE *expression*, ...
[**TYPE** { **INFO** | **ACTION** | **WARNING** | **STATUS** }]
[**TO** { **CONSOLE**
  | **CLIENT** [**FOR** { **CONNECTION** *conn-id* [**IMMEDIATE**] | **ALL** }]

```
| [ EVENT | SYSTEM ] LOG }
[ DEBUG ONLY ] ]
```

conn-id : integer

Parameters

- **TYPE clause** This clause specifies the message type. Acceptable values are INFO, ACTION, WARNING, and STATUS. The client application must decide how to handle the message. For example, Interactive SQL displays messages in the following locations:
  - **INFO** The Messages tab. INFO is the default type.
  - **ACTION** A window with an OK button.
  - **WARNING** A window with an OK button.
  - **STATUS** The Messages tab.
- **TO clause** This clause specifies the destination of a message:
  - **CONSOLE** Send messages to the database server messages window and the database server message log file if one has been specified. CONSOLE is the default.
  - **CLIENT** Send messages to the client application. Your application must decide how to handle the message, and you can use the TYPE as information on which to base that decision.
  - **LOG** Send messages to the server log file specified by the -o option. If EVENT or SYSTEM is specified, the message is also written to the database server messages window and to the Windows event log under event source SQLANY 11.0 Admin and to the Unix Syslog under the name SQLANY 11.0 Admin (servername). Messages in the server log are identified as follows:
    - **i** Messages of type INFO or STATUS.
    - **w** Messages of type WARNING.
    - **e** Messages of type ACTION.
- **FOR clause** For messages TO CLIENT, this clause specifies which connections receive notification about the message. By default, the connection receives the message the next time a SQL statement or a WAITFOR DELAY statement is executed.
  - **CONNECTION conn-id** Specify the recipient's connection ID. If IMMEDIATE is specified, the connection receives the message within a few seconds regardless of when the SQL statement is executed.
  - **ALL** Specify that all open connections receive the message.
- **DEBUG ONLY** This clause allows you to control whether debugging messages added to stored procedures and triggers are enabled or disabled by changing the setting of the debug_messages option. When DEBUG ONLY is specified, the MESSAGE statement is executed only when the debug_messages option is set to On.

Note
DEBUG ONLY messages are inexpensive when the debug_messages option is set to Off, so these statements can usually be left in stored procedures on a production system. However, they should be used sparingly in locations where they would be executed frequently; otherwise, they may result in a small performance penalty.

Remarks

The MESSAGE statement displays a message, which can be any expression. Clauses can specify the message type and where the message appears.

The procedure issuing a MESSAGE ... TO CLIENT statement must be associated with a connection.

For example, the window is not displayed in the following example because the event occurs outside a connection.

```
CREATE EVENT CheckIdleTime
TYPE ServerIdle
WHERE event_condition( 'IdleTime' ) > 100
HANDLER
BEGIN
   MESSAGE 'Idle engine' TYPE WARNING TO CLIENT;
END;
```

However, in the following example, the message is written to the database server messages window.

```
CREATE EVENT CheckIdleTime
TYPE ServerIdle
WHERE event_condition( 'IdleTime' ) > 100
HANDLER
BEGIN
   MESSAGE 'Idle engine' TYPE WARNING TO CONSOLE;
END;
```

Valid expressions can include a quoted string or other constant, variable, or function.

The FOR clause can be used to notify another application of an event detected on the database server without the need for the application to explicitly check for the event. When the FOR clause is used, recipients receive the message the next time that they execute a SQL statement. If the recipient is currently executing a SQL statement, the message is received when the statement completes. If the statement being executed is a stored procedure call, the message is received before the call is completed.

If an application requires notification within a short time after the message is sent and when the connection is not executing SQL statements, use the IMMEDIATE clause to implement client notification and not multiple concurrent WAITFOR DELAY statements.

Typically, messages sent using the IMMEDIATE clause are delivered in less than five seconds, even if the destination connection is not making database server requests. Message delivery could be delayed if the client connection makes several requests per second, receives very large BLOB data, or if the client's message callback executes for more than a second. In addition, sending more than one IMMEDIATE message to a single connection every two seconds could delay message delivery or generate an error message. If the client connection is disconnected, a successful MESSAGE ... IMMEDIATE statement may not be delivered.

Messages sent without the IMMEDIATE clause are only delivered when the client executes a specific request, or a WAITFOR DELAY statement. As a result, the delivery time of messages is unlimited.

The IMMEDIATE clause requires a SQL Anywhere version 11.0.1 or newer DBLib, ODBC or iAnywhere JDBC client library. The IMMEDIATE clause is not supported by non-threaded Unix client libraries. An error message is generated when a message is sent to a destination connection that does not support the IMMEDIATE clause.

A MESSAGE ... TO CLIENT expression can be truncated to 2048 bytes. For messages sent with the IMMEDIATE clause, the message expression can be truncated to the smaller of the packet size of the connection or 2048 bytes.

Embedded SQL and ODBC clients receive messages via message callback functions. In each case, these functions must be registered. In embedded SQL, the message callback is registered with db_register_a_callback using the DB_CALLBACK_MESSAGE parameter. In ODBC, the message callback is registered with SQLSetConnectAttr using the ASA_REGISTER_MESSAGE_CALLBACK parameter.

Permissions

DBA authority is required to execute a MESSAGE statement containing a FOR clause or a TO EVENT LOG or TO SYSTEM LOG clause.

Side effects

None.

See also

- "CREATE PROCEDURE statement (web services)" on page 495
- "debug_messages option [database]" [*SQL Anywhere Server - Database Administration*]
- "db_register_a_callback function" [*SQL Anywhere Server - Programming*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following procedure displays a message on the database server messages window:

```
CREATE PROCEDURE message_text()
BEGIN
MESSAGE 'The current date and time: ', Now();
END;
```

The following statement displays the string `The current date and time`, followed by the current date and time, in the database server messages window.

```
CALL message_text();
```

OPEN statement [ESQL] [SP]

Use this statement to open a previously declared cursor to access information from the database.

Syntax

OPEN *cursor-name*
[**USING** { **DESCRIPTOR** *sqlda-name* | *hostvar*, ... }]
[**WITH HOLD**]
[**ISOLATION LEVEL** *n*]
[**BLOCK** *n*]

cursor-name : *identifier* or *hostvar*

sqlda-name : *identifier*

Parameters

- **USING DESCRIPTOR clause** The USING DESCRIPTOR clause is for embedded SQL only. It specifies the host variables to be bound to the place-holder bind variables in the SELECT statement for which the cursor has been declared.
- **WITH HOLD clause** By default, all cursors are automatically closed at the end of the current transaction (COMMIT or ROLLBACK). The optional WITH HOLD clause keeps the cursor open for subsequent transactions. It will remain open until the end of the current connection or until an explicit CLOSE statement is executed. Cursors are automatically closed when a connection is terminated.
- **ISOLATION LEVEL clause** The ISOLATION LEVEL clause allows this cursor to be opened at an isolation level different from the current setting of the isolation_level option. All operations on this cursor are performed at the specified isolation level regardless of the option setting. If this clause is not specified, then the cursor's isolation level for the entire time the cursor is open is the value of the isolation_level option when the cursor is opened. See "How locking works" [*SQL Anywhere Server - SQL Usage*].

  The following values are supported:

  - 0
  - 1
  - 2
  - 3
  - snapshot
  - statement snapshot
  - readonly statement snapshot

  The cursor is positioned before the first row (see "Using cursors in embedded SQL" [*SQL Anywhere Server - Programming*] or "Using cursors in procedures and triggers" [*SQL Anywhere Server - SQL Usage*]).
- **BLOCK clause** This clause is for embedded SQL use only. Rows may be fetched by the client application more than one at a time. This is referred to as block fetching, prefetching, or multi-row fetching. The BLOCK clause can reduce the number of rows prefetched. Specifying the BLOCK clause on OPEN is the same as specifying the BLOCK clause on each FETCH. See "FETCH statement [ESQL] [SP]" on page 605.

Remarks

The OPEN statement opens the named cursor. The cursor must be previously declared.

When the cursor is on a CALL statement, OPEN causes the procedure to execute until the first result set (SELECT statement with no INTO clause) is encountered. If the procedure completes and no result set is found, the SQLSTATE_PROCEDURE_COMPLETE warning is set.

Embedded SQL usage: After successful execution of the OPEN statement, the *sqlerrd*[*3*] field of the SQLCA (SQLIOESTIMATE) is filled in with an estimate of the number of input/output operations required to fetch all rows of the query. Also, the *sqlerrd*[*2*] field of the SQLCA (SQLCOUNT) is filled with either the actual number of rows in the cursor (a value greater than or equal to 0), or an estimate thereof (a negative number whose absolute value is the estimate). It is the actual number of rows if the database server can compute it without counting the rows. The database can also be configured to always return the actual number of rows (see "row_counts option [database]" [*SQL Anywhere Server - Database Administration*]), but this can be expensive. If *cursor-name* is specified by an identifier or string, the corresponding DECLARE CURSOR must appear prior to the OPEN in the C program; if the *cursor-name* is specified by a host variable, the DECLARE CURSOR statement must execute before the OPEN statement.

Permissions

Must have SELECT permission on all tables in a SELECT statement, or EXECUTE permission on the procedure in a CALL statement.

Side effects

None.

See also

- "DECLARE CURSOR statement [ESQL] [SP]" on page 552
- "RESUME statement" on page 716
- "PREPARE statement [ESQL]" on page 694
- "FETCH statement [ESQL] [SP]" on page 605
- "RESUME statement" on page 716
- "CLOSE statement [ESQL] [SP]" on page 423
- "FOR statement" on page 608

Standards and compatibility

- **SQL/2003** Embedded SQL use is a core feature. Procedures use is a Persistent Stored Modules feature.

Example

The following examples show the use of OPEN in embedded SQL.

```
EXEC SQL OPEN employee_cursor;
```

and

```
EXEC SQL PREPARE emp_stat FROM
'SELECT empnum, empname FROM Employees WHERE name like ?';
EXEC SQL DECLARE employee_cursor CURSOR FOR emp_stat;
EXEC SQL OPEN employee_cursor USING :pattern;
```

The following example is from a procedure or trigger.

```
BEGIN
   DECLARE cur_employee CURSOR FOR
```

```
      SELECT Surname
      FROM Employees;
      DECLARE name CHAR(40);
      OPEN cur_employee;
      LP: LOOP
            FETCH NEXT cur_employee INTO name;
            IF SQLCODE <> 0 THEN LEAVE LP END IF;
       ...
      END LOOP
      CLOSE cur_employee;
   END
```

OUTPUT statement [Interactive SQL]

Use this statement to output the current query results to a file.

Syntax 1 - Output to a file

OUTPUT TO *filename*
[**APPEND**]
[**BYTE ORDER MARK** { **ON** | **OFF** }
[**COLUMN WIDTHS (** *integer*, ... **)**]
[**DELIMITED BY** *string*]
[**ENCODING** *encoding*]
[**ESCAPE CHARACTER** *character*]
[**ESCAPES** { **ON** | **OFF** }
[**FORMAT** *output-format*]
[**HEXADECIMAL** { **ON** | **OFF** | **ASIS** }]
[**QUOTE** *string* [**ALL**]]
[**VERBOSE**]

output-format :
TEXT
| **FIXED**
| **HTML**
| **SQL**
| **XML**

encoding : *string* or *identifier*

Syntax 2 - Output to an ODBC data source

OUTPUT
USING *connection-string*
INTO *destination-table-name*
[**CREATE TABLE** { **ON** | **OFF** }]

connection-string :
{ **DSN =** *odbc_data_source*
| **DRIVER =** *odbc_driver_name* [**;** *connection_parameter* **=** *value* [**;** ...]] }

Parameters

- **APPEND clause** This optional keyword is used to append the results of the query to the end of an existing output file without overwriting the previous contents of the file. If the APPEND clause is not used, the OUTPUT statement overwrites the contents of the output file by default. The APPEND keyword is valid if the output format is TEXT, FIXED, or SQL.

- **BYTE ORDER MARK clause** Use this clause to specify whether to include a byte order mark (BOM) at the start of a Unicode file. By default, this option is ON, which directs Interactive SQL to write a byte order mark (BOM) at the beginning of the file. If BYTE ORDER MARK is OFF, DBISQL does not write a BOM.

  The BYTE ORDER MARK clause is relevant only when writing TEXT formatted files. Attempts to use the BYTE ORDER MARK clause with FORMAT clauses other than TEXT returns an error.

  The BYTE ORDER MARK clause is used only when reading or writing files encoded with UTF-8 or UTF-16 (and their variants). Attempts to use the BYTE ORDER MARK clause with any other encoding returns an error.

- **COLUMN WIDTHS clause** The COLUMN WIDTHS clause is used to specify the column widths for the FIXED format output.

- **CREATE TABLE clause** Use the CREATE TABLE clause to specify whether to create the destination table if it does not exist. The default is ON.

- **DELIMITED BY clause** The DELIMITED BY clause is for the TEXT output format only. The delimiter string is placed between columns. The default is comma.

- **ENCODING clause** The ENCODING clause allows you to specify the encoding that is used to write the file. The ENCODING clause can only be used with the TEXT format.

  For more information about how to obtain the list of SQL Anywhere supported encodings, see "Supported character sets" [*SQL Anywhere Server - Database Administration*].

  With Interactive SQL, if the Encoding clause is not specified, the encoding that is used to write the file is determined in the following order:

  - the encoding specified with the default_isql_encoding option (if this option is set)
  - the default encoding of the operating system character set on the computer that Interactive SQL is running on

  The ENCODING clause is useful when you have data that cannot be represented in the operating system character set. In this case, if you do not use the ENCODING clause, characters that cannot be represented in the default encoding are lost in the output (that is, a lossy conversion occurs).

  If an encoding is specified in the OUTPUT statement, then the same ENCODING clause should be specified if the data is inserted using the INPUT statement.

  For more information about Interactive SQL and encodings, see "default_isql_encoding option [Interactive SQL]" [*SQL Anywhere Server - Database Administration*].

- **ESCAPE CHARACTER clause** The default escape character for characters stored as hexadecimal codes and symbols is a backslash (\). For example, \x0A is the linefeed character.

  This can be changed using the ESCAPE CHARACTER clause. For example, to use the exclamation mark as the escape character, specify:

  ```
  ... ESCAPE CHARACTER '!'
  ```

The new line character can be specified as '\n'. Other characters can be specified using hexadecimal ASCII codes, such as \x09 for the tab character. A sequence of two backslash characters (\\ ) is interpreted as a single backslash. A backslash followed by any character other than n, x, X, or \ is interpreted as two separate characters. For example, \q is interpreted as a backslash and the letter q.

- **ESCAPES clause** With ESCAPES turned on (the default), characters following the backslash character are recognized and interpreted as special characters by the database server. With ESCAPES turned off, the characters are written exactly as they appear in the source data.
- **FORMAT clause** The FORMAT clause allows you to specify the file format for the output. Allowable output formats are:
  - **TEXT** The output is a TEXT format file with one row per line in the file. All values are separated by commas, and strings are enclosed in apostrophes (single quotes). The delimiter and quote strings can be changed using the DELIMITED BY and QUOTE clauses. If ALL is specified in the QUOTE clause, all values (not just strings) are quoted. TEXT is the default output type

    Three other special sequences are also used. The two characters \n represent a newline character, \\ represents a single \, and the sequence \xDD represents the character with hexadecimal code DD.
  - **FIXED** The output is fixed format with each column having a fixed width. The width for each column can be specified using the COLUMN WIDTHS clause. No column headings are output in this format.

    If the COLUMN WIDTHS clause is omitted, the width for each column is computed from the data type for the column, and is large enough to hold any value of that data type. The exception is that LONG VARCHAR and LONG BINARY data default to 32 KB.
  - **HTML** The output is in the Hyper Text Markup Language format.
  - **SQL** The output is an Interactive SQL INPUT statement (required to recreate the information in the table) in a *.sql* file.
  - **XML** The output is an XML file encoded in UTF-8 and containing an embedded DTD. Binary values are encoded in CDATA blocks with the binary data rendered as 2-hex-digit strings.

  Note
  The INPUT statement does not accept XML as a file format.
- **HEXADECIMAL clause** The HEXADECIMAL clause specifies how binary values are output for the TEXT format. Allowable values are:
  - **ON** When set to ON, binary values are written with an Ox prefix followed by a series of hexadecimal pairs; for example, 0xabcd.
  - **OFF** When set to OFF, binary values are written one byte at a time. Each byte is prefixed with the escape character, such as a backslash, followed by x and then the hexadecimal pair for the byte; for example \xab\xcd.
  - **ASIS** When set to ASIS, values are written as is, without any escaping, even if the values contain control characters. ASIS is useful for text that contains formatting characters such as tabs or carriage returns.

- **QUOTE clause** The QUOTE clause is for the TEXT output format only. The quote string is placed around string values. The default is a single quote ('). If ALL is specified in the QUOTE clause, the quote string is placed around all values, not just around strings.
- **USING clause** The USING clause exports data to an ODBC data source. You can either specify the ODBC data source name with the DSN option, or the ODBC driver name and connection parameters with the DRIVER option. *Connection-parameter* is an optional list of database-specific connection parameters.

  Odbc-data-source is the name of a user or ODBC data source name. For example, *odbc-data-source* for the SQL Anywhere demo database is SQL Anywhere 11 Demo.

  Odbc-driver-name is the ODBC driver name. For a SQL Anywhere 11 database, the *odbc-driver-name* is SQL Anywhere 11; for an UltraLite database, *odbc-driver-name* is UltraLite 11.
- **VERBOSE clause** When the optional VERBOSE keyword is included, error messages about the query, the SQL statement used to select the data, and the data itself are written to the output file. Lines that do not contain data are prefixed by two hyphens. If VERBOSE is omitted (the default) only the data is written to the file. The VERBOSE keyword is valid if the output format is TEXT, FIXED, or SQL.

Remarks

The OUTPUT statement outputs data to a file or database. The OUTPUT statement is used directly after a statement that retrieves the data to be output. If the previous statement generates more than one result set, an error is returned.

The output format can be specified with the optional FORMAT clause. If no FORMAT clause is specified, the Interactive SQL output_format option setting is used (see “output_format option [Interactive SQL]” [*SQL Anywhere Server - Database Administration*]).

Because the INPUT statement is an Interactive SQL command, it cannot be used in any compound statement (such as IF), or in a stored procedure. See “Statements allowed in procedures, triggers, events, and batches” [*SQL Anywhere Server - SQL Usage*].

Permissions

None.

Side effects

In Interactive SQL, the Results tab displays the results of the current query.

See also

- “SELECT statement” on page 728
- “INPUT statement [Interactive SQL]” on page 644
- “UNLOAD statement” on page 772
- “Importing and exporting data” [*SQL Anywhere Server - SQL Usage*]
- “Using Interactive SQL” [*SQL Anywhere Server - Database Administration*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Examples

Place the contents of the Employees table in a text file:

```
SELECT *
FROM Employees;
OUTPUT TO 'Employees.txt'
FORMAT TEXT;
```

Place the contents of the Employees table at the end of an existing text file, and include any messages about the query in this file as well:

```
SELECT *
FROM Employees;
OUTPUT TO 'Employees.txt'
APPEND VERBOSE;
```

Suppose you need to export a value that contains an embedded line feed character. A line feed character has the numeric value 10, which you can represent as the string '\x0a' in a SQL statement. For example, execute the following statement, with HEXADECIMAL set to ON:

```
SELECT CAST ('line1\x0aline2' AS VARBINARY);
OUTPUT TO 'file.txt' HEXADECIMAL ON;
```

You get a file with one line in it containing the following text:

```
0x6c696e65310a6c696e6532
```

But if you execute the same statement with HEXADEMICAL set to OFF, you get the following:

```
'line1\x0Aline2'
```

Finally, if you set HEXADECIMAL to ASIS, you get a file with two lines:

```
'line1
line2'
```

You get two lines when you use ASIS because the embedded line feed character has been exported without being converted to a two digit hexadecimal representation, and without being prefixed by anything.

The following example outputs the data from the Customers table to a new table, Customers2:

```
SELECT * FROM Customers;
OUTPUT USING 'dsn=SQL Anywhere 11 Demo'
INTO "Customers2";
```

The following example copies the Customers table from the demo database to a fictitious database called *mydatabase.db*, using the DRIVER option.

```
SELECT * FROM Customers;
OUTPUT USING "DRIVER=SQL Anywhere 11;uid=dba;pwd=sql;dbf=c:\test
\mydatabase.db"
INTO "Customers";
```

The following example copies the Customers table from the SQL Anywhere demo database into a table called Customers in a fictitious UltraLite database, *myULDatabase.db*, using the DRIVER option.

```
SELECT * FROM Customers;
OUTPUT USING "DRIVER=Ultralite 11;dbf=c:\test\myULDatabase.udb"
INTO "Customers";
```

The following example copies the Customers table into a fictitious MySQL database called *mydatabase*, using the DRIVER option.

```
SELECT * FROM Customers;
OUTPUT USING "DRIVER=MySQL ODBC 5.1
Driver;DATABASE=mydatabase;SERVER=mySQLHost;UID=me;PWD=secret"
INTO "Customers";
```

SQL statements (P-Z)

The following sections define the syntax information for SQL statements P-Z.

See also

- "SQL statements (A-D)" on page 357
- "SQL statements (E-O)" on page 595
- "Common elements in SQL syntax" on page 354
- "Syntax conventions" on page 355
- "Statement applicability indicators" on page 356

PARAMETERS statement [Interactive SQL]

Use this statement to specify parameters to an Interactive SQL command file.

Syntax

PARAMETERS *parameter1*, *parameter2*, ...

Remarks

The PARAMETERS statement names the parameters for a command file, so that they can be referenced later in the command file.

Parameters are referenced by putting **{***parameter1***}** into the file where you want the named parameter to be substituted. There must be no spaces between the braces and the parameter name.

If a command file is invoked with less than the required number of parameters, Interactive SQL prompts for values of the missing parameters.

Permissions

None.

Side effects

None.

See also

- "READ statement [Interactive SQL]" on page 700
- "Using Interactive SQL" [*SQL Anywhere Server - Database Administration*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following Interactive SQL command file takes two parameters.

```
PARAMETERS department_id, file;
SELECT Surname
```

```
FROM Employees
WHERE DepartmentID = {department_id}
>#{ file }.dat;
```

If you save this script in a file named *test.sql*, you can run it from Interactive SQL using the following command:

```
READ test.sql [100] [data]
```

PASSTHROUGH statement [SQL Remote]

Use this statement to start or stop passthrough mode for SQL Remote administration. Forms 1 and 2 start passthrough mode, while form 3 stops passthrough mode.

Syntax 1

PASSTHROUGH [**ONLY**] **FOR** *userid*, ...

Syntax 2

PASSTHROUGH [**ONLY**] **FOR SUBSCRIPTION**
TO [*owner.*]*publication-name* [**(** *constant* **)**]

Syntax 3

PASSTHROUGH STOP

Remarks

In passthrough mode, any SQL statements are executed by the database server, and are also placed into the transaction log to be sent in messages to subscribers. If the ONLY keyword is used to start passthrough mode, the statements are not executed at the server; they are sent to recipients only. When a passthrough session contains calls to stored procedures, the procedures must exist in the server that is issuing the passthrough commands, even if they are not being executed locally at the server. The recipients of the passthrough SQL statements are either a list of user IDs (syntax 1) or all subscribers to a given publication. Passthrough mode may be used to apply changes to a remote database from the consolidated database or send statements from a remote database to the consolidated database.

Syntax 2 sends statements to remote databases whose subscriptions are started, and does not send statements to remote databases whose subscriptions are created and not started.

Permissions

Must have DBA authority.

Side effects

None.

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

```
PASSTHROUGH FOR rem_db ;
...
( SQL statements to be executed at the remote database )
...
PASSTHROUGH STOP ;
```

PREPARE statement [ESQL]

Use this statement to prepare a statement to be executed later, or used to define a cursor.

Syntax

PREPARE *statement-name*
  FROM *statement*
[**DESCRIBE** *describe-type* **INTO** [[**SQL**] **DESCRIPTOR**] *descriptor*]
[**WITH EXECUTE**]

statement-name : *identifier* or *hostvar*

statement : *string* or *hostvar*

describe-type :
 [**ALL** | **BIND VARIABLES** | **INPUT** | **OUTPUT** | **SELECT LIST**]
 [**LONG NAMES** [[[**OWNER.**]**TABLE.**]**COLUMN**]
   | **WITH VARIABLE RESULT**]

Parameters

- **statement-name** The statement name can be an identifier or host variable. However, you should not use an identifier when using multiple SQLCAs. If you do, two prepared statements may have the same statement number, which could cause the wrong statement to be executed or opened. Also, using an identifier for a statement name is not recommended for multi-threaded applications where the statement name may be referenced by multiple threads concurrently.

- **DESCRIBE clause** If DESCRIBE INTO DESCRIPTOR is used, the prepared statement is described into the specified descriptor. The describe type may be any of the describe types allowed in the DESCRIBE statement.

- **WITH EXECUTE clause** If the WITH EXECUTE clause is used, the statement is executed if and only if it is not a CALL or SELECT statement, and it has no host variables. The statement is immediately dropped after a successful execution. If the PREPARE and the DESCRIBE (if any) are successful but the statement cannot be executed, a warning SQLCODE 111, SQLSTATE 01W08 is set, and the statement is not dropped.

  The DESCRIBE INTO DESCRIPTOR and WITH EXECUTE clauses may improve performance because they cut down on the required client/server communication.

- **WITH VARIABLE RESULT clause** The WITH VARIABLE RESULT clause is used to describe procedures that may have more than one result set, with different numbers or types of columns.

  If WITH VARIABLE RESULT is used, the database server sets the SQLCOUNT value after the describe to one of the following values:

- ○ **0** The result set may change: The procedure call should be described again following each OPEN statement.
- ○ **1** The result set is fixed. No re-describing is required.

> **Static and dynamic**
> For compatibility reasons, preparing COMMIT, PREPARE TO COMMIT, and ROLLBACK statements is still supported. However, it is recommended that you do all transaction management operations with static embedded SQL because certain application environments may require it. Also, other embedded SQL systems do not support dynamic transaction management operations.

Remarks

The PREPARE statement prepares a SQL statement from the *statement* and associates the prepared statement with *statement-name*. This statement name is referenced to execute the statement, or to open a cursor if the statement is a SELECT statement. The *statement-name* may be a host variable of type a_sql_statement_number defined in the *sqlca.h* header file that is automatically included. If an identifier is used for the *statement-name*, only one statement per module may be prepared with this *statement-name*.

If a host variable is used for *statement-name*, it must have the type SHORT INT. There is a typedef for this type in *sqlca.h* called a_sql_statement_number. This type is recognized by the SQL preprocessor and can be used in a DECLARE section. The host variable is filled in by the database during the PREPARE statement, and need not be initialized by the programmer.

Permissions

None.

Side effects

Any statement previously prepared with the same name is lost.

The statement is dropped after use only if you use WITH EXECUTE and the execution is successful. You should ensure that you DROP the statement after use in other circumstances. If you do not, the memory associated with the statement is not reclaimed.

See also

- "DECLARE CURSOR statement [ESQL] [SP]" on page 552
- "DESCRIBE statement [ESQL]" on page 562
- "OPEN statement [ESQL] [SP]" on page 683
- "EXECUTE statement [ESQL]" on page 597
- "DROP STATEMENT statement [ESQL]" on page 583

Standards and compatibility

- **SQL/2003** Core feature.

Example

The following statement prepares a simple query:

```
EXEC SQL PREPARE employee_statement FROM
'SELECT Surname FROM Employees';
```

PREPARE TO COMMIT statement

Use this statement to check whether a COMMIT can be performed successfully.

Syntax

PREPARE TO COMMIT

Remarks

The PREPARE TO COMMIT statement tests whether a COMMIT can be performed successfully. The statement will cause an error if a COMMIT is impossible without violating the integrity of the database.

The PREPARE TO COMMIT statement cannot be used in stored procedures, triggers, events, or batches.

Permissions

None.

Side effects

None.

See also

- "COMMIT statement" on page 426
- "ROLLBACK statement" on page 723

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following sequence of statements leads to an error because of foreign key checking on the Employees table.

```
EXECUTE IMMEDIATE
   "SET OPTION wait_for_commit = 'On'";
EXECUTE IMMEDIATE "DELETE FROM Employees
   WHERE EmployeeID = 160";
EXECUTE IMMEDIATE "PREPARE TO COMMIT";
```

The following sequence of statements does not cause an error when the delete statement is executed, even though it causes integrity violations. The PREPARE TO COMMIT statement returns an error.

```
SET OPTION wait_for_commit= 'On';
DELETE
FROM Departments
WHERE DepartmentID = 100;
PREPARE TO COMMIT;
```

PRINT statement [T-SQL]

Use this statement to return a message to the client, or display a message in the message window of the database server.

Syntax

PRINT *format-string* [, *arg-list*]

Remarks

The PRINT statement returns a message to the client window if you are connected from an Open Client application or jConnect application. If you are connected from an embedded SQL or ODBC application, the message is displayed on the database server messages window.

The format string can contain placeholders for the arguments in the optional argument list. These placeholders are of the form *%nn!*, where *nn* is an integer between 1 and 20.

Permissions

None.

Side effects

None.

See also

- "MESSAGE statement" on page 680

Standards and compatibility

- **SQL/2003** Transact-SQL extension.

Example

The following statement displays a message:

```
PRINT 'Display this message';
```

The following statement illustrates the use of placeholders in the PRINT statement:

```
DECLARE @var1 INT, @var2 INT
SELECT @var1 = 3, @var2 = 5
PRINT 'Variable 1 = %1!, Variable 2 = %2!', @var1, @var2
```

PUT statement [ESQL]

Use this statement to insert a row into the specified cursor.

Syntax

PUT *cursor-name*
{ **USING DESCRIPTOR** *sqlda-name* | **FROM** *hostvar-list* }

[**INTO** { **DESCRIPTOR** *sqlda-name* | *hostvar-list* }]
[**ARRAY :***row-count*]

cursor-name : *identifier* or *hostvar*

sqlda-name : *identifier*

hostvar-list : may contain indicator variables

row-count : *integer* or *hostvar*

Remarks

Inserts a row into the named cursor. Values for the columns are taken from the first SQLDA or the host variable list, in a one-to-one correspondence with the columns in the INSERT statement (for an INSERT cursor) or the columns in the select list (for a SELECT cursor).

The PUT statement can be used only on a cursor over an INSERT or SELECT statement that references a single table in the FROM clause, or that references an updatable view consisting of a single base table.

If the sqldata pointer in the SQLDA is the null pointer, no value is specified for that column. If the column has a DEFAULT VALUE associated with it, that is used; otherwise, a NULL value is used.

The second SQLDA or host variable list contains the results of the PUT statement.

The optional ARRAY clause can be used to carry out wide puts, which insert more than one row at a time and which may improve performance. The integer value is the number of rows to be inserted. The SQLDA must contain a variable for each entry (number of rows * number of columns). The first row is placed in SQLDA variables 0 to (columns per row)-1, and so on.

Inserting into a cursor
For scroll (values sensitive) cursors, the inserted row will appear if the new row matches the WHERE clause and the keyset cursor has not finished populating. For dynamic cursors, if the inserted row matches the WHERE clause, the row may appear. Insensitive cursors cannot be updated.

For information about putting LONG VARCHAR or LONG BINARY values into the database, see "SET statement" on page 735.

Permissions

Must have INSERT permission.

Side effects

When inserting rows into a value-sensitive (keyset driven) cursor, the inserted rows appear at the end of the result set, even when they do not match the WHERE clause of the query or if an ORDER BY clause would normally have placed them at another location in the result set. See "Modifying rows through a cursor" [*SQL Anywhere Server - Programming*].

See also

- "UPDATE statement" on page 777
- "UPDATE (positioned) statement [ESQL] [SP]" on page 782
- "DELETE statement" on page 558
- "DELETE (positioned) statement [ESQL] [SP]" on page 561
- "INSERT statement" on page 649

Standards and compatibility

- **SQL/2003** Core feature.

Example

The following statement illustrates the use of PUT in embedded SQL:

```
EXEC SQL PUT cur_employee FROM :employeeID, :surname;
```

RAISERROR statement

Use this statement to signal an error and to send a message to the client.

Syntax

RAISERROR *error-number* [*format-string*] [, *arg-list*]

Parameters

- **error-number** The *error-number* is a five-digit integer greater than 17000. The error number is stored in the global variable @@error.

- **format-string** If *format-string* is not supplied or is empty, the error number is used to locate an error message in the system tables. Adaptive Server Enterprise obtains messages 17000-19999 from the SYSMESSAGES table. In SQL Anywhere this table is an empty view, so errors in this range should provide a format string. Messages for error numbers of 20000 or greater are obtained from the ISYSUSERMESSAGE table.

  In SQL Anywhere, the *format-string* length can be up to 255 bytes.

  The extended values supported by the Adaptive Server Enterprise RAISERROR statement are not supported in SQL Anywhere.

  The format string can contain placeholders for the arguments in the optional argument list. These placeholders are of the form %*nn*!, where *nn* is an integer between 1 and 20.

  Intermediate RAISERROR status and code information is lost after the procedure terminates. If at return time an error occurs along with the RAISERROR then the error information is returned and the RAISERROR information is lost. The application can query intermediate RAISERROR statuses by examining @@error global variable at different execution points.

Remarks

The RAISERROR statement allows user-defined errors to be signaled and sends a message on the client.

Permissions

None.

Side effects

None.

See also

- "CREATE TRIGGER statement [T-SQL]" on page 544
- "CREATE TRIGGER statement" on page 538
- "on_tsql_error option [compatibility]" [*SQL Anywhere Server - Database Administration*]
- "continue_after_raiserror option [compatibility]" [*SQL Anywhere Server - Database Administration*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement raises error 23000, which is in the range for user-defined errors, and sends a message to the client. Note that there is no comma between the *error-number* and the *format-string* parameters. The first item following a comma is interpreted as the first item in the argument list.

```
RAISERROR 23000 'Invalid entry for this column: %1!', @val
```

The next example uses RAISERROR to disallow connections.

```
CREATE PROCEDURE DBA.login_check()
BEGIN
    // Allow a maximum of 3 concurrent connections
    IF( DB_PROPERTY('ConnCount') > 3 ) THEN
         RAISERROR 28000
      'User %1! is not allowed to connect -- there are ' ||
                      'already %2! users logged on',
      Current User,
      CAST( DB_PROPERTY( 'ConnCount' ) AS INT )-1;
    ELSE
         CALL sp_login_environment;
    END IF;
END
go
GRANT EXECUTE ON DBA.login_check TO PUBLIC
go
SET OPTION PUBLIC.login_procedure='DBA.login_check'
go
```

For an alternate way to disallow connections, see "login_procedure option [database]" [*SQL Anywhere Server - Database Administration*].

READ statement [Interactive SQL]

Use this statement to read Interactive SQL statements from a file.

Syntax

READ [**ENCODING** *encoding*] *filename* **[** *parameter* **]** ...

encoding : *identifier* or *string*

Remarks

The READ statement reads a sequence of Interactive SQL statements from the named file. This file can contain any valid Interactive SQL statements, including other READ statements. READ statements can be nested to any depth. If the file name does not contain an absolute path, Interactive SQL searches for the file. Interactive SQL first searches the directory of the calling file (if there is one), then the current directory, and then the directories specified in the environment variable SQLPATH, and then the directories specified in the environment variable PATH. If the named file has no file extension, Interactive SQL searches each directory for the same file name with the extension *.sql*.

The ENCODING clause allows you to specify the encoding that is used to read the file. The READ statement does not process escape characters when it reads a file. It assumes that the entire file is in the specified encoding.

With Interactive SQL, if the ENCODING clause is not specified, the encoding that is used to read the file is determined in the following order:

- the encoding specified with the default_isql_encoding option (if this option is set)
- the default encoding of the operating system character set on the computer that Interactive SQL is running on

For more information about Interactive SQL and encodings, see "default_isql_encoding option [Interactive SQL]" [*SQL Anywhere Server - Database Administration*].

Parameters can be listed after the name of the command file. These parameters correspond to the parameters named in the PARAMETERS statement at the beginning of the statement file. See "PARAMETERS statement [Interactive SQL]" on page 692)

Parameter names must be enclosed in square brackets. Interactive SQL substitutes the corresponding parameter wherever the source file contains **{**`parameter-name`**}**, where *parameter-name* is the name of the appropriate parameter.

The parameters passed to a command file can be identifiers, numbers, quoted identifiers, or strings. When quotes are used around a parameter, the quotes are put into the text during the substitution. Parameters that are not identifiers, numbers, or strings (contain spaces or tabs) must be enclosed in square brackets ([ ]). This allows for arbitrary textual substitution in the command file.

If not enough parameters are passed to the command file, Interactive SQL prompts for values for the missing parameters.

When executing a *reload.sql* file with Interactive SQL, you must specify the encryption key as a parameter. If you do not provide the key in the READ statement, Interactive SQL prompts for the key. See "Interactive SQL utility (dbisql)" [*SQL Anywhere Server - Database Administration*].

The location of *filename* is determined based on the location of the READ statement, as follows:

- If the READ statement is executed directly in Interactive SQL, the path to *filename* is resolved relative to the directory in which Interactive SQL is running.
- If the READ statements reside in an external file (for example, a *.sql* file), Interactive SQL first attempts to resolve the path to *filename* relative to the location of the external file. If unsuccessful, Interactive SQL looks for *filename* in a path relative to the directory in which Interactive SQL is running.

Permissions

None.

Side effects

None.

See also

- "PARAMETERS statement [Interactive SQL]" on page 692
- "Using Interactive SQL" [*SQL Anywhere Server - Database Administration*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following are examples of the READ statement.

```
READ status.rpt '160';
READ birthday.sql [>= '1988-1-1'] [<= '1988-1-30'];
```

READTEXT statement [T-SQL]

Use this statement to read text and image values from the database, starting from a specified offset and reading a specified number of bytes.

Syntax

READTEXT *table-name.column-name*
text-pointer-offset-size
[**HOLDLOCK**]

Remarks

READTEXT is used to read image and text values from the database. You cannot perform READTEXT operations on views.

Permissions

SELECT permissions on the table.

Side effects

None.

See also

- "WRITETEXT statement [T-SQL]" on page 795
- "GET DATA statement [ESQL]" on page 622
- "TEXTPTR function [Text and image]" on page 323

Standards and compatibility

- **SQL/2003** Transact-SQL extension.

REFRESH MATERIALIZED VIEW statement

Initializes or refreshes the data in a materialized view by executing its query definition.

Syntax

```
REFRESH MATERIALIZED VIEW view-list
 [ WITH {
     ISOLATION LEVEL isolation-level
     | { EXCLUSIVE | SHARE } MODE } ]
 [ FORCE BUILD ]

view-list :
[ owner.]materialized-view-name [, ... ]

isolation-level :
READ UNCOMMITTED
| READ COMMITTED
| SERIALIZABLE
| REPEATABLE READ
| SNAPSHOT
```

Parameters

- **WITH clause** Use the WITH clause to specify the type of locking to use on the underlying base tables during the refresh. The type of locking determines how the materialized view is populated and how concurrency for transactions is affected. The WITH clause setting does not impact the type of lock placed on the materialized view itself, which is always an exclusive lock. The possible locking clauses you can specify are:
  - **ISOLATION LEVEL isolation-level** Use WITH ISOLATION LEVEL to change the isolation level for the execution of the refresh operation. The original isolation level is restored for the connection when statement execution finishes.

    For immediate views, *isolation-level* can only be SERIALIZABLE.

    For snapshot isolation, only snapshot level is supported (specify SNAPSHOT); statement-level and readonly-statement-snapshot are not supported.

    For information about isolation levels, see "Using transactions and isolation levels" [*SQL Anywhere Server - SQL Usage*], and "Isolation levels and consistency" [*SQL Anywhere Server - SQL Usage*].
  - **EXCLUSIVE MODE** Use WITH EXCLUSIVE MODE if you do not want to change the isolation level, but want to guarantee that the data is updated to be consistent with committed data in the

underlying tables. When using WITH EXCLUSIVE MODE, exclusive table locks are placed on all underlying base tables and no other transaction can execute queries, updates, or any other action against the underlying table(s) until the refresh operation is complete. If exclusive table locks cannot be obtained, the refresh operation fails and an error is returned. See "Table locks" [*SQL Anywhere Server - SQL Usage*].

  - **SHARE MODE** Use WITH SHARE MODE to give read access on underlying tables to other transactions while the refresh operation takes place. When this clause is specified, shared table locks are obtained on all underlying base tables before the refresh operation is performed and until the refresh operation completes. See "Table locks" [*SQL Anywhere Server - SQL Usage*].

- **FORCE BUILD clause** By default, when you execute a REFRESH MATERIALIZED VIEW statement, the database server checks whether the materialized view is stale (that is, underlying tables have changed since the materialized view was last refreshed). If it is not stale, the refresh does not take place. Specify the FORCE BUILD clause to force a refresh of the materialized view regardless of whether the materialized view is stale.

Remarks

Use this statement to initialize or refresh the materialized views listed in *view-list*.

If a REFRESH MATERIALIZED VIEW statement is executed against a materialized view that is not stale, a refresh is not performed unless the FORCE BUILD clause is specified.

The default refresh behavior for locking and data concurrency is as follows:

- If the view is an immediate view, the default refresh behavior is WITH SHARE MODE, regardless of whether snapshot isolation is enabled.
- If the view is a manual view and snapshot isolation *is in use*, the default is WITH ISOLATION LEVEL SNAPSHOT.
- If the view is a manual view and snapshot isolation *is not in use*, the default is WITH SHARE MODE.

For more information about isolation levels and on enabling snapshot isolation, see "Isolation levels and consistency" [*SQL Anywhere Server - SQL Usage*], and "allow_snapshot_isolation option [database]" [*SQL Anywhere Server - Database Administration*].

Several options need to have specific values for a REFRESH MATERIALIZED VIEW to succeed, and for the view to be used in optimization. Additionally, there are option settings that are stored for each materialized view when it is created. To refresh the view, or to use the view in optimization these option settings must match the current options. See "Restrictions on materialized views" [*SQL Anywhere Server - SQL Usage*].

When a refresh fails after having done partial work, the view is left in an uninitialized state, and the data cannot be restored to what it was before the refresh started. Examine the error that occurred when the refresh failed, resolve the issue that caused the failure, and execute the REFRESH MATERIALIZED VIEW statement again.

You can also use the IMMEDIATE REFRESH clause of the ALTER MATERIALIZED VIEW statement to change the view to be refreshed immediately when underlying data changes. See "ALTER MATERIALIZED VIEW statement" on page 373.

This statement cannot be executed when there are cursors opened with the WITH HOLD clause that use either statement or transaction snapshots. See "Snapshot isolation" [*SQL Anywhere Server - SQL Usage*].

Permissions

Must have INSERT permission on the materialized view, and SELECT permission on the tables in the materialized view definition.

Side effects

Any open cursors that reference the materialized view are closed.

A checkpoint is performed at the beginning of execution.

Automatic commits are performed at the beginning and end of execution.

While executing, an exclusive schema lock is placed on the materialized view being refreshed using the connection blocking option, and shared schema locks, without blocking, are placed on all tables referenced by the materialized view. If the WITH clause is specified, extra locks may be acquired on the underlying tables. Also, until refreshing is complete, the materialized view is in an uninitialized state, making it unavailable to the database server or optimizer.

See also

- "Working with materialized views" [*SQL Anywhere Server - SQL Usage*]
- "CREATE MATERIALIZED VIEW statement" on page 477
- "ALTER MATERIALIZED VIEW statement" on page 373
- "Isolation levels and consistency" [*SQL Anywhere Server - SQL Usage*]
- "blocking option [database]" [*SQL Anywhere Server - Database Administration*]
- "Table locks" [*SQL Anywhere Server - SQL Usage*]
- "Schema locks" [*SQL Anywhere Server - SQL Usage*]
- "sa_refresh_materialized_views system procedure" on page 926
- "sa_materialized_view_info system procedure" on page 899
- "sa_materialized_view_can_be_immediate system procedure" on page 897

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

Suppose you create a materialized view, EmployeeConfid99, and then populate it with data using the following statements:

```
CREATE MATERIALIZED VIEW EmployeeConfid99 AS
   SELECT EmployeeID, Employees.DepartmentID, SocialSecurityNumber, Salary,
ManagerID,
      Departments.DepartmentName, Departments.DepartmentHeadID
   FROM Employees, Departments
   WHERE Employees.DepartmentID=Departments.DepartmentID;
REFRESH MATERIALIZED VIEW EmployeeConfid99;
```

Later, after the view has been in use, you want to refresh the view using the READ COMMITTED isolation level (isolation level 1), and you want the view to be rebuilt. You could execute the following statement:

```
REFRESH MATERIALIZED VIEW EmployeeConfid99
   WITH ISOLATION LEVEL READ COMMITTED
   FORCE BUILD;
```

Caution
When you are done with this example, you should drop the materialized view you created. Otherwise, you will not be able to make schema changes to its underlying tables Employees and Departments, when trying out other examples. You cannot alter the schema of a table that has enabled, dependent materialized view. See "Drop materialized views" [*SQL Anywhere Server - SQL Usage*].

REFRESH TEXT INDEX statement

Refreshes a text index.

Syntax

REFRESH TEXT INDEX *text-index-name* **ON** [*owner.*]*table-name*
[**WITH** {
    ISOLATION LEVEL *isolation-level*
    | **EXCLUSIVE MODE**
    | **SHARE MODE** }]
[**FORCE** { **BUILD** | **INCREMENTAL** }]

Parameters

- **WITH clause** Use the WITH clause to specify what kind of locks to obtain on the underlying base tables during the refresh. The types of locks obtained determine how the text index is populated and how concurrency for transactions is affected. If you do not specify the WITH clause, the default is WITH ISOLATION LEVEL READ UNCOMMITTED, regardless of any isolation level set for the connection.

  You can specify the following WITH clause options:

  - **ISOLATION LEVEL isolation-level** Use WITH ISOLATION LEVEL to change the isolation level for the execution of the refresh operation. For information about isolation levels, see "Using transactions and isolation levels" [*SQL Anywhere Server - SQL Usage*], and "Isolation levels and consistency" [*SQL Anywhere Server - SQL Usage*].

    The original isolation level of the connection is restored at the end of the statement execution.

  - **EXCLUSIVE MODE** Use WITH EXCLUSIVE MODE if you do not want to change the isolation level, but want to guarantee that the data is updated to be consistent with committed data in the underlying table. When using WITH EXCLUSIVE MODE, exclusive table locks are placed on the underlying base table and no other transaction can execute queries, updates, or any other action against the underlying table(s) until the refresh operation is complete. If table locks cannot be obtained, the refresh operation fails and an error is returned. See "Table locks" [*SQL Anywhere Server - SQL Usage*].

  - **SHARE MODE** Use WITH SHARE MODE to give read access on the underlying table to other transactions while the refresh operation takes place. When this clause is specified, shared table locks are obtained on the underlying base table before the refresh operation is performed and and are held until the refresh operation completes. See "Table locks" [*SQL Anywhere Server - SQL Usage*].

- **FORCE clause** Use this clause to specify the refresh method. If this clause is not specified, the database server decides whether to do an incremental update or a full rebuild based on how much of the table has changed. See "Text index refresh types" [*SQL Anywhere Server - SQL Usage*].

- **FORCE BUILD clause** Use this clause to force a complete rebuild of the text index.
- **FORCE INCREMENTAL clause** Use this clause to force an incremental update of the text index.

Remarks

This statement can only be used on text indexes defined as MANUAL REFRESH or AUTO REFRESH.

When using the FORCE clause, you can examine the results of the sa_text_index_stats system procedure to decide whether a complete rebuild (FORCE BUILD), or incremental update (FORCE INCREMENTAL) is most appropriate. See "sa_text_index_stats system procedure" on page 957.

You cannot execute the REFRESH TEXT INDEX statement on a text index that is defined as IMMEDIATE REFRESH.

For MANUAL REFRESH text indexes, use the sa_text_index_stats system procedure to determine whether the text index should be refreshed. Divide pending_length by doc_length, and use the percentage as a guide for deciding whether a refresh is required. To determine the type of rebuild required, use the same process for deleted_length and doc_count.

Permissions

Must be the owner of the underlying table, or have either DBA authority or REFERENCES permission.

This statement cannot be executed when there are cursors opened with the WITH HOLD clause that use either statement or transaction snapshots. See "Snapshot isolation" [*SQL Anywhere Server - SQL Usage*].

Side effects

Automatic commit.

See also

- "Text indexes" [*SQL Anywhere Server - SQL Usage*]
- "CREATE TEXT INDEX statement" on page 536
- "ALTER TEXT INDEX statement" on page 399
- "DROP TEXT INDEX statement" on page 590
- "TRUNCATE TEXT INDEX statement" on page 770
- "sa_refresh_text_indexes system procedure" on page 926
- "sa_text_index_stats system procedure" on page 957

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement refreshes a text index called MarketingTextIndex, forcing it to be rebuilt.

```
REFRESH TEXT INDEX MarketingTextIndex ON MarketingInformation
   FORCE BUILD;
```

REFRESH TRACING LEVEL statement

Use the REFRESH TRACING LEVEL statement to reload the tracing levels from the sa_diagnostic_tracing_level table while a tracing session is in progress.

Syntax

REFRESH TRACING LEVEL

Remarks

This statement is used to reload the tracing level information from the sa_diagnostic_tracing_level table. It must be called from the database being profiled.

When a tracing session is first started, rows from the sa_diagnostic_tracing_level table are loaded into server memory to control what kind of information is traced. If you want to change the types of data being traced, without stopping and restarting the tracing session to do so, you can do so by manually deleting or inserting the appropriate rows in the sa_diagnostic_tracing_level table, and then executing the REFRESH TRACING LEVEL statement to reload the settings.

To see the current tracing levels, query the sa_diagnostic_tracing_level table as follows:

```
SELECT * FROM sa_diagnostic_tracing_level WHERE enabled = 1;
```

For more information about the sa_diagnostic_tracing_level system table, see "sa_diagnostic_tracing_level table" on page 824.

Permissions

Must have DBA authority.

Side effects

None.

See also

- "ATTACH TRACING statement" on page 406
- "DETACH TRACING statement" on page 568
- "Advanced application profiling using diagnostic tracing" [*SQL Anywhere Server - SQL Usage*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

Suppose you are troubleshooting a performance problem. You turn on a high level of tracing for the entire database to capture the queries that are causing the problem. After starting the tracing session, you find that capturing all queries for all users on your system slows down your database too much, so you decide you'd rather limit tracing to one user and wait for that user to report a problem. However, you do not want to stop the tracing session to change the settings.

You can do this in Sybase Central by using the Database Tracing wizard, which is the recommended method. However, you can also do this from the command line by replacing the rows in sa_diagnostic_tracing_level table where scope=DATABASE and enabled=1, with equivalent rows where scope=USER, identifier=*userid*, enabled=1, and so on. Then, you execute a REFRESH TRACING LEVEL statement to continue tracing using use the new settings.

RELEASE SAVEPOINT statement

Use this statement to release a savepoint within the current transaction.

Syntax

RELEASE SAVEPOINT [*savepoint-name*]

Remarks

Release a savepoint. The *savepoint-name* is an identifier specified on a SAVEPOINT statement within the current transaction. If *savepoint-name* is omitted, the most recent savepoint is released.

Releasing a savepoint does not do any type of COMMIT. It simply removes the savepoint from the list of currently active savepoints.

Permissions

There must have been a corresponding SAVEPOINT within the current transaction.

Side effects

None.

See also

- "BEGIN TRANSACTION statement [T-SQL]" on page 416
- "COMMIT statement" on page 426
- "ROLLBACK statement" on page 723
- "ROLLBACK TO SAVEPOINT statement" on page 724
- "SAVEPOINT statement" on page 727
- "Savepoints within transactions" [*SQL Anywhere Server - SQL Usage*]

Standards and compatibility

- **SQL/2003** Vendor extension.

REMOTE RESET statement [SQL Remote]

Use this statement in custom database-extraction procedures to start all subscriptions for a remote user in a single transaction.

Syntax

REMOTE RESET *userid*

Remarks

This statement starts all subscriptions for a remote user in a single transaction. It sets the log_sent and confirm_sent values in ISYSREMOTEUSER table to the current position in the transaction log. It also sets the created and started values in ISYSSUBSCRIPTION to the current position in the transaction log for all subscriptions for this remote user. The statement does not do a commit. You must do an explicit commit after this call.

To write an extraction process that is safe on a live database, the data must be extracted at isolation level 3 in the same transaction as the subscriptions are started.

This statement is an alternative to start subscription. START SUBSCRIPTION has an implicit commit as a side effect, so that if a remote user has several subscriptions, it is impossible to start them all in one transaction using START SUBSCRIPTION.

Permissions

Must have DBA authority.

Side effects

No automatic commit is done by this statement.

See also

- "START SUBSCRIPTION statement [SQL Remote]" on page 756
- "ISYSREMOTEUSER system table" on page 806

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement resets the subscriptions for remote user SamS:

```
REMOTE RESET SamS;
COMMIT;
```

REMOVE EXTERNAL OBJECT statement

Use this statement to remove an external object from the database.

Syntax

REMOVE EXTERNAL OBJECT *object-name*

Parameters

- **object-name** The name of the external object.

Remarks

For more information about external environments, see "Overview of external environments" [*SQL Anywhere Server - Programming*].

Permissions

Must have DBA authority.

Side effects

None

See also

- "Overview of external environments" [*SQL Anywhere Server - Programming*]
- "ALTER EXTERNAL ENVIRONMENT statement" on page 367
- "INSTALL EXTERNAL OBJECT statement" on page 654
- "START EXTERNAL ENVIRONMENT statement" on page 754
- "STOP EXTERNAL ENVIRONMENT statement" on page 761
- "SYSEXTERNENV system view" on page 1000
- "SYSEXTERNENVOBJECT system view" on page 1000

Standards and compatibility

- **SQL/2003** Vendor extension.

REMOVE JAVA statement

Use this statement to remove a class or a jar file from a database. When a class is removed it is no longer available for use as a column or variable type.

The class or jar must already be installed.

Syntax

REMOVE JAVA *classes-to-remove*

classes-to-remove :
 CLASS *java-class-name*, ... | **JAR** *jar-name*, ...

Parameters

- **CLASS clause** The *java-class-name* parameter is the name of one or more Java class to be removed. These classes must be installed classes in the current database.

- **JAR clause** The *jar-name* is a character string value of maximum length 255.

  Each *jar-name* must be equal to the *jar-name* of a retained jar in the current database. Equality of *jar-name* is determined by the character string comparison rules of the SQL system.

Remarks

Removes a class or jar file from the database.

Permissions

- Must have DBA authority.

- Not supported on Windows Mobile.

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement removes a Java class named Demo from the current database.

```
REMOVE JAVA CLASS Demo;
```

REORGANIZE TABLE statement

Use this statement to defragment tables when a full rebuild of the database is not possible due to the requirements for continuous access to the database.

Syntax

REORGANIZE TABLE [*owner*.]*table-name*
[{ **PRIMARY KEY**
| **FOREIGN KEY** *foreign-key-name*
| **INDEX** *index-name* }]

Parameters

Reorganize the table according to the values in one of the following:

- **PRIMARY KEY clause** Reorganizes the primary key index for the table.
- **FOREIGN KEY clause** Reorganizes the specified foreign key.
- **INDEX clause** Reorganizes the specified index.

Remarks

Table fragmentation can impede performance. Use this statement to defragment rows in a table, or to compress indexes which have become sparse due to DELETEs. It may also reduce the total number of pages used to store the table and its indexes, and it may reduce the number of levels in an index tree. However, it will not result in a reduction of the total size of the database file. It is recommended that you use the sa_table_fragmentation and sa_index_density system procedures to select tables worth processing.

If an index or key is not specified, the reorganization process defragments rows in the table by deleting and re-inserting groups of rows. For each group, an exclusive lock on the table is obtained. Once the group has been processed, the lock is released and re-acquired (waiting if necessary), providing an opportunity for other connections to access the table. Checkpoints are suspended while a group is being processed; once a group is finished, a checkpoint may occur. The rows are processed in order by primary key; if the table has no primary key, an error results. The processed rows are re-inserted at the end of the table, resulting in the rows being clustered by primary key at the end of the process. Note that the same amount of work is required, regardless of how fragmented the rows initially were.

If an index or key is specified, the specified index is processed. For the duration of the operation, an exclusive lock is held on the table and checkpoints are suspended. Any attempts to access the table by other connections will block or fail, depending on their setting of the blocking option. The duration of the lock is minimized by pre-reading the index pages prior to obtaining the exclusive lock.

Since reorganization may modify many pages, the checkpoint log can become large. This can result in a increase in the database file size. However, this increase is temporary since the checkpoint log is deleted at shutdown and the file is truncated at that point.

This statement is not logged to the transaction log.

This statement cannot be executed when there are cursors opened with the WITH HOLD clause that use either statement or transaction snapshots. See "Snapshot isolation" [*SQL Anywhere Server - SQL Usage*].

Permissions

- Must be either the owner of the table, or a user with DBA authority.
- Not supported on Windows Mobile.

Side effects

Prior to starting the reorganization, a checkpoint is done to try to maximize the number of free pages. Also, when executing the REORGANIZE TABLE statement, there is an implied commit for approximately every 100 rows, so reorganizing a large table causes multiple commits to take place.

Standards and compatibility

- **SQL/2003** Vendor extension.

Examples

The following statement reorganizes the primary key index for the Employees table:

```
REORGANIZE TABLE Employees
PRIMARY KEY;
```

The following statement reorganizes the table pages of the Employees table:

```
REORGANIZE TABLE Employees;
```

The following statement reorganizes the index IX_product_name on the Products table:

```
REORGANIZE TABLE Products
   INDEX IX_product_name;
```

The following statement reorganizes the foreign key FK_DepartmentID_DepartmentID for the Employees table:

```
REORGANIZE TABLE Employees
   FOREIGN KEY FK_DepartmentID_DepartmentID;
```

RESIGNAL statement

Use this statement to resignal an exception condition.

Syntax

RESIGNAL [*exception-name*]

Remarks

Within an exception handler, RESIGNAL allows you to quit the compound statement with the exception still active, or to quit reporting another named exception. The exception is handled by another exception handler or returned to the application. Any actions by the exception handler before the RESIGNAL are undone.

Permissions

None.

Side effects

None.

See also

- "SIGNAL statement" on page 750
- "BEGIN statement" on page 413
- "Using exception handlers in procedures and triggers" [*SQL Anywhere Server - SQL Usage*]
- "RAISERROR statement" on page 699

Standards and compatibility

- **SQL/2003** Persistent Stored Module feature.

Example

The following fragment returns all exceptions except Column Not Found to the application.

```
...
DECLARE COLUMN_NOT_FOUND EXCEPTION
   FOR SQLSTATE '52003';
...
EXCEPTION
WHEN COLUMN_NOT_FOUND THEN
SET message='Column not found';
WHEN OTHERS THEN
RESIGNAL;
```

RESTORE DATABASE statement

Use this statement to restore a backed up database from an archive.

Syntax

RESTORE DATABASE *filename*
FROM *archive-root*
[**CATALOG ONLY**
  | [**RENAME** *dbspace-name* **TO** *new-dbspace-name*] ...]
[**HISTORY** { **ON** | **OFF** }]

filename : *string* | *variable*
archive-root : *string* | *variable*
new-dbspace-name : *string* | *variable*

Parameters

- **CATALOG ONLY clause** Retrieves information about the named archive, and places it in the backup history file (*backup.syb*), but does not restore any data from the archive.
- **RENAME clause** Allows you to specify a new location for each dbspace. You cannot use the RENAME clause to change the dbspace name. However, you can use the RENAME clause to change the file name.
- **HISTORY clause** Allows you to control whether the RESTORE DATABASE operation is recorded in the history file, *backup.syb*.

Remarks

Unless HISTORY OFF is specified, each RESTORE DATABASE operation updates a backup history file called *backup.syb*. This file records the BACKUP and RESTORE operations that have been performed on a database server. You may want to prevent the RESTORE DATABASE operation from being recorded in *backup.syb* if the following conditions apply:

- your RESTORE DATABASE operations occur frequently
- there is no procedure to periodically archive or delete the *backup.syb* file
- disk space is very limited

RESTORE DATABASE replaces the database that is being restored. If you need incremental backups, use the image format of the BACKUP command and save only the transaction log; however, image backups to tape are not supported.

Permissions

The permissions required to execute this statement are set on the server command line, using the -gu option. The default setting is to require DBA authority. See "-gu server option" [*SQL Anywhere Server - Database Administration*].

This statement is not supported on Windows Mobile.

Side effects

None.

See also

- "Pre-defined dbspaces" [*SQL Anywhere Server - Database Administration*]
- "BACKUP statement" on page 407
- "Backup and data recovery" [*SQL Anywhere Server - Database Administration*]
- "SALOGDIR environment variable" [*SQL Anywhere Server - Database Administration*]

Standards and compatibility

- **SQL/2003** Vendor extension.
- **Windows Mobile** Not supported on Windows Mobile.

Example

The following example restores a database from a tape drive. The number of backslashes that are required depends on which database you are connected to when you execute RESTORE DATABASE. The database affects the setting of the escape_character option. It is normally set to On, but is set to Off in utility_db. When connected to any database other than utility_db, the extra backslashes are required.

```
RESTORE DATABASE 'd:\\dbhome\\mydatabase.db'
FROM '\\\\.\\tape0';
```

RESUME statement

Use this statement to resume execution of a cursor that returns result sets.

Syntax

RESUME *cursor-name*

cursor-name : *identifier* | *hostvar*

Remarks

This statement resumes execution of a procedure that returns result sets. The procedure executes until the next result set (SELECT statement with no INTO clause) is encountered. If the procedure completes and no result set is found, the SQLSTATE_PROCEDURE_COMPLETE warning is set. This warning is also set when you RESUME a cursor for a SELECT statement.

The RESUME statement is not supported in Interactive SQL. If you want to view multiple result sets in Interactive SQL, you can set the isql_show_multiple_result_sets option to ON, or choose Tools » Options, and then select Show Multiple Result sets on the Results tab.

Permissions

The cursor must have been previously opened.

Side effects

None.

See also

- "DECLARE CURSOR statement [ESQL] [SP]" on page 552
- "FETCH statement [ESQL] [SP]" on page 605
- "Returning results from procedures" [*SQL Anywhere Server - SQL Usage*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

Following are embedded SQL examples.

```
1. EXEC SQL RESUME cur_employee;
2. EXEC SQL RESUME :cursor_var;
```

RETURN statement

Use this statement to exit from a function, procedure or batch unconditionally, optionally providing a return value.

Syntax

RETURN [*expression*]

Remarks

A RETURN statement causes an immediate exit from a block of SQL. If *expression* is supplied, the value of *expression* is returned as the value of the function or procedure.

If the RETURN appears inside an inner BEGIN block, it is the outer BEGIN block that is terminated.

Statements following a RETURN statement are not executed.

Within a function, the expression should be of the same data type as the function's RETURNS data type.

Within a procedure, RETURN is used for Transact-SQL-compatibility, and is used to return an integer error code.

Permissions

None.

Side effects

None.

See also

- "CREATE FUNCTION statement (web services)" on page 467
- "CREATE PROCEDURE statement (web services)" on page 495
- "BEGIN statement" on page 413

Standards and compatibility

- **SQL/2003** Persistent Stored Module feature.

Example

The following function returns the product of three numbers:

```
CREATE FUNCTION product (
   a NUMERIC,
   b NUMERIC,
   c NUMERIC )
RETURNS NUMERIC
BEGIN
   RETURN ( a * b * c );
END;
```

Calculate the product of three numbers:

```
SELECT product(2, 3, 4);
```

product(2, 3, 4)
24

The following procedure uses the RETURN statement to avoid executing a complex query if it is meaningless:

```
CREATE PROCEDURE customer_products
( in customer_ID integer DEFAULT NULL)
RESULT ( ID integer, quantity_ordered integer )
BEGIN
   IF customer_ID NOT IN (SELECT ID FROM Customers)
   OR customer_ID IS NULL THEN
      RETURN
   ELSE
      SELECT Products.ID,sum(
         SalesOrderItems.Quantity )
      FROM  Products,
            SalesOrderItems,
            SalesOrders
      WHERE SalesOrders.CustomerID=customer_ID
      AND SalesOrders.ID=SalesOrderItems.ID
      AND SalesOrderItems.ProductID=Products.ID
      GROUP BY Products.ID
   END IF
END;
```

REVOKE statement

Use this statement to remove permissions from users.

Syntax 1

REVOKE *permission*, ... **FROM** *userid*, ...

permission :
CONNECT
| **DBA**
| **BACKUP**
| **CREATE ON** *dbspace*
| **GROUP**
| **INTEGRATED LOGIN**
| **KERBEROS LOGIN**
| **MEMBERSHIP IN GROUP** *userid*, ...
| **PROFILE**
| **RESOURCE**
| **VALIDATE**

Syntax 2

REVOKE *table-permission*, ...
ON [*owner*.]*table-name*
FROM *userid*, ...

table-permission :
ALL [**PRIVILEGES**]

| **ALTER**
| **DELETE**
| **INSERT**
| **REFERENCES** [**(** *column-name*, ... **)**]
| **SELECT** [**(** *column-name*, ... **)**]
| **UPDATE** [**(** *column-name*, ... **)**]

Syntax 3

REVOKE EXECUTE
ON [*owner*.]*procedure-name*
FROM *userid*, ...

Remarks

The REVOKE statement removes permissions given using the GRANT statement. Syntax 1 revokes special user permissions. Syntax 2 revokes table permissions. Syntax 3 revokes permission to execute a procedure.

REVOKE CONNECT removes a user ID from a database, and also destroys any objects (tables, views, procedures, and so on) owned by that user and any permissions granted by that user. You cannot execute a REVOKE CONNECT on a user if the user being dropped owns a table referenced by a view owned by another user.

REVOKE GROUP automatically revokes MEMBERSHIP IN GROUP from all members of the group.

When you add a user to a group, the user inherits all the permissions assigned to that group. SQL Anywhere does not allow you to revoke a subset of the permissions that a user inherits as a member of a group because you can only revoke permissions that are explicitly given by a GRANT statement. If you need to have different permissions for different users, you can create different groups with the appropriate permissions, or you can explicitly grant each user the permissions they require.

When you grant or revoke group permissions for tables, views, or procedures, all members of the group inherit those changes. The DBA, RESOURCE, and GROUP permissions are not inherited: you must assign them to each individual user ID that requires them.

If you give a user WITH GRANT OPTION permission, and later revoke that permission, you also revoke any permissions that user granted to others while they had the WITH GRANT OPTION permission.

Permissions

Must be the grantor of the permissions that are being revoked or have DBA authority.

If you are revoking connect permissions or table permissions from another user, the other user must not be connected to the database. You cannot revoke connect permissions from DBO.

When you are connected to the utility database, executing REVOKE CONNECT FROM DBA disables future connections to the utility database. This means that no future connections can be made to the utility database unless you use a connection that existed before the REVOKE CONNECT was done, or restart the database server.

Side effects

Automatic commit.

See also

- "GRANT statement" on page 627

Standards and compatibility

- **SQL/2003** Syntax 1 is a vendor extension. Syntax 2 is a core feature. Syntax 3 is a Persistent Stored Modules feature.

Example

Prevent user Dave from updating the Employees table.

```
REVOKE UPDATE ON Employees FROM Dave;
```

Revoke resource permissions from user Jim.

```
REVOKE RESOURCE FROM Jim;
```

Revoke an integrated login mapping from the user profile named Administrator.

```
REVOKE INTEGRATED LOGIN FROM Administrator;
```

Disallow the Finance group from executing the procedure ShowCustomers.

```
REVOKE EXECUTE ON ShowCustomers FROM Finance;
```

Drop the user ID FranW from the database.

```
REVOKE CONNECT FROM FranW;
```

REVOKE CONSOLIDATE statement [SQL Remote]

Use this statement to stop a consolidated database from receiving SQL Remote messages from this database.

Syntax

REVOKE CONSOLIDATE FROM *userid*

Remarks

CONSOLIDATE permissions must be granted at a remote database for the user ID representing the consolidated database. The REVOKE CONSOLIDATE statement removes the consolidated database user ID from the list of users receiving messages from the current database.

Permissions

Must have DBA authority.

Side effects

Automatic commit. Drops all subscriptions for the user.

See also

- "REVOKE PUBLISH statement [SQL Remote]" on page 721
- "REVOKE REMOTE statement [SQL Remote]" on page 722
- "REVOKE REMOTE DBA statement [SQL Remote]" on page 723
- "GRANT CONSOLIDATE statement [SQL Remote]" on page 632

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

- The following statement revokes consolidated status from the condb user ID:

```
REVOKE CONSOLIDATE FROM condb;
```

REVOKE PUBLISH statement [SQL Remote]

Use this statement to terminate the identification of the named user ID as the CURRENT publisher.

Syntax

REVOKE PUBLISH FROM *userid*

Remarks

Each database in a SQL Remote installation is identified in outgoing messages by a publisher user ID. The current publisher user ID can be found using the CURRENT PUBLISHER special constant. The following query identifies the current publisher:

```
SELECT CURRENT PUBLISHER;
```

The REVOKE PUBLISH statement ends the identification of the named user ID as the publisher.

You should not REVOKE PUBLISH from a database while the database has active SQL Remote publications or subscriptions.

Issuing a REVOKE PUBLISH statement at a database has several consequences for a SQL Remote installation:

- You will not be able to insert data into any tables with a CURRENT PUBLISHER column as part of the primary key. Any outgoing messages will not be identified with a publisher user ID, and so will not be accepted by recipient databases.

If you change the publisher user ID at any consolidated or remote database in a SQL Remote installation, you must ensure that the new publisher user ID is granted REMOTE permissions on all databases receiving messages from the database. This will generally require all subscriptions to be dropped and recreated.

Permissions

Must have DBA authority.

Side effects

Automatic commit.

See also

- "GRANT PUBLISH statement [SQL Remote]" on page 633
- "REVOKE REMOTE statement [SQL Remote]" on page 722
- "REVOKE REMOTE DBA statement [SQL Remote]" on page 723
- "REVOKE CONSOLIDATE statement [SQL Remote]" on page 720

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

```
REVOKE PUBLISH FROM publisher_ID;
```

REVOKE REMOTE statement [SQL Remote]

Use this statement to stop a user from being able to receive SQL Remote messages from this database.

Syntax

REVOKE REMOTE FROM *userid*, ...

Remarks

REMOTE permissions are required for a user ID to receive messages in a SQL Remote replication installation. The REVOKE REMOTE statement removes a user ID from the list of users receiving messages from the current database.

Permissions

Must have DBA authority.

Side effects

Automatic commit. Drops all subscriptions for the user.

See also

- "REVOKE PUBLISH statement [SQL Remote]" on page 721
- "GRANT REMOTE statement [SQL Remote]" on page 634
- "REVOKE REMOTE DBA statement [SQL Remote]" on page 723
- "REVOKE CONSOLIDATE statement [SQL Remote]" on page 720

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

```
REVOKE REMOTE FROM SamS;
```

REVOKE REMOTE DBA statement [SQL Remote]

This statement revokes REMOTE DBA authority from a user ID.

Syntax 1

REVOKE REMOTE DBA
FROM *userid*, ...

Remarks

In MobiLink, REMOTE DBA authority is a level of permission required by the SQL Anywhere synchronization client (dbmlsync).

In SQL Remote, REMOTE DBA authority enables the Message Agent to have full access to the database to make any changes contained in the messages, while avoiding security problems associated with distributing DBA user IDs passwords.

Permissions

Must have DBA authority.

Side effects

Automatic commit.

See also

- "REVOKE PUBLISH statement [SQL Remote]" on page 721
- "REVOKE REMOTE statement [SQL Remote]" on page 722
- "GRANT REMOTE DBA statement [MobiLink] [SQL Remote]" on page 636
- "REVOKE CONSOLIDATE statement [SQL Remote]" on page 720
- "Initiating synchronization" [*MobiLink - Client Administration*]
- "Grant REMOTE DBA authority" [*SQL Remote*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Examples

The following statement revokes REMOTE DBA permission from user S_Beaulieu.

```
REVOKE REMOTE DBA FROM S_Beaulieu;
```

ROLLBACK statement

Use this statement to end a transaction and undo any changes made since the last COMMIT or ROLLBACK.

Syntax

ROLLBACK [**WORK**]

Remarks

A transaction is the logical unit of work done on one database connection to a database between COMMIT or ROLLBACK statements. The ROLLBACK statement ends the current transaction and undoes all changes made to the database since the previous COMMIT or ROLLBACK.

Permissions

None.

Side effects

Closes all cursors not opened WITH HOLD.

See also

- "COMMIT statement" on page 426
- "ROLLBACK TO SAVEPOINT statement" on page 724

Standards and compatibility

- **SQL/2003** Core feature.

ROLLBACK TO SAVEPOINT statement

To cancel any changes made since a SAVEPOINT.

Syntax

ROLLBACK TO SAVEPOINT [*savepoint-name*]

Remarks

The ROLLBACK TO SAVEPOINT statement will undo any changes that have been made since the SAVEPOINT was established. Changes made prior to the SAVEPOINT are not undone; they are still pending.

The *savepoint-name* is an identifier that was specified on a SAVEPOINT statement within the current transaction. If *savepoint-name* is omitted, the most recent savepoint is used. Any savepoints since the named savepoint are automatically released.

Permissions

There must have been a corresponding SAVEPOINT within the current transaction.

Side effects

None.

See also

- “BEGIN TRANSACTION statement [T-SQL]” on page 416
- “COMMIT statement” on page 426
- “RELEASE SAVEPOINT statement” on page 709
- “ROLLBACK statement” on page 723
- “SAVEPOINT statement” on page 727
- “Savepoints within transactions” [*SQL Anywhere Server - SQL Usage*]

Standards and compatibility

- **SQL/2003** SQL/foundation feature outside core SQL.

ROLLBACK TRANSACTION statement [T-SQL]

Use this statement to cancel any changes made since a SAVE TRANSACTION.

Syntax

ROLLBACK TRANSACTION [*savepoint-name*]

Remarks

The ROLLBACK TRANSACTION statement undoes any changes that have been made since a savepoint was established using SAVE TRANSACTION. Changes made prior to the SAVE TRANSACTION are not undone; they are still pending.

The *savepoint-name* is an identifier that was specified on a SAVE TRANSACTION statement within the current transaction. If *savepoint-name* is omitted, all outstanding changes are rolled back. Any savepoints since the named savepoint are automatically released.

Permissions

There must be a corresponding SAVE TRANSACTION within the current transaction.

Side effects

None.

See also

- “ROLLBACK TO SAVEPOINT statement” on page 724
- “BEGIN TRANSACTION statement [T-SQL]” on page 416
- “COMMIT statement” on page 426,
- “SAVE TRANSACTION statement [T-SQL]” on page 726

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following example displays five rows with values 10, 20, and so on. The effect of the DELETE, but not the prior INSERTs or UPDATE, is undone by the ROLLBACK TRANSACTION statement.

```
BEGIN
    SELECT row_num INTO #tmp
    FROM sa_rowgenerator( 1, 5 )
    UPDATE #tmp SET row_num=row_num*10
    SAVE TRANSACTION before_delete
    DELETE FROM #tmp WHERE row_num >= 3
    ROLLBACK TRANSACTION before_delete
    SELECT * FROM #tmp
END
```

ROLLBACK TRIGGER statement

Use this statement to undo any changes made in a trigger.

Syntax

ROLLBACK TRIGGER [**WITH** *raiserror-statement*]

Remarks

The ROLLBACK TRIGGER statement rolls back the work done in a trigger, including the data modification that caused the trigger to fire.

Optionally, a RAISERROR statement can be issued. If a RAISERROR statement is issued, an error is returned to the application. If no RAISERROR statement is issued, no error is returned.

If a ROLLBACK TRIGGER statement is used within a nested trigger and without a RAISERROR statement, only the innermost trigger and the statement which caused it to fire are undone.

Permissions

None.

Side effects

None

See also

- "CREATE TRIGGER statement" on page 538
- "ROLLBACK statement" on page 723
- "ROLLBACK TO SAVEPOINT statement" on page 724
- "RAISERROR statement" on page 699

Standards and compatibility

- **SQL/2003** Transact-SQL extension.

SAVE TRANSACTION statement [T-SQL]

Use this statement to establish a savepoint within the current transaction.

Syntax

SAVE TRANSACTION *savepoint-name*

Remarks

Establish a savepoint within the current transaction. The *savepoint-name* is an identifier that can be used in a ROLLBACK TRANSACTION statement. All savepoints are automatically released when a transaction ends. See "Savepoints within transactions" [*SQL Anywhere Server - SQL Usage*].

Permissions

None.

Side effects

None.

See also

- "SAVEPOINT statement" on page 727
- "BEGIN TRANSACTION statement [T-SQL]" on page 416
- "COMMIT statement" on page 426
- "ROLLBACK TRANSACTION statement [T-SQL]" on page 725

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following example displays five rows with values 10, 20, and so on. The effect of the DELETE, but not the prior INSERTs or UPDATE, is undone by the ROLLBACK TRANSACTION statement.

```
BEGIN
    SELECT row_num INTO #tmp
    FROM sa_rowgenerator( 1, 5 )
    UPDATE #tmp SET row_num=row_num*10
    SAVE TRANSACTION before_delete
    DELETE FROM #tmp WHERE row_num >= 3
    ROLLBACK TRANSACTION before_delete
    SELECT * FROM #tmp
END
```

SAVEPOINT statement

Use this statement to establish a savepoint within the current transaction.

Syntax

SAVEPOINT [*savepoint-name*]

Remarks

Establish a savepoint within the current transaction. The *savepoint-name* is an identifier that can be used in a RELEASE SAVEPOINT or ROLLBACK TO SAVEPOINT statement. All savepoints are automatically released when a transaction ends. See "Savepoints within transactions" [*SQL Anywhere Server - SQL Usage*].

Savepoints that are established while a trigger or atomic compound statement is executing are automatically released when the atomic operation ends.

You cannot modify data in a proxy table from within a savepoint.

Permissions

None.

Side effects

None.

See also

- "RELEASE SAVEPOINT statement" on page 709
- "ROLLBACK TO SAVEPOINT statement" on page 724

Standards and compatibility

- **SQL/2003** SQL/foundation feature outside core SQL.

SELECT statement

Use this statement to retrieve information from the database.

Syntax

```
[ WITH temporary-views ]
  SELECT [ ALL | DISTINCT ] [ row-limitation ] select-list
[ INTO  { hostvar-list | variable-list | table-name } ]
[ INTO LOCAL TEMPORARY TABLE { table-name } ]
[ FROM from-expression ]
[ WHERE search-condition ]
[ GROUP BY group-by-expression ]
[ HAVING search-condition ]
[ WINDOW window-expression ]
[ ORDER BY { expression | integer } [ ASC | DESC ], ... ]
[ FOR { UPDATE [ cursor-concurrency ] | READ ONLY } ]
[ FOR XML xml-mode ]
[ OPTION( query-hint, ... ) ]

temporary-views :
  regular-view, ...
| RECURSIVE { regular-view | recursive-view }, ...

regular-view :
  view-name [ ( column-name, ... ) ]
  AS ( subquery )
```

recursive-view :
  view-name **(** *column-name*, ... **)**
  AS (*initial-subquery* **UNION ALL** *recursive-subquery* **)**

row-limitation :
  FIRST | **TOP** *n* [**START AT** *m*]

select-list :
expression [[**AS**] *alias-name*], ...
| *****
| *window-function* **OVER** { *window-name* | *window-spec* }
  [[**AS**] *alias-name*]

from-expression : See "FROM clause" on page 613.

group-by-expression : See "GROUP BY clause" on page 637.

search-condition : See "Search conditions" on page 35.

window-name : *identifier*

window-expression : See "WINDOW clause" on page 793.

window-spec : See "WINDOW clause" on page 793.

window-function :
RANK( )
| **DENSE_RANK( )**
| **PERCENT_RANK( )**
| **CUME_DIST( )**
| **ROW_NUMBER( )**
| *aggregate-function*

cursor-concurrency :
BY { **VALUES** | **TIMESTAMP** | **LOCK** }

xml-mode :
RAW [**, ELEMENTS**]
| **AUTO** [**, ELEMENTS**]
| **EXPLICIT**

query-hint :
MATERIALIZED VIEW OPTIMIZATION *option-value*
| **FORCE OPTIMIZATION**
| **FORCE NO OPTIMIZATION**
| *option-name* = *option-value*

option-name : *identifier*

option-value : *hostvar* (indicator allowed), *string*, *identifier*, or *number*

Parameters

- **WITH or WITH RECURSIVE clause** Define one or more common table expressions, also known as temporary views, to be used elsewhere in the remainder of the statement. These expressions may be non-recursive, or may be self-recursive. Recursive common table expressions may appear alone, or intermixed with non-recursive expressions, only if the RECURSIVE keyword is specified. Mutually recursive common table expressions are not supported.

This clause is permitted only if the SELECT statement appears in one of the following locations:

- Within a top-level SELECT statement
- Within the top-level SELECT statement of a VIEW definition
- Within a top-level SELECT statement within an INSERT statement

Recursive expressions consist of an initial subquery and a recursive subquery. The initial-query implicitly defines the schema of the view. The recursive subquery must contain a reference to the view within the FROM clause. During each iteration, this reference refers only to the rows added to the view in the previous iteration. The reference must not appear on the null-supplying side of an outer join. A recursive common table expression must not use aggregate functions and must not contain a GROUP BY, ORDER BY, or DISTINCT clause. See "Common table expressions" [*SQL Anywhere Server - SQL Usage*].

The WITH clause is not supported with remote tables.

- **ALL or DISTINCT clause** ALL (the default) returns all rows that satisfy the clauses of the SELECT statement. If DISTINCT is specified, duplicate output rows are eliminated. Many statements take significantly longer to execute when DISTINCT is specified, so you should reserve DISTINCT for cases where it is necessary.

- ***row-limitation* clause** The row limiting clause allows you to return only a subset of the rows that satisfy the WHERE clause. The TOP and START AT values can be a host variable, integer constant, or integer variable. The TOP value must be greater than or equal to 0. The START AT value must be greater than 0. Normally, when specifying these clauses, an ORDER BY clause is specified as well to order the rows in a meaningful manner. See "Explicitly limiting the number of rows returned by a query" [*SQL Anywhere Server - SQL Usage*].

- ***select-list* clause** The *select-list* is a list of expressions, separated by commas, specifying what is retrieved from the database. An asterisk (*) means select all columns of all tables in the FROM clause.

  Aggregate functions are allowed in the *select-list*. Subqueries are also allowed in the *select-list*. Each subquery must be within parentheses.

  Alias names can be used throughout the query to represent the aliased expression.

  Alias names are also displayed by Interactive SQL at the top of each column of output from the SELECT statement. If the optional alias name is not specified after an expression, Interactive SQL will display the expression itself.

- **INTO clause** Following are the three uses of the INTO clause:

  - **INTO *hostvar-list* clause** This clause is used in embedded SQL only. It specifies where the results of the SELECT statement will go. There must be one host variable item for each item in the *select-list*. *select-list* items are put into the host variables in order. An indicator host variable is also allowed with each host variable, so the program can tell if the *select-list* item was NULL.

  - **INTO *variable-list* clause** This clause is used in procedures and triggers only. It specifies where the results of the SELECT statement will go. There must be one variable for each item in the *select-list*. *select-list* items are put into the variables in order.

  - **INTO *table-name* clause** This clause is used to create a table and fill it with data.

For permanent tables to be created, the query must satisfy one of the following conditions:

- The *select-list* contains more than one item, and the INTO target is a single *table-name* identifier.
- The *select-list* contains a * and the INTO target is specified as *owner.table*.

To create a permanent table with one column, the table name must be specified as *owner.table*.

This statement causes a COMMIT before execution as a side effect of creating the table. RESOURCE authority is required to execute this statement. No permissions are granted on the new table: the statement is a short form for CREATE TABLE followed by INSERT ... SELECT.

Tables created using this clause do not have a primary key defined. You can add a primary key using ALTER TABLE. A primary key should be added before applying any UPDATEs or DELETEs to the table; otherwise, these operations result in all column values being logged in the transaction log for the affected rows.

- **INTO LOCAL TEMPORARY TABLE** This clause is used to create a local, temporary table and populate it with the results of the query. When you use this clause, you do not need to start the temporary table name with #.
- **FROM clause** Rows are retrieved from the tables and views specified in the *table-expression*. A SELECT statement with no FROM clause can be used to display the values of expressions not derived from tables. For example, these two statements are equivalent and display the value of the global variable @@version.

  ```
  SELECT @@version;
  SELECT @@version FROM DUMMY;
  ```

  See "FROM clause" on page 613.
- **WHERE clause** This clause specifies which rows are selected from the tables named in the FROM clause. It can be used to do joins between multiple tables, as an alternative to the ON phrase (which is part of the FROM clause). See "Search conditions" on page 35 and "FROM clause" on page 613.
- **GROUP BY clause** You can group by columns, alias names, or functions. The result of the query contains one row for each distinct set of values in the named columns, aliases, or functions. As with DISTINCT and the set operations UNION, INTERSECT, and EXCEPT, the GROUP BY clause treats NULL values in the same manner as any other value in each domain. In other words, multiple NULL values in a grouping attribute will form a single group. Aggregate functions can then be applied to these groups to get meaningful results.

  When GROUP BY is used, the *select-list*, HAVING clause, and ORDER BY clause must not reference any identifier that is not named in the GROUP BY clause. The exception is that the *select-list* and HAVING clause can contain aggregate functions.
- **HAVING clause** This clause selects rows based on the group values and not on the individual row values. The HAVING clause can only be used if either the statement has a GROUP BY clause or the *select-list* consists solely of aggregate functions. Any column names referenced in the HAVING clause must either be in the GROUP BY clause or be used as a parameter to an aggregate function in the HAVING clause.
- **WINDOW clause** This clause defines all or part of a window for use with window functions such as AVG and RANK. See "WINDOW clause" on page 793.

- **ORDER BY clause** This clause sorts the results of a query. Each item in the ORDER BY list can be labeled as ASC for ascending order (the default) or DESC for descending order. If the expression is an integer *n*, then the query results are sorted by the *n*th item in the *select-list*.

  The only way to ensure that rows are returned in a particular order is to use ORDER BY. In the absence of an ORDER BY clause, SQL Anywhere returns rows in whatever order is most efficient. This means that the appearance of result sets may vary depending on when you last accessed the row and other factors.

  In embedded SQL, the SELECT statement is used for retrieving results from the database and placing the values into host variables via the INTO clause. The SELECT statement must return only one row. For multiple row queries, you must use cursors.

- **FOR UPDATE or FOR READ ONLY clause** These clauses specify whether updates are allowed through a cursor opened on the query, and if so, what concurrency semantics will be used. This clause cannot be used with the FOR XML clause.

  When you specify FOR UPDATE BY TIMESTAMP or FOR UPDATE BY VALUES, the database server uses optimistic concurrency by using a keyset-driven cursor. In this situation, lost updates can occur.

  If you do not use a FOR clause in the SELECT statement, the updatability of a cursor depends on the cursor's declaration (see "DECLARE statement" on page 551 and "FOR statement" on page 608) and how cursor concurrency is specified by the API. In ODBC, JDBC, and OLE DB, statement updatability is explicit and a read-only, forward-only cursor is used unless it is overridden by the application. In Open Client, embedded SQL, and within stored procedures, cursor updatability does not have to be specified, and the default is FOR UPDATE.

  To ensure that a statement acquires an intent lock, you must do one of the following:

  - specify FOR UPDATE BY LOCK in the query
  - specify HOLDLOCK, WITH (HOLDLOCK), WITH (UPDLOCK), or WITH (XLOCK) in the FROM clause of the query
  - open the cursor with API calls that specify CONCUR_LOCK
  - fetch the rows with attributes indicating fetch for update

  In addition to cursor updatability, statement updatability is also dependent on the setting of the ansi_update_constraints database option and the specific characteristics of the statement, including whether the statement contains ORDER BY, DISTINCT, GROUP BY, HAVING, UNION, aggregate functions, joins, or non-updatable views.

  For more information about cursor sensitivity, see "SQL Anywhere cursors" [*SQL Anywhere Server - Programming*].

  For more information about ODBC concurrency, see the discussion of SQLSetStmtAttr in "Choosing ODBC cursor characteristics" [*SQL Anywhere Server - Programming*].

  For more information about the ansi_update_constraints database option, see "ansi_update_constraints option [compatibility]" [*SQL Anywhere Server - Database Administration*].

  For more information about cursor updatability, see "Understanding updatable statements" [*SQL Anywhere Server - Programming*].

- **FOR XML clause** This clause specifies that the result set is to be returned as an XML document. The format of the XML depends on the mode you specify. This clause cannot be used with the FOR UPDATE or FOR READ ONLY clause.

  When you specify RAW mode, each row in the result set is represented as an XML <row> element, and each column is represented as an attribute of the <row> element.

  AUTO mode returns the query results as nested XML elements. Each table referenced in the *select-list* is represented as an element in the XML. The order of nesting for the elements is based on the order that tables are referenced in the *select-list*.

  EXPLICIT mode allows you to control the form of the generated XML document. Using EXPLICIT mode offers more flexibility in naming elements and specifying the nesting structure than either RAW or AUTO mode. See “Using FOR XML EXPLICIT” [*SQL Anywhere Server - SQL Usage*].

  For more information about using the FOR XML clause, see “Using the FOR XML clause to retrieve query results as XML” [*SQL Anywhere Server - SQL Usage*].

- **OPTION clause** This clause provides hints about how to process the query. The following query hints are supported:

  - **MATERIALIZED VIEW OPTIMIZATION clause** Use the MATERIALIZED VIEW OPTIMIZATION clause to specify how the optimizer should make use of materialized views when processing the query. The specified *option-value* overrides the materialized_view_optimization database option for this query only. Possible values for *option-value* are the same values available for the materialized_view_optimization database option. See “materialized_view_optimization option [database]” [*SQL Anywhere Server - Database Administration*].

  - **FORCE OPTIMIZATION clause** When a query specification contains only simple queries (single-block, single-table queries that contain equality conditions in the WHERE clause that uniquely identify a specific row), it typically bypasses cost-based optimization during processing. In some cases you may want cost-based optimization to occur. For example, if you want materialized views to be considered during query processing, view matching must occur. However, view matching only occurs during cost-base optimization. If you want cost-based optimization to occur for a query, but your query specification contains only simple queries, specify the FORCE OPTIMIZATION option to ensure that the optimizer performs cost-based optimization on the query.

    Similarly, specifying the FORCE OPTIMIZATION option in a SELECT statement inside of a procedure forces the use of the optimizer for any call to the procedure. In this case, plans for the statement are not cached.

    For more information about simple queries and view matching, see “Query processing phases” [*SQL Anywhere Server - SQL Usage*], and “Eligibility to skip query processing phases” [*SQL Anywhere Server - SQL Usage*].

  - **FORCE NO OPTIMIZATION clause** Specify the FORCE NO OPTIMIZATION clause if you want the statement to bypass the optimizer. If the statement is too complex to process in this way--possibly due to the setting of database options or characteristics of the schema or query--the statement fails and the database server returns an error. For more information about statements that can bypass the optimizer, see “Eligibility to skip query processing phases” [*SQL Anywhere Server - SQL Usage*].

- ***option-name* = *option-value*** Specify an option setting that takes precedence over any public or temporary option settings that are in effect, for this statement only. The supported options are:

  - "isolation_level option [database] [compatibility]" [*SQL Anywhere Server - Database Administration*]
  - "max_query_tasks option [database]" [*SQL Anywhere Server - Database Administration*]
  - "optimization_goal option [database]" [*SQL Anywhere Server - Database Administration*]
  - "optimization_level option [database]" [*SQL Anywhere Server - Database Administration*]
  - "optimization_workload option [database]" [*SQL Anywhere Server - Database Administration*]
  - "user_estimates option [database]" [*SQL Anywhere Server - Database Administration*]

Remarks

The SELECT statement can be used:

- for retrieving results from the database.
- in Interactive SQL to browse data in the database, or to export data from the database to an external file.
- in procedures and triggers or in embedded SQL. A SELECT statement with an INTO clause is used for retrieving results from the database when the SELECT statement only returns one row. For multiple row queries, you must use cursors.
- to return a result set from a procedure.

Note
When a GROUP BY expression is used in a SELECT statement, the *select-list*, HAVING clause, and ORDER BY clause can reference only identifiers named in the GROUP BY clause. The exception is that the *select-list* and HAVING clause may contain aggregate functions.

Permissions

Must have SELECT permission on the named tables and views.

Side effects

None.

See also

- "Expressions" on page 16
- "FROM clause" on page 613
- "Search conditions" on page 35
- "UNION clause" on page 771
- "EXCEPT clause" on page 595
- "INTERSECT clause" on page 657
- "Joins: Retrieving data from several tables" [*SQL Anywhere Server - SQL Usage*]

Standards and compatibility

- **SQL/2003** Core feature. The complexity of the SELECT statement means that you should check individual clauses against the standard. For example, the ROLLUP keyword is part of feature T431.

FOR UPDATE, FOR READ ONLY, and FOR UPDATE (*column-list*) are core features.

FOR UPDATE BY [LOCK | TIMESTAMP | VALUES] is a SQL Anywhere vendor extension.

Example

This example returns the total number of employees in the Employees table.

```
SELECT COUNT(*)
FROM Employees;
```

This example lists all customers and the total value of their orders.

```
SELECT CompanyName,
   CAST( SUM( SalesOrderItems.Quantity *
   Products.UnitPrice ) AS INTEGER ) VALUE
FROM Customers
   JOIN SalesOrders
   JOIN SalesOrderItems
   JOIN Products
GROUP BY CompanyName
ORDER BY VALUE DESC;
```

The following statement shows an embedded SQL SELECT statement where the number of employees in the Employees table is selected into the :size host variable:

```
SELECT count(*) INTO :size
FROM Employees;
```

The following statement is optimized to return the first row in the result set quickly:

```
SELECT Name
FROM Products
GROUP BY Name
HAVING COUNT( * ) > 1
AND MAX( UnitPrice ) > 10
OPTION( optimization_goal = 'first-row' );
```

SET statement

Use this statement to assign a value to a SQL variable.

Syntax

SET *identifier* **=** *expression*

Remarks

The SET statement assigns a new value to a variable. The variable must have been previously created using a CREATE VARIABLE statement or DECLARE statement, or it must be an OUTPUT parameter for a procedure. The variable name can optionally use the Transact-SQL convention of an @ sign preceding the name. For example:

```
SET @localvar = 42
```

A variable can be used in a SQL statement anywhere a column name is allowed. If a column name exists with the same name as the variable, the variable value is used.

Variables are local to the current connection, and disappear when you disconnect from the database or use the DROP VARIABLE statement. They are not affected by COMMIT or ROLLBACK statements.

Variables are necessary for creating large text or binary objects for INSERT or UPDATE statements from embedded SQL programs because embedded SQL host variables are limited to 32,767 bytes.

Permissions

None.

Side effects

None.

See also

- "CREATE VARIABLE statement" on page 545
- "DECLARE statement" on page 551
- "DROP VARIABLE statement" on page 592
- "Expressions" on page 16

Standards and compatibility

- **SQL/2003** Persistent Stored Module feature.

Example

This simple example shows the creation of a variable called 'birthday', and uses SET to set the date to CURRENT DATE.

```
CREATE VARIABLE @birthday DATE;
    SET @birthday = CURRENT DATE;
```

The following code fragment inserts a large text value into the database.

```
EXEC SQL BEGIN DECLARE SECTION;
DECL_VARCHAR( 500 ) buffer;
/* Note: maximum DECL_VARCHAR size is 32765 */
EXEC SQL END DECLARE SECTION;

EXEC SQL CREATE VARIABLE hold_blob LONG VARCHAR;
EXEC SQL SET hold_blob = '';
for(;;) {
   /* read some data into buffer ... */
   size = fread( buffer, 1, 5000, fp );
   if( size <= 0 ) break;
   /* Does not work if data contains null chars */
   EXEC SQL SET hold_blob = hold_blob || :buffer;
}
EXEC SQL INSERT INTO some_table VALUES( 1, hold_blob );
EXEC SQL DROP VARIABLE hold_blob;
```

The following code fragment inserts a large binary value into the database.

```
EXEC SQL BEGIN DECLARE SECTION;
DECL_BINARY( 5000 ) buffer;
EXEC SQL END DECLARE SECTION;

EXEC SQL CREATE VARIABLE hold_blob LONG BINARY;
EXEC SQL SET hold_blob = '';
for(;;) {
   /* read some data into buffer ... */
   size = fread( &(buffer.array), 1, 5000, fp );
   if( size <= 0 ) break;
   buffer.len = size;
   /* add data to blob using concatenation */
   EXEC SQL SET hold_blob = hold_blob || :buffer;
}
EXEC SQL INSERT INTO some_table VALUES ( 1, hold_blob );
EXEC SQL DROP VARIABLE hold_blob;
```

SET statement [T-SQL]

Use this statement to set database options for the current connection in an Adaptive Server Enterprise-compatible manner.

Syntax

SET *option-name option-value*

Remarks

The available options are as follows:

Option name	Option value
ansinull	On or Off
ansi_permissions	On or Off
close_on_endtrans	On or Off
datefirst	1, 2, 3, 4, 5, 6, or 7 The setting of this option affects the DATEPART function when obtaining a weekday value. For more information about specifying the first day of the week, see "first_day_of_week option [database]" [*SQL Anywhere Server - Database Administration*] and "DATEPART function [Date and time]" on page 177.
quoted_identifier	On | Off
rowcount	*integer*
self_recursion	On | Off
string_rtruncation	On | Off
textsize	*integer*

Option name	Option value
transaction isolation level	0, 1, 2, 3, snapshot, statement snapshot, or read only statement snapshot

Database options in SQL Anywhere are set using the SET OPTION statement. However, SQL Anywhere also provides support for the Adaptive Server Enterprise SET statement for options that are useful for compatibility.

The following options can be set using the Transact-SQL SET statement in SQL Anywhere and Adaptive Server Enterprise:

- **SET ansinull { On | Off }** The default behavior for comparing values to NULL is different in SQL Anywhere and Adaptive Server Enterprise. Setting ansinull to Off provides Transact-SQL compatible comparisons with NULL.

  SQL Anywhere also supports the following syntax:

  SET ansi_nulls { **On** | **Off** }

  For more information, see "ansinull option [compatibility]" [*SQL Anywhere Server - Database Administration*].

- **SET ansi_permissions { On | Off }** The default behavior is different in SQL Anywhere and Adaptive Server Enterprise regarding permissions required to carry out an UPDATE or DELETE containing a column reference. Setting ansi_permissions to Off provides Transact-SQL-compatible permissions on UPDATE and DELETE. See "ansi_permissions option [compatibility]" [*SQL Anywhere Server - Database Administration*].

- **SET close_on_endtrans { On | Off }** The default behavior is different in SQL Anywhere and Adaptive Server Enterprise for closing cursors at the end of a transaction. Setting close_on_endtrans to Off provides Transact-SQL compatible behavior. See "close_on_endtrans option [compatibility]" [*SQL Anywhere Server - Database Administration*].

- **SET datefirst { 1 | 2 | 3 | 4 | 5 | 6 | 7 }** The default is 7, which means that the first day of the week is by default Sunday. To set this option permanently, see "first_day_of_week option [database]" [*SQL Anywhere Server - Database Administration*].

- **SET quoted_identifier { On | Off }** Controls whether strings enclosed in double quotes are interpreted as identifiers (On) or as literal strings (Off). See "Setting options for Transact-SQL compatibility" [*SQL Anywhere Server - SQL Usage*] and "quoted_identifier option [compatibility]" [*SQL Anywhere Server - Database Administration*].

- **SET rowcount** *integer* The Transact-SQL ROWCOUNT option limits the number of rows fetched for any cursor to the specified integer. This includes rows fetched by re-positioning the cursor. Any fetches beyond this maximum return a warning. The option setting is considered when returning the estimate of the number of rows for a cursor on an OPEN request.

  SET ROWCOUNT also limits the number of rows affected by a searched UPDATE or DELETE statement to *integer*. This might be used, for example, to allow COMMIT statements to be performed at regular intervals to limit the size of the rollback log and lock table. The application (or procedure) would need to provide a loop to cause the update/delete to be re-issued for rows that are not affected by the first operation. A simple example is given below:

```
BEGIN
   DECLARE @count INTEGER
   SET rowcount 20
   WHILE(1=1) BEGIN
      UPDATE Employees SET Surname='new_name'
      WHERE Surname <> 'old_name'
      /* Stop when no rows changed */
      SELECT @count = @@rowcount
      IF @count = 0 BREAK
      PRINT string('Updated ',
               @count,' rows; repeating...')
      COMMIT
   END
   SET rowcount 0
END
```

In SQL Anywhere, if the ROWCOUNT setting is greater than the number of rows that Interactive SQL can display, Interactive SQL may do some extra fetches to reposition the cursor. So, the number of rows actually displayed may be less than the number requested. Also, if any rows are re-fetched due to truncation warnings, the count may be inaccurate.

A value of zero resets the option to get all rows.

- **SET self_recursion { On | Off }** The self_recursion option is used within triggers to enable (On) or prevent (Off) operations on the table associated with the trigger from firing other triggers.
- **SET string_rtruncation { On | Off }** The default behavior is different between SQL Anywhere and Adaptive Server Enterprise when non-space characters are truncated during assignment of SQL string data. Setting string_rtruncation to On provides Transact-SQL-compatible string comparisons. See "string_rtruncation option [compatibility]" [*SQL Anywhere Server - Database Administration*].
- **SET textsize** Specifies the maximum size (in bytes) of text or image type data to be returned with a select statement. The @@textsize global variable stores the current setting. To reset to the default size (32 KB), use the command:

  ```
  set textsize 0
  ```

- **SET transaction isolation level { 0 | 1 | 2 | 3 | snapshot | statement snapshot | read only statement snapshot }** Sets the locking isolation level for the current connection, as described in "Isolation levels and consistency" [*SQL Anywhere Server - SQL Usage*]. For Adaptive Server Enterprise, only 1 and 3 are valid options. For SQL Anywhere, any of 0, 1, 2, 3, snapshot, statement snapshot, and read only statement snapshot is a valid option. See "isolation_level option [database] [compatibility]" [*SQL Anywhere Server - Database Administration*].

The SET statement is allowed by SQL Anywhere for the prefetch option, for compatibility, but has no effect.

Permissions

None.

Side effects

None.

See also

- "SET OPTION statement" on page 742
- "Setting options for Transact-SQL compatibility" [*SQL Anywhere Server - SQL Usage*]
- "Compatibility options" [*SQL Anywhere Server - Database Administration*]

Standards and compatibility

- **SQL/2003** Transact-SQL extension.

SET CONNECTION statement [Interactive SQL] [ESQL]

Use this statement to change the active database connection.

Syntax

SET CONNECTION [*connection-name*]

connection-name : *identifier*, *string*, or *hostvar*

Remarks

The SET CONNECTION statement changes the active database connection to connection-name. The current connection state is saved, and is resumed when it again becomes the active connection. If connection-name is omitted and there is a connection that was not named, that connection becomes the active connection.

When cursors are opened in embedded SQL, they are associated with the current connection. When the connection is changed, the cursor names of the previously active connection become inaccessible. These cursors remain active and in position, and become accessible when the associated connection becomes active again.

Permissions

None.

Side effects

None.

See also

- "CONNECT statement [ESQL] [Interactive SQL]" on page 429
- "DISCONNECT statement [ESQL] [Interactive SQL]" on page 569
- "Using Interactive SQL" [*SQL Anywhere Server - Database Administration*]

Standards and compatibility

- **SQL/2003** Interactive SQL is a vendor extension. Embedded SQL is a core feature.

Example

The following example is in embedded SQL.

```
EXEC SQL SET CONNECTION :conn_name;
```

From Interactive SQL, set the current connection to the connection named conn1.

```
SET CONNECTION conn1;
```

SET DESCRIPTOR statement [ESQL]

Use this statement to describe the variables in a SQL descriptor area and to place data into the descriptor area.

Syntax

SET DESCRIPTOR *descriptor-name*
{ **COUNT** = { *integer* | *hostvar* }
| **VALUE** { *integer* | *hostvar* } *assignment*, ... }

assignment :
{ **TYPE** | **SCALE** | **PRECISION** | **LENGTH** | **INDICATOR** }
    = { *integer* | *hostvar* }
| **DATA** = *hostvar*

descriptor-name : identifier

Remarks

The SET DESCRIPTOR statement is used to describe the variables in a descriptor area, and to place data into the descriptor area.

The SET ... COUNT statement sets the number of described variables within the descriptor area. The value for count must not exceed the number of variables specified when the descriptor area was allocated.

The value { *integer* | *hostvar* } specifies the variable in the descriptor area upon which the assignment(s) is performed.

Type checking is performed when doing SET ... DATA, to ensure that the variable in the descriptor area has the same type as the host variable. LONG VARCHAR and LONG BINARY are not supported by SET DESCRIPTOR ... DATA.

If an error occurs, the code is returned in the SQLCA.

Permissions

None.

Side effects

None.

See also

- “ALLOCATE DESCRIPTOR statement [ESQL]” on page 357
- “DEALLOCATE DESCRIPTOR statement [ESQL]” on page 550
- “The SQL descriptor area (SQLDA)” [*SQL Anywhere Server - Programming*]

Standards and compatibility

- **SQL/2003** SQL/foundation feature outside core SQL.

Example

The following example sets the type of the column with position col_num in sqlda.

```
void set_type( SQLDA *sqlda, int col_num, int new_type )
{
    EXEC SQL BEGIN DECLARE SECTION;
    INT new_type1 = new_type;
    INT col = col_num;
    EXEC SQL END DECLARE SECTION;
    EXEC SQL SET DESCRIPTOR sqlda VALUE :col TYPE = :new_type1;
}
```

For a longer example, see "ALLOCATE DESCRIPTOR statement [ESQL]" on page 357.

SET OPTION statement

Use this statement to change the values of database options.

Syntax

SET [**EXISTING**] [**TEMPORARY**] **OPTION**
[*userid*.| **PUBLIC**.]*option-name* = [*option-value*]

userid : *identifier*, *string*, or *hostvar*

option-name : *identifier*

option-value : *string literal*

Embedded SQL syntax

SET [**TEMPORARY**] **OPTION**
[*userid*.| **PUBLIC**.]*option-name* = [*option-value*]

userid : *identifier*, *string*, or *hostvar*

option-name : *identifier*, *string*, or *hostvar*

option-value : *hostvar* (indicator allowed), *string*, *identifier*, or *number*

Remarks

The SET OPTION statement is used to change options that affect the behavior of the database server. Setting the value of an option can change the behavior for all users or only for an individual user. The scope of the change can be either temporary or permanent.

Any option, whether user-defined or not, must have a public setting before a user-specific value can be assigned. The database server does not support setting TEMPORARY values for user-defined options.

The classes of options are:

- database options
- Transact-SQL compatibility options
- replication database options

For a listing and description of all available options, see "Database options" [*SQL Anywhere Server - Database Administration*].

You can set options at three levels of scope: public, user, and temporary. A temporary option takes precedence over other options, and user options take precedence over public options. If you set a user level option for the current user, the corresponding temporary option gets set as well.

Syntax 1 does not allow you to specify a host variable for *option-value*. However, you may be able to achieve the outcome you want using the EXECUTE IMMEDIATE statement instead. See "EXECUTE IMMEDIATE statement [SP]" on page 600.

If you use the EXISTING keyword, option values cannot be set for an individual user ID unless there is already a PUBLIC user ID setting for that option.

If you specify a user ID, the option value applies to that user (or, for a group user ID, the members of that group). If you specify PUBLIC, the option value applies to all users who do not have an individual setting for the option. By default, the option value applies to the currently logged on user ID that issued the SET OPTION statement.

For example, the following statement applies an option change to the user DBA, if DBA is the user issuing the SQL statement:

```
SET OPTION precision = 40;
```

However the following statement applies the change to the PUBLIC user ID, a user group to which all users belong.

```
SET OPTION Public.login_mode = Standard;
```

Only users with DBA privileges have the authority to set an option for the PUBLIC user ID.

Users can use the SET OPTION statement to change the values for their own user ID. Setting the value of an option for a user ID other then your own is permitted only if you have DBA authority.

Adding the TEMPORARY keyword to the SET OPTION statement changes the duration that the change takes effect. By default, the option value is permanent; it will not change until it is explicitly changed using the SET OPTION statement.

When the SET TEMPORARY OPTION statement is not qualified with a user ID, the new option value is in effect only for the current connection.

When SET TEMPORARY OPTION is used for the PUBLIC user ID, the change is in place for as long as the database is running. When the database is shut down, TEMPORARY options for the PUBLIC group revert back to their permanent value.

Setting temporary options for the PUBLIC user ID offers a security benefit. For example, when the login_mode option is enabled, the database relies on the login security of the system on which it is running. Enabling it temporarily means that a database relying on the security of a Windows domain will not be compromised if the database is shut down and copied to a local computer. In that case, the temporary enabling of the login_mode option reverts to its permanent value, which could be Standard, a mode where integrated logins are not permitted.

If *option-value* is omitted, the specified option setting is deleted from the database. If it was a personal option setting, the value reverts back to the PUBLIC setting. If a TEMPORARY option is deleted, the option setting reverts back to the permanent setting.

Caution
Changing option settings while fetching rows from a cursor is not supported, as it can lead to ill-defined behavior. For example, changing the date_format setting while fetching from a cursor would lead to different date formats among the rows in the result set. Do not change option settings while fetching rows.

The SET OPTION statement is ignored by the SQL Flagger.

Permissions

None required to set your own options.

DBA authority is required to set database options for another user or PUBLIC.

Side effects

If TEMPORARY is not specified, an automatic commit is performed.

See also

- "Database options" [*SQL Anywhere Server - Database Administration*]
- "Compatibility options" [*SQL Anywhere Server - Database Administration*]
- "SQL Remote options" [*SQL Anywhere Server - Database Administration*]
- "SET OPTION statement [Interactive SQL]" on page 745

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

Set the date format option to on:

```
SET OPTION public.date_format = 'Mmm dd yyyy';
```

Set the wait_for_commit option to On:

```
SET OPTION wait_for_commit = 'On';
```

Following are two embedded SQL examples.

```
1. EXEC SQL SET OPTION :user.:option_name = :value;
2. EXEC SQL SET TEMPORARY OPTION date_format = 'mm/dd/yyyy';
```

Set the date_format option for the user that is currently connected. Future connections for the same user ID use this option value.

```
SET OPTION date_format = 'yyyy/mm/dd';
```

The following statement removes the setting of the date_format option for the current user ID. After executing this statement, the date_format setting for PUBLIC is used instead.

```
SET OPTION date_format=;
```

SET OPTION statement [Interactive SQL]

Use this statement to change the values of Interactive SQL options.

Syntax 1 - Set an option

SET OPTION *option-name* = [*option-value*]
| **SET TEMPORARY OPTION** *option-name* = [*option-value*]

option-name : *identifier*, *string*, or *hostvar*

option-value : *string*, *identifier*, or *number*

Syntax 2 - Save current options permanently

SET PERMANENT

Syntax 3 - Review current options

SET

Remarks

By default, when you set an option using the SET OPTION syntax, the option setting is stored permanently and does not change unless another SET OPTION statement changes it.

Using the SET TEMPORARY OPTION syntax allows you to temporarily change an option setting. The temporary setting remains in effect until you close Interactive SQL. The next time you start Interactive SQL, the option reverts to its permanent setting.

Use the SET PERMANENT syntax to permanently save all current Interactive SQL option settings (any temporary settings become permanent).

Use Syntax 3 to display all the current database option settings. If there are temporary options settings for the database server, these are displayed instead of the permanent settings.

Interactive SQL option settings are stored on the client computer, not in the database.

For descriptions of the current Interactive SQL options, see "Interactive SQL options" [*SQL Anywhere Server - Database Administration*].

Standards and compatibility

- **SQL/2003** Vendor extension.

See also

- "Interactive SQL options" [*SQL Anywhere Server - Database Administration*]
- "Using Interactive SQL" [*SQL Anywhere Server - Database Administration*]

SET REMOTE OPTION statement [SQL Remote]

Use this statement to set a message control parameter for a SQL Remote message link.

Syntax

SET REMOTE *link-name* **OPTION**
[*userid*.| **PUBLIC**.]*link-option-name* = *link-option-value*

link-name:
file
| **ftp**
| **smtp**

link-option-name:
common-option
| *file-option*
| *ftp-option*
| *smtp-option*

common-option:
debug
| **output_log_send_on_error**
| **output_log_send_limit**
| **output_log_send_now**

file-option:
directory
| **invalid_extensions**
| **unlink_delay**

ftp-option:
active_mode
| **host**
| **invalid_extensions**
| **password**
| **port**
| **root_directory**
| **user**
| **reconnect_retries**
| **reconnect_pause**

smtp-option:
local_host
| **pop3_host**
| **pop3_password**
| **pop3_userid**
| **smtp_host**
| **top_supported**

link-option-value : *string*

Parameters

- **userid** If you do not specify a *userid*, then the current publisher is assumed.

- **options** The option values are message-link dependent. For more information, see:
  - "The FILE message system" [*SQL Remote*]
  - "The FTP message system" [*SQL Remote*]
  - "The SMTP message system" [*SQL Remote*]

Remarks

The Message Agent saves message link parameters when the user enters them in the message link window when the message link is first used. In this case, it is not necessary to use this statement explicitly. This statement is most useful when preparing a consolidated database for extracting many databases.

The option names are case sensitive. The case sensitivity of option values depends on the option: Boolean values are case insensitive, while the case sensitivity of passwords, directory names, and other strings depend on the cases sensitivity of the file system (for directory names), or the database (for user IDs and passwords).

Permissions

Must have DBA authority. The publisher can set their own options.

Side effects

Automatic commit.

See also

- "Collect errors from the remote database" [*SQL Remote*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Examples

The following statement sets the FTP host to *ftp.mycompany.com* for the FTP link for user myuser:

```
SET REMOTE FTP OPTION myuser.host = 'ftp.mycompany.com';
```

The following statement stops SQL Remote from using the specified file extensions for messages that are generated:

```
SET REMOTE ftp OPTION "Public"."invalid_extensions" =
'exe,pif,dll,bat,cmd,vbs';
```

SET SQLCA statement [ESQL]

Use this statement to tell the SQL preprocessor to use a SQLCA other than the default, global *sqlca*.

Syntax

SET SQLCA *sqlca*

sqlca : *identifier* or *string*

Remarks

The SET SQLCA statement tells the SQL preprocessor to use a SQLCA other than the default global *sqlca*. The sqlca must be an identifier or string that is a C language reference to a SQLCA pointer.

The current SQLCA pointer is implicitly passed to the database interface library on every embedded SQL statement. All embedded SQL statements that follow this statement in the C source file will use the new SQLCA.

This statement is necessary only when you are writing code that is reentrant (see "SQLCA management for multi-threaded or reentrant code" [*SQL Anywhere Server - Programming*]). The sqlca should reference a local variable. Any global or module static variable is subject to being modified by another thread.

Permissions

None.

Side effects

None.

See also

- "SQLCA management for multi-threaded or reentrant code" [*SQL Anywhere Server - Programming*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The owning function could be found in a Windows DLL. Each application that uses the DLL has its own SQLCA.

```
an_sql_code FAR PASCAL ExecuteSQL( an_application *app, char *com )
{
   EXEC SQL BEGIN DECLARE SECTION;
   char *sqlcommand;
   EXEC SQL END DECLARE SECTION;
   EXEC SQL SET SQLCA "&app->.sqlca";
   sqlcommand = com;
   EXEC SQL WHENEVER SQLERROR CONTINUE;
   EXEC SQL EXECUTE IMMEDIATE :sqlcommand;
return( SQLCODE );
}
```

SETUSER statement

Use this statement to allow a database administrator to impersonate another user, and to enable connection pooling.

Syntax

{ **SET SESSION AUTHORIZATION** | **SETUSER** }
[[**WITH OPTION**] *userid*]

Parameters

- **WITH OPTION clause** By default, only permissions (including group membership) are altered. If WITH OPTION is specified, the database options in effect are changed to the current database options of *userid*.
- **userid** The user ID is an identifier (SETUSER syntax) or a string (SET SESSION AUTHORIZATION syntax). See "Identifiers" on page 8 and "Strings" on page 9.

Remarks

The SETUSER statement is provided to make database administration easier. It enables a user with DBA authority to impersonate another user of the database. After running a SETUSER statement, you can check which user you are impersonating by running one of the following commands:

- SELECT USER
- SELECT CURRENT USER

SETUSER can also be used from an application server to take advantage of connection pooling. Connection pooling cuts down the number of distinct connections that need to be made, which can improve performance.

SETUSER with no user ID undoes all earlier SETUSER statements.

The SETUSER statement cannot be used inside a procedure, trigger, event handler or batch.

There are several uses for the SETUSER statement, including the following:

- **Creating objects** You can use SETUSER to create a database object that is to be owned by another user.
- **Permissions checking** By acting as another user, with their permissions and group memberships, a database administrator can test the permissions and name resolution of queries, procedures, views, and so on.
- **Providing a safer environment for administrators** The database administrator has permission to carry out any action in the database. If you want to ensure that you do not accidentally carry out an unintended action, you can use SETUSER to switch to a different user ID with fewer permissions.

Note
The SETUSER statement cannot be used within procedures, triggers, events, or batches.

Permissions

Must have DBA authority.

See also

- "EXECUTE IMMEDIATE statement [SP]" on page 600
- "GRANT statement" on page 627
- "REVOKE statement" on page 718
- "SET OPTION statement" on page 742

Standards and compatibility

- **SQL/2003** SET SESSION AUTHORIZATION is a core feature. SETUSER is a vendor extension.

Example

The following statements, executed by a user named DBA, change the user ID to be Joe, then Jane, and then back to DBA.

```
SETUSER "Joe"
// ... operations...
SETUSER WITH OPTION "Jane"
// ... operations...
SETUSER
```

The following statement sets the user to Jane. The user ID is supplied as a string rather than as an identifier.

```
SET SESSION AUTHORIZATION 'Jane';
```

SIGNAL statement

Use this statement to signal an exception condition.

Syntax

SIGNAL *exception-name*

Remarks

SIGNAL allows you to raise an exception. For a description of how exceptions are handled, see "Using exception handlers in procedures and triggers" [*SQL Anywhere Server - SQL Usage*].

Use *exception-name* to specify the name of an exception declared using a DECLARE statement at the beginning of the current compound statement. The exception must correspond to a system-defined SQLSTATE or a user-defined SQLSTATE. User-defined SQLSTATE values must be in the range 99000 to 99999.

Permissions

None.

Side effects

None.

See also

- "RESIGNAL statement" on page 713
- "BEGIN statement" on page 413
- "Using exception handlers in procedures and triggers" [*SQL Anywhere Server - SQL Usage*]

Standards and compatibility

- **SQL/2003** Persistent Stored Module feature.

Example

The following compound statement declares and signals a user-defined exception. If you execute this example from Interactive SQL, the message **My exception signaled** appears on the **Messages** tab in the **Results** area.

```
BEGIN
   DECLARE myexception EXCEPTION
   FOR SQLSTATE '99001';
   SIGNAL myexception;
   EXCEPTION
      WHEN myexception THEN
         MESSAGE 'My exception signaled'
         TO CLIENT;
END
```

START DATABASE statement

Use this statement to start a database on the current database server.

Syntax

START DATABASE *database-file* [*start-options* ...]

start-options :
[**AS** *database-name*]
[**ON** *database-server-name*]
[**WITH TRUNCATE AT CHECKPOINT**]
[**FOR READ ONLY**]
[**AUTOSTOP** { **ON** | **OFF** }]
[**KEY** *key*]
[**WITH SERVER NAME** *alternative-database-server-name*]
[**DIRECTORY** *dbspace-directory*]

Parameters

- **database-file** The *database-file* parameter is a string. If a relative path is supplied in *database-file*, it is relative to the database server starting directory.
- **start-options clauses** The *start-options* can be listed in any order:
  - **AS clause** If *database-name* is not specified, a default name is assigned to the database. This default name is the root of the database file. For example, a database in file *C:\Database Files \demo.db* would be given the default name of demo. The *database-name* parameter is an identifier.
  - **ON clause** This clause is supported from Interactive SQL only. In Interactive SQL, if *server-name* is not specified, the default server is the first started server among those currently running. The *server-name* parameter is an identifier.
  - **WITH TRUNCATE AT CHECKPOINT clause** Starts a database with log truncation on checkpoint enabled.
  - **FOR READ ONLY clause** Starts a database in read-only mode. When used on a database requiring recovery, the statement fails and the error message is returned.

- **AUTOSTOP clause** The default setting for the AUTOSTOP clause is ON. With AUTOSTOP set to ON, the database is unloaded when the last connection to it is dropped. If AUTOSTOP is set to OFF, the database is not unloaded.

  In Interactive SQL, you can use YES or NO as alternatives to ON and OFF.

- **KEY clause** If the database is strongly encrypted, enter the KEY value (password) using this clause

- **WITH SERVER NAME clause** Use this clause to specify an alternate name for the database server when connecting to this database. If you are using database mirroring, the primary and mirror servers must both have the same database server name because clients do not know to which server they are connecting.

  For more information about alternate server names and database mirroring, see "-sn database option" [*SQL Anywhere Server - Database Administration*] and "Introduction to database mirroring" [*SQL Anywhere Server - Database Administration*].

- **DIRECTORY clause** Use this clause to specify the directory where the dbspace files are located for the database that is being started. For example, if the database server is started in the same directory as the dbspaces, and you include the `DIRECTORY '.'` clause, then this instructs the database server to find all dbspaces in the current directory. See "-ds database option" [*SQL Anywhere Server - Database Administration*].

Remarks

Starts a specified database on the current database server.

If you are not connected to a database and you want to use the START DATABASE statement, you must first connect to a database, such as the utility database.

For information about the utility database, see "Using the utility database" [*SQL Anywhere Server - Database Administration*].

The START DATABASE statement does not connect the current application to the specified database: an explicit connection is still needed.

Interactive SQL supports the ON clause, which allows the database to be started on a database server other than the current.

You can only use the database name utility_db to connect to the SQL Anywhere utility database. See "Using the utility database" [*SQL Anywhere Server - Database Administration*].

Permissions

The required permissions are specified by the database server -gd option. This option defaults to all on the personal database server, and DBA on the network server.

Side effects

None

See also

- “STOP DATABASE statement” on page 759
- “CONNECT statement [ESQL] [Interactive SQL]” on page 429
- “-gd server option” [*SQL Anywhere Server - Database Administration*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

Start the database file *C:\Database Files\sample_2.db* on the current server.

```
START DATABASE 'c:\database files\sample_2.db';
```

From Interactive SQL, start the database file *c:\Database Files\sample_2.db* as sam2 on the server named sample.

```
START DATABASE 'c:\database files\sample_2.db'
AS sam2
ON sample;
```

START ENGINE statement [Interactive SQL]

Use this statement to start a database server.

Syntax

START ENGINE AS *database-server-name* [**STARTLINE** *command-string*]

Remarks

The START ENGINE statement starts a database server. If you want to specify a set of options for the database server, use the STARTLINE keyword together with a command string. Valid command strings are those that conform to the database server description in “The SQL Anywhere database server” [*SQL Anywhere Server - Database Administration*].

Permissions

None

Side effects

None

See also

- “STOP ENGINE statement” on page 760
- “The SQL Anywhere database server” [*SQL Anywhere Server - Database Administration*]
- “Using Interactive SQL” [*SQL Anywhere Server - Database Administration*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

Start a database server, named sample, without starting any databases on it.

```
START ENGINE AS sample;
```

The following example shows the use of a STARTLINE clause.

```
START ENGINE AS eng1 STARTLINE 'dbeng11 -c 8M';
```

START EXTERNAL ENVIRONMENT statement

Use this statement to start an external environment.

Syntax

START EXTERNAL ENVIRONMENT *environment-name*

environment-name :
JAVA
| **PERL**
| **PHP**
| **CLR**
| **C_ESQL32**
| **C_ESQL64**
| **C_ODBC32**
| **C_ODBC64**

Parameters

- **environment-name** The name of the external environment to start.

Remarks

For more information about external environments, see "Overview of external environments" [*SQL Anywhere Server - Programming*].

Permissions

Must have DBA authority.

Side effects

None

See also

- "Overview of external environments" [*SQL Anywhere Server - Programming*]
- "ALTER EXTERNAL ENVIRONMENT statement" on page 367
- "STOP EXTERNAL ENVIRONMENT statement" on page 761
- "INSTALL EXTERNAL OBJECT statement" on page 654
- "REMOVE EXTERNAL OBJECT statement" on page 710
- "SYSEXTERNENV system view" on page 1000
- "SYSEXTERNENVOBJECT system view" on page 1000

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

Start the Perl external environment.

```
START EXTERNAL ENVIRONMENT PERL;
```

START JAVA statement

Use this statement to start the Java VM.

Syntax

START JAVA

Remarks

The START JAVA statement starts the Java VM. The main use is to load the Java VM at a convenient time so that when the user starts to use Java functionality there is no initial pause while the Java VM is loaded.

The database server must be set up to locate a Java VM. Since you can specify different Java VMs for each database, the java_location option can be used to indicate the location (path) of the Java VM. See "java_location option [database]" [*SQL Anywhere Server - Database Administration*].

For more information about starting the Java VM, see "Starting and stopping the Java VM" [*SQL Anywhere Server - Programming*].

Permissions

A Java VM must be installed, and the database must be Java-enabled.

This statement is not supported on Windows Mobile.

Side effects

None

See also

- "STOP JAVA statement" on page 762

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

Start the Java VM.

```
START JAVA;
```

START LOGGING statement [Interactive SQL]

Use this statement to start logging executed SQL statements to a log file.

Syntax

START LOGGING *filename*

Remarks

The START LOGGING statement starts copying all subsequent executed SQL statements to the log file that you specify. If the file does not exist, Interactive SQL creates it. Logging continues until you explicitly stop the logging process with the STOP LOGGING statement, or until you end the current Interactive SQL session. You can also start and stop logging by clicking SQL » Start Logging and SQL » Stop Logging.

Permissions

None.

Side effects

None.

See also

- "STOP LOGGING statement [Interactive SQL]" on page 762
- "Using Interactive SQL" [*SQL Anywhere Server - Database Administration*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

Start logging to a file called *filename.sql*, located in the c: directory.

```
START LOGGING 'c:\filename.sql';
```

START SUBSCRIPTION statement [SQL Remote]

Use this statement to start a subscription for a user to a publication.

Syntax

START SUBSCRIPTION
TO *publication-name* [**(** *subscription-value* **)**]
FOR *subscriber-id*, ...

Parameters

- **publication-name** The name of the publication to which the user is being subscribed. This may include the owner of the publication.

- **subscription-value** A string that is compared to the subscription expression of the publication. The value is required here because each subscriber may have more than one subscription to a publication.
- **subscriber-id** The user ID of the subscriber to the publication. This user must have a subscription to the publication.

Remarks

A SQL Remote subscription is said to be started when publication updates are being sent from the consolidated database to the remote database.

The START SUBSCRIPTION statement is one of a set of statements that manage subscriptions. The CREATE SUBSCRIPTION statement defines the data that the subscriber is to receive. The SYNCHRONIZE SUBSCRIPTION statement ensures that the consolidated and remote databases are consistent with each other. The START SUBSCRIPTION statement is required to start messages being sent to the subscriber.

Data at each end of the subscription must be consistent before a subscription is started. It is recommended that you use the database extraction utility to manage the creation, synchronization, and starting of subscriptions. If you use the database extraction utility, you do not need to execute an explicit START SUBSCRIPTION statement. Also, the Message Agent starts subscriptions once they are synchronized.

Permissions

Must have DBA authority.

Side effects

Automatic commit.

See also

- "CREATE SUBSCRIPTION statement [SQL Remote]" on page 517
- "REMOTE RESET statement [SQL Remote]" on page 709
- "SYNCHRONIZE SUBSCRIPTION statement [SQL Remote]" on page 765
- "STOP SUBSCRIPTION statement [SQL Remote]" on page 763
- "Extraction utility (dbxtract)" [*SQL Remote*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement starts the subscription of user SamS to the pub_contact publication.

```
START SUBSCRIPTION TO pub_contact
FOR SamS;
```

START SYNCHRONIZATION DELETE statement [MobiLink]

Use this statement to restart logging of deletes for MobiLink synchronization.

Syntax

START SYNCHRONIZATION DELETE

Remarks

Ordinarily, SQL Anywhere and UltraLite automatically log any changes made to tables or columns that are part of a synchronization, and upload these changes to the consolidated database during the next synchronization. You can temporarily suspend automatic logging of delete operations using the STOP SYNCHRONIZATION DELETE statement. The START SYNCHRONIZATION DELETE statement allows you to restart the automatic logging.

When a STOP SYNCHRONIZATION DELETE statement is executed, none of the delete operations executed on that connection are synchronized. The effect continues until a START SYNCHRONIZATION DELETE statement is executed. Repeating STOP SYNCHRONIZATION DELETE has no additional effect.

A single START SYNCHRONIZATION DELETE statement restarts the logging, regardless of the number of STOP SYNCHRONIZATION DELETE statements preceding it.

Do not use START SYNCHRONIZATION DELETE if your application does not synchronize data.

Permissions

Must have DBA authority.

Side effects

None.

See also

- "STOP SYNCHRONIZATION DELETE statement [MobiLink]" on page 764
- "StartSynchronizationDelete method" [*UltraLite - .NET Programming*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following sequence of SQL statements illustrates how to use START SYNCHRONIZATION DELETE and STOP SYNCHRONIZATION DELETE:

```
-- Prevent deletes from being sent
-- to the consolidated database
STOP SYNCHRONIZATION DELETE;

-- Remove all records older than 1 month
-- from the remote database,
-- NOT the consolidated database
DELETE FROM PROPOSAL
WHERE last_modified < months( CURRENT TIMESTAMP, -1 )

-- Re-enable all deletes to be sent
-- to the consolidated database
-- DO NOT FORGET to start this
START SYNCHRONIZATION DELETE;

-- Commit the entire operation,
-- otherwise rollback everything
-- including the stopping of the deletes
commit;
```

STOP DATABASE statement

Use this statement to stop a database on the current database server.

Syntax

STOP DATABASE *database-name*
[**ON** *database-server-name*]
[**UNCONDITIONALLY**]

Parameters

- **STOP DATABASE clause** The *database-name* is the name of a database (other than the current database) running on the current server.
- **ON clause** This clause is supported in Interactive SQL only. If *database-server-name* is not specified in Interactive SQL, all running servers are searched for a database of the specified name.

  When not using this statement in Interactive SQL, the database is stopped only if it is started on the current database server.
- **UNCONDITIONALLY clause** Stop the database even if there are connections to the database. By default, the database is not stopped if there are connections to it.

Remarks

The STOP DATABASE statement stops a specified database on the current database server.

Permissions

The required permissions are specified by the database server -gk option. This option defaults to all on the personal database server, and DBA on the network server.

You cannot use STOP DATABASE on the database to which you are currently connected.

Side effects

None

See also

- "START DATABASE statement" on page 751
- "DISCONNECT statement [ESQL] [Interactive SQL]" on page 569
- "-gd server option" [*SQL Anywhere Server - Database Administration*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

Stop the database named *sample* on the current server.

```
STOP DATABASE sample;
```

STOP ENGINE statement

Use this statement to stop a database server.

Syntax

STOP ENGINE [*database-server-name*] [**UNCONDITIONALLY**]

Parameters

- **STOP ENGINE clause** The *database-server-name* can be used in Interactive SQL only. If you are not running this statement in Interactive SQL, the current database server is stopped.
- **UNCONDITIONALLY clause** If you are the only connection to the database server, you do not need to use UNCONDITIONALLY. If there are other connections, the database server stops only if you use the UNCONDITIONALLY keyword.

Remarks

The STOP ENGINE statement stops the specified database server. If the UNCONDITIONALLY keyword is supplied, the database server is stopped even if there are other connections to the database server. By default, the database server will not be stopped if there are other connections to it.

The STOP ENGINE statement cannot be used in stored procedures, triggers, events, or batches.

Permissions

The permissions to shut down a server depend on the -gk setting on the database server command line. The default setting is all for the personal server, and DBA for the network server.

Side effects

None

See also

- "START ENGINE statement [Interactive SQL]" on page 753
- "-gk server option" [*SQL Anywhere Server - Database Administration*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

Stop the current database server, as long as there are no other connections.

```
STOP ENGINE;
```

STOP EXTERNAL ENVIRONMENT statement

Use this statement to stop an external environment.

Syntax

STOP EXTERNAL ENVIRONMENT *environment-name*

environment-name :
JAVA
| PERL
| PHP
| CLR
| C_ESQL32
| C_ESQL64
| C_ODBC32
| C_ODBC64

Parameters

- **environment-name** The name of the external environment to stop.

Remarks

For more information about external environments, see "Overview of external environments" [*SQL Anywhere Server - Programming*].

Permissions

Must have DBA authority.

Side effects

None

See also

- "Overview of external environments" [*SQL Anywhere Server - Programming*]
- "ALTER EXTERNAL ENVIRONMENT statement" on page 367
- "START EXTERNAL ENVIRONMENT statement" on page 754
- "INSTALL EXTERNAL OBJECT statement" on page 654
- "REMOVE EXTERNAL OBJECT statement" on page 710
- "SYSEXTERNENV system view" on page 1000
- "SYSEXTERNENVOBJECT system view" on page 1000

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

Stop the Perl external environment.

```
STOP EXTERNAL ENVIRONMENT PERL;
```

STOP JAVA statement

Use this statement to stop the Java VM.

Syntax

STOP JAVA

Remarks

The STOP JAVA statement unloads the Java VM when it is not in use. The main use is to economize on the use of system resources.

Permissions

This statement is not supported on Windows Mobile.

Side effects

None

See also

- "START JAVA statement" on page 755

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

Stop the Java VM.

```
STOP JAVA;
```

STOP LOGGING statement [Interactive SQL]

Use this statement to stop logging of SQL statements in the current session.

Syntax

STOP LOGGING

Remarks

The STOP LOGGING statement stops Interactive SQL from writing each SQL statement you execute to a log file. You can start logging with the START LOGGING statement. You can also start and stop logging by clicking SQL » Start Logging and SQL » Stop Logging.

Permissions

None.

Side effects

None.

See also

- “START LOGGING statement [Interactive SQL]” on page 756
- “Using Interactive SQL” [*SQL Anywhere Server - Database Administration*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following example stops the current logging session.

```
STOP LOGGING;
```

STOP SUBSCRIPTION statement [SQL Remote]

Use this statement to stop a subscription for a user to a publication.

Syntax

STOP SUBSCRIPTION
TO *publication-name* [**(** *subscription-value* **)**]
FOR *subscriber-id*, ...

Parameters

- **publication-name** The name of the publication to which the user is being subscribed. This may include the owner of the publication.
- **subscription-value** A string that is compared to the subscription expression of the publication. The value is required here because each subscriber may have more than one subscription to a publication.
- **subscriber-id** The user ID of the subscriber to the publication. This user must have a subscription to the publication.

Remarks

A SQL Remote subscription is said to be started when publication updates are being sent from the consolidated database to the remote database.

The STOP SUBSCRIPTION statement prevents any further messages being sent to the subscriber. The START SUBSCRIPTION statement is required to restart messages being sent to the subscriber. However, you should ensure that the subscription is properly synchronized before restarting: that no messages have been missed.

Permissions

Must have DBA authority.

Side effects

Automatic commit.

See also

- “DROP SUBSCRIPTION statement [SQL Remote]” on page 584
- “START SUBSCRIPTION statement [SQL Remote]” on page 756
- “SYNCHRONIZE SUBSCRIPTION statement [SQL Remote]” on page 765
- “Extraction utility (dbxtract)” [*SQL Remote*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement stops the subscription of user SamS to the pub_contact publication.

```
STOP SUBSCRIPTION TO pub_contact
FOR SamS;
```

STOP SYNCHRONIZATION DELETE statement [MobiLink]

Use this statement to temporarily stop logging of deletes for MobiLink synchronization.

Syntax

STOP SYNCHRONIZATION DELETE

Remarks

Ordinarily, SQL Anywhere and UltraLite remote databases automatically log any changes made to tables or columns that are included in a synchronization, and then upload these changes to the consolidated database during the next synchronization. This statement allows you to temporarily suspend logging of delete operations to a SQL Anywhere or UltraLite remote database.

When a STOP SYNCHRONIZATION DELETE statement is executed, none of the subsequent delete operations executed on that connection are synchronized. The effect continues until a START SYNCHRONIZATION DELETE statement is executed.

Repeating STOP SYNCHRONIZATION DELETE has no additional effect. A single START SYNCHRONIZATION DELETE statement restarts the logging, regardless of the number of STOP SYNCHRONIZATION DELETE statements preceding it.

This statement can be useful to make corrections to a remote database, but should be used with caution as it effectively disables MobiLink synchronization.

Do not use STOP SYNCHRONIZATION DELETE if your application does not synchronize data.

Permissions

Must have DBA authority.

Side Effects

None.

See also

- UltraLite "StartSynchronizationDelete method" [*UltraLite - .NET Programming*]
- UltraLite "StopSynchronizationDelete method" [*UltraLite - .NET Programming*]
- "START SYNCHRONIZATION DELETE statement [MobiLink]" on page 757

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

For an example, see "START SYNCHRONIZATION DELETE statement [MobiLink]" on page 757.

SYNCHRONIZE SUBSCRIPTION statement [SQL Remote]

Use this statement to synchronize a subscription for a user to a publication.

Syntax

SYNCHRONIZE SUBSCRIPTION
TO *publication-name* [**(** *subscription-value* **)**]
FOR *remote-user*, ...

Parameters

- **publication-name** The name of the publication to which the user is being subscribed. This may include the owner of the publication.
- **subscription-value** A string that is compared to the subscription expression of the publication. The value is required here because each subscriber may have more than one subscription to a publication.
- **remote-user** The user ID of the subscriber to the publication. This user must have a subscription to the publication.

Remarks

A SQL Remote subscription is said to be **synchronized** when the data in the remote database is consistent with that in the consolidated database, so that publication updates sent from the consolidated database to the remote database will not result in conflicts and errors.

To synchronize a subscription, a copy of the data in the publication at the consolidated database is sent to the remote database. The SYNCHRONIZE SUBSCRIPTION statement does this through the message system. It is recommended that where possible you use the database extraction utility (dbxtract) instead to synchronize subscriptions without using a message system.

Permissions

Must have DBA authority.

Side effects

Automatic commit.

See also

- "CREATE SUBSCRIPTION statement [SQL Remote]" on page 517
- "START SUBSCRIPTION statement [SQL Remote]" on page 756
- "STOP SUBSCRIPTION statement [SQL Remote]" on page 763
- "Extraction utility (dbxtract)" [*SQL Remote*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement synchronizes the subscription of user SamS to the pub_contact publication.

```
SYNCHRONIZE SUBSCRIPTION
   TO pub_contact
   FOR SamS;
```

SYSTEM statement [Interactive SQL]

Use this statement to launch an executable file from within Interactive SQL.

Syntax

SYSTEM *'* [*path*] *filename '*

Remarks

Launches the specified executable file.

- The SYSTEM statement must be entirely contained on one line.
- Comments are not allowed at the end of a SYSTEM statement.
- Enclose the path and file name in single quotation marks.

Permissions

None.

Side effects

None.

See also

- "CONNECT statement [ESQL] [Interactive SQL]" on page 429
- "Using Interactive SQL" [*SQL Anywhere Server - Database Administration*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following statement launches the Notepad program, assuming that the Notepad executable is in your path.

```
SYSTEM 'notepad.exe';
```

TRIGGER EVENT statement

Use this statement to trigger a named event. The event may be defined for event triggers or be a scheduled event.

Syntax

TRIGGER EVENT *event-name* [**(** *parm* = *value*, ... **)**]

Parameters

- **parm = value** When a triggering condition causes an event handler to execute, the database server can provide context information to the event handler using the event_parameter function. The TRIGGER EVENT statement allows you to explicitly supply these parameters, to simulate a context for the event handler.

Remarks

Actions are tied to particular trigger conditions or schedules by a CREATE EVENT statement. You can use the TRIGGER EVENT statement to force the event handler to execute, even when the scheduled time or trigger condition has not occurred. TRIGGER EVENT does not execute disabled event handlers.

Each *value* is a string. The maximum length of each *value* is limited by the maximum page size specified by the -gp server option. If the length of *value* exceeds the page size, the string is truncated at the point at which the page is full.

Permissions

Must have DBA authority.

Side effects

None.

See also

- “-gp server option” [*SQL Anywhere Server - Database Administration*]
- “ALTER EVENT statement” on page 365
- “CREATE EVENT statement” on page 450
- “EVENT_PARAMETER function [System]” on page 200

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following example shows how to pass a string parameter to an event. The event displays the time it was triggered on the database server messages window.

```
CREATE EVENT ev_PassedParameter
HANDLER
BEGIN
  MESSAGE 'ev_PassedParameter - was triggered at ' ||
event_parameter( 'time' );
END;
TRIGGER EVENT ev_PassedParameter( "Time"=string( current timestamp ) );
```

TRUNCATE statement

Use this statement to delete all rows from a table without deleting the table definition.

Syntax

TRUNCATE
TABLE [*owner*.]*table-name*
| **MATERIALIZED VIEW** [*owner*.]*materialized-view-name*

Remarks

The TRUNCATE statement deletes all rows from the table or materialized view.

Note
The TRUNCATE TABLE statement should be used with great care on a database involved in synchronization or replication because the statement deletes all rows from a table, similar to a DELETE statement that doesn't have a WHERE clause. However, no triggers are fired as a result of a TRUNCATE statement. Furthermore, the row deletions are not entered into the transaction log and therefore are not synchronized or replicated. This can lead to inconsistencies that can cause synchronization or replication to fail.

After a TRUNCATE statement, the object's schema and all the indexes continue to exist until you issue a DROP statement. The schema definitions and constraints remain intact, and triggers and permissions remain in effect.

table-name can be the name of a base table or a temporary table.

With TRUNCATE TABLE, if all the following criteria are satisfied, a fast form of table truncation is executed:

- There are no foreign keys either to or from the table.
- The TRUNCATE TABLE statement is not executed within a trigger.
- The TRUNCATE TABLE statement is not executed within an atomic statement.

If a fast truncation is carried out, individual DELETEs are not recorded in the transaction log, and a COMMIT is carried out before and after the operation. Fast truncation cannot be used within snapshot transactions. See “Snapshot isolation” [*SQL Anywhere Server - SQL Usage*].

If you attempt to use TRUNCATE TABLE on a table on which an immediate text index is built, or that is referenced by an immediate view, the truncation fails. This does not occur for non-immediate text indexes or materialized views; however, it is strongly recommended that you truncate the data in dependent indexes and materialized views before executing the TRUNCATE TABLE statement on a table, and then refreshing the indexes and materialized views after. See “TRUNCATE statement” on page 768, and “TRUNCATE TEXT INDEX statement” on page 770.

Permissions

- Must be the table owner, or have DBA authority, or have ALTER permissions on the table.
- For base tables and materialized views, the TRUNCATE statement requires exclusive access to the table, as the operation is atomic (either all rows are deleted, or none are). This means that any cursors that were previously opened and that reference the table being truncated must be closed and a COMMIT or ROLLBACK must be issued to release the reference to the table.
- For temporary tables, each user has their own copy of the data, and exclusive access is not required when executing the TRUNCATE statement.

Side effects

- When you truncate a materialized view, you change the status of the view to uninitialized. See “Materialized view statuses and properties” [*SQL Anywhere Server - SQL Usage*].
- Delete triggers are not fired by the TRUNCATE statement.
- A COMMIT is performed before and after a TRUNCATE statement is executed.
- Individual deletions of rows are not entered into the transaction log, so the TRUNCATE operation is not replicated. Do not use this statement in SQL Remote replication or on a MobiLink remote database.
- If the table contains a column defined as DEFAULT AUTOINCREMENT or DEFAULT GLOBAL AUTOINCREMENT, the truncation operation resets the next available value for the column.

See also

- “DELETE statement” on page 558
- “TRUNCATE TEXT INDEX statement” on page 770

Standards and compatibility

- **SQL/2003** Transact-SQL extension.

Example

Delete all rows from the Departments table:

```
TRUNCATE TABLE Departments;
```

TRUNCATE TEXT INDEX statement

Deletes the data in a MANUAL or an AUTO REFRESH text index.

Syntax

TRUNCATE TEXT INDEX *text-index-name*
ON [*owner*.]*table-name*

Parameters

- **ON clause** The name of the table on which the text index is built.

Remarks

Use the TRUNCATE TEXT INDEX statement when you want to delete data from a manual text index without dropping the text index definition. For example, if you want to alter the text configuration object for the text index to change the stoplist, you must first truncate the text index, change the text configuration object it refers to, and then refresh the text index to populate it with new data.

You cannot perform a TRUNCATE TEXT INDEX statement on a text index defined as IMMEDIATE REFRESH (the default). For IMMEDIATE REFRESH text indexes, you must drop the index instead.

Permissions

- Must be the owner of the table the text index is built on, or have DBA authority, or have ALTER permissions on the table.
- The TRUNCATE TEXT INDEX requires exclusive access to the table. This means that any open cursors that reference the table being truncated must be closed, and a COMMIT or ROLLBACK statement must be issued to release the reference to the table.

Side effects

Automatic commit

See also

- “Full text searching” [*SQL Anywhere Server - SQL Usage*]
- “Text indexes” [*SQL Anywhere Server - SQL Usage*]
- “CREATE TEXT INDEX statement” on page 536
- “ALTER TEXT INDEX statement” on page 399
- “DROP TEXT INDEX statement” on page 590
- “REFRESH TEXT INDEX statement” on page 706

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The first statement creates the txt_index_manual text index. The second statement populates the text index with data. The third statement truncates the text index data.

```
CREATE TEXT INDEX txt_index_manual ON MarketingInformation ( Description )
   MANUAL REFRESH;
REFRESH TEXT INDEX txt_index_manual ON MarketingInformation;
TRUNCATE TEXT INDEX txt_index_manual ON MarketingInformation;
```

The truncated text index is repopulated with data the next time it is refreshed.

UNION clause

Use this statement to combine the results of two or more select statements.

Syntax

[**WITH** *temporary-views*] *query-block*
  UNION [**ALL** | **DISTINCT**] *query-block*
[**ORDER BY** [*integer* | *select-list-expression-name*] [**ASC** | **DESC**], ...]
[**FOR XML** *xml-mode*]
[**OPTION(** *query-hint*, ... **)**]

query-block : see "Common elements in SQL syntax" on page 354

query-hint :
MATERIALIZED VIEW OPTIMIZATION *option-value*
| **FORCE OPTIMIZATION**
| *option-name* = *option-value*

option-name : *identifier*

option-value : *hostvar* (indicator allowed), *string*, *identifier*, or *number*

Parameters

- **OPTION clause** Use this clause to specify hints for executing the statement. The following hints are supported:

  - MATERIALIZED VIEW OPTIMIZATION *option-value*
  - FORCE OPTIMIZATION
  - *option-name* = *option-value*

  For a description of these options, see the OPTIONS clause of the "SELECT statement" on page 728.

Remarks

The results of several query blocks can be combined into a larger result using UNION. Each *query-block* must have the same number of items in the select list.

The results of UNION ALL are the combined results of the query blocks. The results of UNION are the same as UNION ALL, except that duplicate rows are eliminated. Eliminating duplicates requires extra processing, so UNION ALL should be used instead of UNION where possible. UNION DISTINCT is identical to UNION.

If corresponding items in two select lists have different data types, SQL Anywhere chooses a data type for the corresponding column in the result and automatically convert the columns in each *query-block* appropriately.

The first query block of the UNION is used to determine the names to be matched with the ORDER BY clause.

The column names displayed are the same column names that are displayed for the first *query-block*. An alternative way of customizing result set column names is to use the WITH clause on the *query-block*.

Permissions

Must have SELECT permission for each *query-block*.

Side effects

None.

See also

- "EXCEPT clause" on page 595
- "INTERSECT clause" on page 657
- "UNION clause" on page 771
- "SELECT statement" on page 728

Standards and compatibility

- **SQL/2003** Core feature.

Example

List all distinct surnames of employees and customers.

```
SELECT Surname
FROM Employees
UNION
SELECT Surname
FROM Customers;
```

UNLOAD statement

Use this statement to unload data from a data source into a file.

Syntax

UNLOAD *data-source*
{ **TO** *filename*
  | **INTO FILE** *filename*
  | **INTO CLIENT FILE** *client-filename*
  | **INTO VARIABLE** *variable-name* }
[*unload-option* ...]

data-source
[**FROM**] [**TABLE**] [*owner.*]*table-name*
| [**FROM**] [**MATERIALIZED VIEW**] [*owner.*]*materialized-view-name*
| *select-statement*

filename : *string* | *variable*

client-filename : *string* | *variable*

Syntax

unload-option :
APPEND { **ON** | **OFF** }
| **BYTE ORDER MARK** { **ON** | **OFF** }
| { **COMPRESSED** | **NOT COMPRESSED** }
| { **COMPRESSED** | **NOT COMPRESSED** }
| **COLUMN DELIMITED BY** *string*
| **DELIMITED BY** *string*
| **ENCODING** *encoding*
| { **ENCRYPTED KEY '***key***'** [**ALGORITHM '***algorithm***'**] | **NOT ENCRYPTED** }
| **ESCAPE CHARACTER** *character*
| **ESCAPES** { **ON** | **OFF** }
| **FORMAT** { **TEXT** | **BCP** }
| **HEXADECIMAL** { **ON** | **OFF** }
| **ORDER** { **ON** | **OFF** }
| **QUOTE** *string*
| **QUOTES** { **ON** | **OFF** }
| **ROW DELIMITED BY** *string*

encoding : *string*

Parameters

- **TO clause** The name of the file to unload data into. The *filename* path is relative to the database server's starting directory. If the file does not exist, it is created. If it already exists, it is overwritten unless APPEND ON is also specified.

- **INTO FILE clause** Semantically equivalent to TO *filename*.

- **INTO CLIENT FILE clause** The file on the client computer into which the data is unloaded. If the file doesn't exist, it is created. If it already exists, it is overwritten unless APPEND ON is also specified. The path is resolved on the client computer relative to the current working directory of the client application.

  To unload data onto a client computer using SQL Remote, see "PASSTHROUGH statement [SQL Remote]" on page 693.

- **INTO VARIABLE clause** The variable to unload the data into. The variable must already exist and be of CHAR, NCHAR or BINARY type. The APPEND option causes the unloaded data to be concatenated to the current contents of the variable.

- **APPEND clause** When APPEND is ON, unloaded data is appended to the end of the file specified. When APPEND is OFF, unloaded data replaces the contents of the file specified. APPEND is OFF by default. This clause cannot be specified when specifying the COMPRESSED or ENCRYPTED clauses, and cannot be used if the file being appended to is compressed or encrypted.

- **BYTE ORDER MARK clause** Use this clause to specify whether a byte order mark (BOM) is present in the encoding. By default, this option is ON, provided the destination for the unload is a local or client file. When the BYTE ORDER MARK option is ON, UTF-8 and UTF-16 data contains a BOM. If BYTE ORDER MARK is OFF, a BOM is not unloaded.

- **COMPRESSED clause** Specifies whether to compress the data. The default is NOT COMPRESSED. You cannot compress the data if you want the data to be appended (APPEND ON).

If the file you are appending to is compressed, you must specify the COMPRESSED clause.

- **DELIMITED BY clause** The string used between columns. The default column delimiter is a comma. You can specify an alternative column delimiter by providing a string. Only the first byte (character) of the string is used as the delimiter.
- **ENCODING clause** All database data is translated from the database character encoding to the specified character encoding. When ENCODING is not specified, the database's character encoding is used, and translation is not performed.

  For information about how to obtain the list of SQL Anywhere supported encodings, see "Supported character sets" [*SQL Anywhere Server - Database Administration*].

  If a translation error occurs during the unload operation, it is reported based on the setting of the on_charset_conversion_failure option. See "on_charset_conversion_failure option [database]" [*SQL Anywhere Server - Database Administration*].

  The following example unloads the data using the UTF-8 character encoding:

  ```
  UNLOAD TABLE mytable TO 'mytable_data_in_utf8.dat' ENCODING 'UTF-8';
  ```

  Specify the BYTE ORDER clause to include a byte order mark in the data.
- **ENCRYPTED clause** Specifies whether to encrypt the data. If you specify NOT ENCRYPTED (the default), the data is not encrypted. If you specify ENCRYPTED KEY with a key and no algorithm, the data is encrypted using AES128 and the specified key. If you specify ENCRYPTED KEY with a key and algorithm, the data is encrypted using the specified key and algorithm. The algorithm can be any of the algorithms accepted by the CREATE DATABASE statement. You cannot specify simple encryption. See "CREATE DATABASE statement" on page 432.

  You cannot encrypt the data if you want the data to be appended (APPEND ON).

  If the file you are appending to is encrypted, you must specify the ENCRYPTED clause.
- **ESCAPES clause** With ESCAPES turned ON (the default), characters following the backslash character are recognized and interpreted as special characters by the database server. Newline characters can be included as the combination \n, other characters can be included in data as hexadecimal ASCII codes, such as \x09 for the tab character. A sequence of two backslash characters (\\ ) is interpreted as a single backslash. A backslash followed by any character other than n, x, X, or \ is interpreted as two separate characters. For example, \q inserts a backslash and the letter q.
- **FORMAT clause** Outputs data in either TEXT format or in BCP out format. If you choose TEXT, input lines are assumed to be text characters, one row per line, with values separated by the column delimiter string. Choosing BCP allows the import of Adaptive Server Enterprise-generated BCP out files containing BLOBs.
- **HEXADECIMAL clause** By default, HEXADECIMAL is ON. Binary column values are written as 0x*nnnnnn*..., where 0x is a zero followed by an x, and each *n* is a hexadecimal digit. It is important to use HEXADECIMAL ON when dealing with multibyte character sets.

  The HEXADECIMAL clause can be used only with the FORMAT TEXT clause.
- **ORDER clause** With ORDER ON (the default), the exported data is ordered by clustered index if one exists. If a clustered index does not exist, the exported data is ordered by primary key values. With ORDER OFF, the data is exported in the same order you see when selecting from the table without an

ORDER BY clause. Exporting is slower with ORDER ON. However, reloading using the LOAD TABLE statement is quicker because of the simplicity of the indexing step.

For UNLOAD *select-statement*, the ORDER clause is ignored. However, you can still order the data by specifying an ORDER BY clause in the SELECT statement.

- **QUOTE clause** The QUOTE clause is for TEXT data only; the *string* is placed around string values. The default is a single quote (apostrophe).
- **QUOTES clause** With QUOTES turned on (the default), single quotes are placed around all exported strings.
- **ROW DELIMITED BY clause** Use this clause to specify the string that indicates the end of a record. The default delimiter string is a newline (\n). However, it can be any string up to up to 255 bytes in length; for example, `... ROW DELIMITED BY '###' ...`. The same formatting requirements apply to other SQL strings. If you want to specify tab-delimited values, you could specify the hexadecimal escape sequence for the tab character (9), `... ROW DELIMITED BY '\x09' ...`. If your delimiter string contains a \n, it will match either \r\n or \n.

Remarks

The UNLOAD statement allows data from a SELECT statement to be exported to a comma-delimited file. The result set is not ordered unless the SELECT statement contains an ORDER BY clause.

The UNLOAD TABLE statement allows efficient mass exporting from a database table or materialized view into a file. The UNLOAD TABLE statement is more efficient than the Interactive SQL statement OUTPUT, and can be called from any client application.

The database server, or the client application, depending upon whether TO FILE or INTO CLIENT FILE was specified, respectively, must have operating system permissions to write to the specified file.

For UNLOAD TABLE, when unloading table columns with binary data types, UNLOAD TABLE writes hexadecimal strings, of the form \x*nnnn*, where *n* is a hexadecimal digit. For UNLOAD *select-statement*, when unloading result set columns with binary data types, UNLOAD writes hexadecimal strings of the form *\0xnnnn*, where *n* is a hexadecimal digit.

When unloading and reloading a database that has proxy tables, you must create an external login to map the local user to the remote user, even if the user has the same password on both the local and remote databases. If you do not have an external login, the reload may fail because you cannot connect to the remote server. See "Working with external logins" [*SQL Anywhere Server - SQL Usage*].

When unloading into a variable (INTO VARIABLE), the output is converted to a character set as follows:

1. Use the character set specified in the ENCODING clause.
2. If no ENCODING clause is specified, then the database NCHAR character set is used if the variable is of type NCHAR; otherwise, the database CHAR character set is used.

Also, the chosen encoding must match the database CHAR encoding if the variable is of CHAR type The chosen encoding must match the database NCHAR encoding if the variable if of NCHAR type Any encoding can be used for BINARY variables

If you choose to compress and encrypt the unloaded data, it is compressed first.

UNLOAD TABLE places an exclusive lock on the whole table or materialized view.

Permissions

When unloading into a variable, no permissions are required (other than the normal permissions required to access the data source).

The permissions required to execute an UNLOAD statement depend on the database server -gl option, as follows:

- If the -gl option is set to ALL, you must have SELECT permissions on the table or tables referenced in the UNLOAD statement.
- If the -gl option is set to DBA, you must have DBA authority.
- If the -gl option is set to NONE, UNLOAD is not permitted.

See “-gl server option” [*SQL Anywhere Server - Database Administration*].

When writing to a file on a client computer:

- WRITECLIENTFILE authority is required. See “WRITECLIENTFILE authority” [*SQL Anywhere Server - Database Administration*].
- Write permissions are required for the directory being written to.
- The allow_write_client_file database option must be enabled. See “allow_write_client_file option [database]” [*SQL Anywhere Server - Database Administration*].
- The write_client_file secured feature must be enabled. See “-sf server option” [*SQL Anywhere Server - Database Administration*].

Side effects

None. The query is executed at the current isolation level.

See also

- “CREATE DATABASE statement” on page 432
- “LOAD TABLE statement” on page 660
- “Using clustered indexes” [*SQL Anywhere Server - SQL Usage*]
- “OUTPUT statement [Interactive SQL]” on page 686
- “Export data with the UNLOAD statement” [*SQL Anywhere Server - SQL Usage*]
- “Export data with the UNLOAD statement” [*SQL Anywhere Server - SQL Usage*]
- “Accessing data on client computers” [*SQL Anywhere Server - SQL Usage*]
- “Importing and exporting data” [*SQL Anywhere Server - SQL Usage*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following example unloads the contents of the Products table to a UTF-8-encoded file, *productsT.dat*:

```
UNLOAD TABLE Products TO 'productsT.dat' ENCODING 'UTF-8';
```

The following example creates a variable called @myProducts and then unloads the Products.Name column into the variable:

```
CREATE VARIABLE @myProducts LONG VARCHAR;
UNLOAD SELECT NAME FROM Products INTO VARIABLE @myProducts;
```

UPDATE statement

Use this statement to modify existing rows in database tables.

Syntax 1

UPDATE [*row-limitation*] *table-list*]
SET *set-item*, ...
[**FROM** *table-expression* [,...]]
[**WHERE** *search-condition*]
[**ORDER BY** *expression* [**ASC** | **DESC**] , ...]
[**OPTION(** *query-hint*, ... **)**]

table-list :
table-name [,...]

table-name :
[*owner*.]*table-name* [[**AS**] *correlation-name*]
| [*owner*.]*view-name* [[**AS**] *correlation-name*]
| *derived-table*

derived-table :
(*select-statement* **)**
[**AS**] *correlation-name* [**(** *column-name* [,...] **)**]

table-expression :
A full table expression that can include joins. See "FROM clause" on page 613.

Syntax 2

UPDATE *table-name*
SET *set-item*, ...
VERIFY (*column-name*, ... **) VALUES (** *expression*, ... **)**
[**WHERE** *search-condition*]
[**ORDER BY** *expression* [**ASC** | **DESC**], ...]
[**OPTION(** *query-hint*, ... **)**]

Syntax 3

UPDATE [*owner*.]*table-name*
PUBLICATION *publication*
{ **SUBSCRIBE BY** *expression*
| **OLD SUBSCRIBE BY** *expression* **NEW SUBSCRIBE BY** *expression*
}
WHERE *search-condition*

row-limitation :
FIRST
| **TOP** *n* [**START AT** *m*]

set-item :
[*correlation-name.*]*column-name* **=** { *expression* | **DEFAULT** }
| [*owner-name.*]*table-name***.***column-name* **=** { *expression* | **DEFAULT** }
| *@variable-name* **=** *expression*

query-hint :
MATERIALIZED VIEW OPTIMIZATION *option-value*
| **FORCE OPTIMIZATION**
| **FORCE NO OPTIMIZATION**
| *option-name* = *option-value*

table-name :
[*owner*.]*base-table-name*
| *temporary-table-name*
| *derived-table-name*
| [*owner*.]*view-name*

option-name : *identifier*

option-value : *hostvar* (indicator allowed), *string*, *identifier*, or *number*

Parameters

- **UPDATE clause** For Syntax 1, *table-list* can include temporary tables, derived tables, or views. Views and derived tables can be updated unless they are non-updatable. For Syntax 2 and 3, *table-name* must be a base table.

  UPDATES can be performed on views only if the query specification defining the view is updatable. For more information about identifying views that are inherently non-updatable, see "Working with views" [*SQL Anywhere Server - SQL Usage*].

- **row-limitation clause** The row limiting clause allows you to return only a subset of the rows that satisfy the WHERE clause. The TOP and START AT values can be a host variable, integer constant, or integer variable. The TOP value must be greater than or equal to 0. The START AT value must be greater than 0. Normally, when specifying these clauses, an ORDER BY clause is specified as well to order the rows in a meaningful manner. See "Explicitly limiting the number of rows returned by a query" [*SQL Anywhere Server - SQL Usage*].

- **SET clause** The set clause specifies the columns and how the values are changed.

  You can use the SET clause to set the column to a computed column value using this format:

  ```
  SET column-name = expression, ...
  ```

  Each named column is set to the value of the expression on the right-hand side of the equal sign. There are no restrictions on the *expression*. If the expression is a *column-name*, the old value is used.

  If a column has a default defined, you can use the SET clause to set a column to its default value. See the Examples section for an example of this.

  You can also use the SET clause to assign a variable using this format:

  ```
  SET @variable-name = expression, ...
  ```

  When assigning a variable, the variable must already be declared, and its name must begin with the "at" sign (@). Variable and column assignments can be mixed together, and any number can be used. If a name on the left side of an assignment in the SET list matches a column in the updated table and the variable name, the statement updates the column.

Following is an example of part of an UPDATE statement. It assigns a variable in addition to updating the table:

```
UPDATE T SET @var = expression1, col1 = expression2
WHERE...
```

This is equivalent to:

```
SELECT @var = expression1
FROM T
WHERE... ;
UPDATE T SET col1 = expression2
WHERE...
```

- **FROM clause** If the FROM clause is present, the WHERE clause qualifies the rows of the FROM clause.

  The FROM *table-expression* clause allows tables to be updated based on joins. *table-expression* can contain arbitrary complex table expressions, such as KEY and NATURAL joins. For a full description of the FROM clause and joins, see "FROM clause" on page 613.

  If a FROM clause is used, it is important to qualify the table name the same way in both parts of the statement. If a correlation name is used in one place, the same correlation name must be used elsewhere. Otherwise, an error is generated.

  The following statement illustrates a potential ambiguity in table names in UPDATE statements with two FROM clauses that use correlation names:

```
UPDATE
FROM table_1
FROM table_1 AS alias_1, table_2 AS alias_2
WHERE ...
```

  table table_1 doesn't have a correlation name in the first FROM clause but does in the second FROM clause. In this case, table_1 in the first clause is identified with alias_1 in the second clause—there is only one instance of table_1 in this statement. This is allowed as an exception to the general rule that where a table is identified with a correlation name and without a correlation name in the same statement, two instances of the table are considered.

  However, in the following example, there are two instances of table_1 in the second FROM clause. The statement fails with a syntax error because it is not clear which instance of the table_1 from the second FROM clause matches the first instance of table_1 in the first FROM clause.

```
UPDATE
FROM table_1
FROM table_1 AS alias_1, table_1 AS alias_2
WHERE ...
```

  This clause is allowed only if ansi_update_constraints is set to Off. See "ansi_update_constraints option [compatibility]" [*SQL Anywhere Server - Database Administration*].

  For a full description of joins, see "Joins: Retrieving data from several tables" [*SQL Anywhere Server - SQL Usage*].

For more information, see "FROM clause" on page 613.

- **WHERE clause** If a WHERE clause is specified, only rows satisfying the search condition are updated. If no WHERE clause is specified, every row is updated.
- **ORDER BY clause** Normally, the order in which rows are updated does not matter. However, in conjunction with the FIRST or TOP clause the order can be significant.

  You cannot use ordinal column numbers in the ORDER BY clause.

  You must not update columns that appear in the ORDER BY clause unless you set the ansi_update_constraints option to Off. See "ansi_update_constraints option [compatibility]" [*SQL Anywhere Server - Database Administration*].
- **OPTION clause** Use this clause to specify hints for executing the statement. The following hints are supported:

  - MATERIALIZED VIEW OPTIMIZATION *option-value*
  - FORCE OPTIMIZATION
  - FORCE NO OPTIMIZATION
  - *option-name* = *option-value*

Remarks

Character strings inserted into tables are always stored in the same case as they are entered, regardless of whether the database is case sensitive or not. A CHAR data type column updated with the string Street is always held in the database with an uppercase S and the remainder of the letters lowercase. SELECT statements return the string as Street. If the database is not case sensitive, however, all comparisons make Street the same as street, STREET, and so on. Further, if a single-column primary key already contains an entry Street, an INSERT of street is rejected, as it would make the primary key not unique.

If the new value does not differ from the old value, no change is made to the data. However, BEFORE UPDATE triggers fire any time an UPDATE occurs on a row, whether the new value differs from the old value. AFTER UPDATE triggers fire only if the new value is different from the old value.

Syntax 1 of the UPDATE statement modifies values in rows of one or more tables. Syntax 2 and 3 are applicable only to SQL Remote.

Syntax 2 is intended for use with SQL Remote only, in single-row updates of a single table executed by the Message Agent. The VERIFY clause contains a set of values that are expected to be present in the row being updated. If the values do not match, any RESOLVE UPDATE triggers are fired before the UPDATE proceeds. The UPDATE does not fail simply because the VERIFY clause fails to match.

Syntax 3 of the UPDATE statement is used to implement a specific SQL Remote feature, and is to be used inside a BEFORE trigger. It provides a full list of SUBSCRIBE BY values any time the list changes. It is placed in SQL Remote triggers so that the database server can compute the current list of SUBSCRIBE BY values. Both lists are placed in the transaction log.

The Message Agent uses the two lists to make sure that the row moves to any remote database that did not have the row and now needs it. The Message Agent also removes the row from any remote database that has the row and no longer needs it. A remote database that has the row and still needs it is not affected by the UPDATE statement.

For publications created using a subquery in a SUBSCRIBE BY clause, you must write a trigger containing syntax 3 of the UPDATE statement to ensure that the rows are kept in their proper subscriptions.

Syntax 3 of the UPDATE statement allows the old SUBSCRIBE BY list and the new SUBSCRIBE BY list to be explicitly specified, which can make SQL Remote triggers more efficient. In the absence of these lists, the database server computes the old SUBSCRIBE BY list from the publication definition. Since the new SUBSCRIBE BY list is commonly only slightly different from the old SUBSCRIBE BY list, the work to compute the old list may be done twice. By specifying both the old and new lists, you can avoid this extra work.

The SUBSCRIBE BY expression is either a value or a subquery.

Syntax 3 of the UPDATE statement makes an entry in the transaction log, but does not change the database table.

Updating a significant amount of data using the UPDATE statement also updates column statistics.

Permissions

Must have UPDATE permission for the columns being modified.

Side effects

Column statistics are updated.

See also

- "DELETE statement" on page 558
- "INSERT statement" on page 649
- "FROM clause" on page 613
- "Joins: Retrieving data from several tables" [*SQL Anywhere Server - SQL Usage*]
- "UPDATE (positioned) statement [ESQL] [SP]" on page 782
- "Locking during updates" [*SQL Anywhere Server - SQL Usage*]

Standards and compatibility

- **SQL/2003** Syntax 1 is a core feature, except for the FROM and ORDER BY clauses, which are vendor extensions. Syntax 2 and 3 are vendor extensions for use only with SQL Remote.

  To enforce SQL/2003 compatibility, ensure that the ansi_update_constraints option is set to Strict. See "ansi_update_constraints option [compatibility]" [*SQL Anywhere Server - Database Administration*].

Examples

Using the sample database, this example transfers employee Philip Chin (employee 129) from the sales department to the marketing department.

```
UPDATE Employees
SET DepartmentID = 400
WHERE EmployeeID = 129;
```

Using the sample database, this example renumbers all existing sales orders by subtracting 2000 from the ID.

```
UPDATE SalesOrders AS orders
SET orders.ID = orders.ID - 2000
ORDER BY orders.ID ASC;
```

This update is possible only if the foreign key of the SalesOrderItems table (referencing the primary key SalesOrders.ID) is defined with the action ON UPDATE CASCADE. The SalesOrderItems table is then updated as well.

For more information about foreign key properties, see "ALTER TABLE statement" on page 389 and "CREATE TABLE statement" on page 523.

Using the sample database, this example changes the price of a product at isolation level 2, rather than using the current isolation level setting of the database.

```
UPDATE Products
SET UnitPrice = 7.00
WHERE ID = 501
OPTION( isolation_level = 2 );
```

The following example shows how to update a table to set a column to its default value. In this example, you create a table, MyTable, populate it with data, and then execute an UPDATE statement specifying the SET clause to change the column values to their defaults.

```
CREATE TABLE MyTable(
   PK INT PRIMARY KEY  DEFAULT AUTOINCREMENT,
   TableName CHAR(128) NOT NULL,
   TableNameLen INT DEFAULT 20,
   LastUser CHAR(10) DEFAULT last user,
   LastTime TIMESTAMP DEFAULT TIMESTAMP,
   LastTimestamp TIMESTAMP DEFAULT @@dbts );

INSERT INTO MyTable WITH AUTO NAME
   SELECT
      LENGTH(t.table_name) AS TableNameLen,
      t.table_name AS TableName
   FROM SYS.SYSTAB t
   WHERE table_id<=10;

UPDATE MyTable SET  LastTime = DEFAULT, LastTimestamp = DEFAULT
   WHERE TableName LIKE '%sys%';
```

UPDATE (positioned) statement [ESQL] [SP]

Use this statement to modify the data at the current location of a cursor.

Syntax 1

UPDATE WHERE CURRENT OF *cursor-name*
{ **USING DESCRIPTOR** *sqlda-name* | **FROM** *hostvar-list* }

Syntax 2

UPDATE *update-table*, ...
SET *set-item*, ...
WHERE CURRENT OF *cursor-name*

hostvar-list : *indicator variables allowed*

update-table :
[*owner-name.*]*object-name* [[**AS**] *correlation-name*]

set-item :
[*correlation-name.*]*column-name* **=** { *expression* | **DEFAULT** }
| [*owner-name.*]*object-name***.***column-name* **=** { *expression* | **DEFAULT** }

object-name : *identifier* (a table or view name)

sqlda-name : *identifier*

Parameters

- **USING DESCRIPTOR clause** When assigning a variable, the variable must already be declared, and its name must begin with the "at" sign (@). Variable and column assignments can be mixed together, and any number can be used. If a name on the left side of an assignment in the SET list matches a column in the updated table and the variable name, the statement updates the column.
- **SET clause** The columns that are referenced in *set-item* must be in the table or view that is updated. They cannot refer to aliases, nor to columns from other tables or views. If the table or view you are updating is given a correlation name in the cursor specification, you must use the correlation name in the SET clause.

  Each *set-item* is associated with a single *update-table*, and the corresponding column of the matching table in the cursor's query is modified. The *expression* references columns of the tables identified in the UPDATE list and may use constants, variables, expressions from the select list of the query, or combinations of the above using operators such as +, -, ..., COALESCE, IF, and so on. The *expression* can not reference aliases of expressions from the cursor's query, nor can they reference columns of other tables of the cursor's query which do not appear in the UPDATE list. Subselects, subquery predicates, and aggregate functions can not be used in the *set-items*.

  Each *update-table* is matched to a table in the query for the cursor as follows:

  - If a correlation name is specified, it is matched to a table in the cursor's query that has the same *table-or-view-name* and the same *correlation-name*.
  - Otherwise, if there is a table in the cursor's query that has the same *table-or-view-name* that does not have a correlation name specified, or has a correlation name that is the same as the *table-or-view-name*, then the update table is matched with this table in the cursor's query.
  - Otherwise, if there is a single table in the cursor's query that has the same *table-or-view-name* as the update table, then the update table is matched with this table in the cursor's query.

  If a column has a default defined, you can use the SET clause to set a column to its default value. For an example of this, see the Examples section of "UPDATE statement" on page 777.

Remarks

This form of the UPDATE statement updates the current row of the specified cursor. The current row is defined to be the last row successfully fetched from the cursor, and the last operation on the cursor must not have been a positioned DELETE statement.

For syntax 1, columns from the SQLDA or values from the host variable list correspond one-to-one with the columns returned from the specified cursor. If the sqldata pointer in the SQLDA is the null pointer, the corresponding select list item is not updated.

In syntax 2, the requested columns are set to the specified values for the row at the current row of the specified query. The columns do not need to be in the select list of the specified open cursor. This format can be prepared.

Also, when assigning a variable, the variable must already be declared, and its name must begin with the "at" sign (@). Variable and column assignments can be mixed together, and any number can be used. If a name on the left side of an assignment in the SET list matches a column in the updated table and the variable name, the statement updates the column.

The USING DESCRIPTOR, FROM *hostvar-list*, and *hostvar* formats are for embedded SQL only.

Permissions

Must have UPDATE permission on the columns being modified.

Side effects

None.

See also

- "INSERT statement" on page 649
- "LOAD TABLE statement" on page 660
- "MERGE statement" on page 673
- "DELETE statement" on page 558
- "DELETE (positioned) statement [ESQL] [SP]" on page 561
- "UPDATE statement" on page 777

Standards and compatibility

- **SQL/2003** Core feature. The range of cursors that can be updated may contain vendor extensions if the ansi_update_constraints option is set to Off.
- **Sybase** Embedded SQL use is supported by Open Client/Open Server, and procedure and trigger use is supported in SQL Anywhere.

Example

The following is an example of an UPDATE statement WHERE CURRENT OF cursor:

```
UPDATE Employees
SET Surname = 'Jones'
WHERE CURRENT OF emp_cursor;
```

UPDATE statement [SQL Remote]

Use this statement to modify data in the database.

Syntax 1

UPDATE *table-list*
SET *column-name* = *expression*, ...
[**VERIFY (** *column-name*, ... **) VALUES (** *expression*, ... **)**]
[**WHERE** *search-condition*]
[**ORDER BY** *expression* [**ASC** | **DESC**], ...]

Syntax 2

UPDATE *table-name*
PUBLICATION *publication-name*
{ **SUBSCRIBE BY** *subscription-expression* |
  OLD SUBSCRIBE BY *old-subscription-expression*
  NEW SUBSCRIBE BY *new-subscription-expression* }
WHERE *search-condition*

expression: *value* | *subquery*

Parameters

- **table-name** The *table-name* indicates the table that must be modified on the remote databases.
- **publication-name** The *publication-name* indicates the publication for which subscriptions must be changed.
- **subscription-expression** The value of *subscription-expression* is used by the SQL Remote to determine both new and existing recipients of the rows. The *subscription-expression* is either a value or a subquery.Alternatively, you can provide both OLD and NEW subscription expressions.
- **WHERE** The WHERE clause specifies which rows are to be transferred between subscribed databases.

Remarks

The UPDATE statement is used to modify rows of one or more tables. Each named column is set to the value of the expression on the right-hand side of the equal sign. There are no restrictions on the *expression*. Even *column-name* can be used in the expression—the old value is used.

If no WHERE clause is specified, every row is updated. If a WHERE clause is specified, then only those rows which satisfy the search condition are updated.

Normally, the order that rows are updated does not matter. However, in conjunction with the NUMBER(*) function, an ordering can be useful to get increasing numbers added to the rows in some specified order. Also, if you want to do something like add 1 to the primary key values of a table, it is necessary to do this in descending order by primary key, so that you do not get duplicate primary keys during the operation.

Views can be updated provided the SELECT statement defining the view does not contain a GROUP BY clause, an aggregate function, or involve a UNION clause.

Character strings inserted into tables are always stored in the case they are entered, regardless of whether the database is case sensitive or not. So, a character data type column updated with a string Value is always held in the database with an uppercase V and the remainder of the letters lowercase. SELECT statements return the string as Value. If the database is not case sensitive, however, all comparisons make Value the same as value, VALUE, and so on. Further, if a single-column primary key already contains an entry Value, an INSERT of value is rejected, as it would make the primary key not unique.

The optional FROM clause allows tables to be updated based on joins. If the FROM clause is present, the WHERE clause qualifies the rows of the FROM clause. Data is updated only in the table list immediately following the UPDATE keyword.

If a FROM clause is used, it is important to qualify the table name that is being updated the same way in both parts of the statement. If a correlation name is used in one place, the same correlation name must be used in the other. Otherwise, an error is generated.

Syntax 1 and Syntax 2 are applicable only to SQL Remote.

Syntax 2 with no OLD and NEW SUBSCRIBE BY expressions must be used in a BEFORE trigger.

Syntax 2 with OLD and NEW SUBSCRIBE BY expressions can be used anywhere.

Syntax 1 is intended for use with SQL Remote only, in single-row updates executed by the Message Agent. The VERIFY clause contains a set of values that are expected to be present in the row being updated. If the values do not match, any RESOLVE UPDATE triggers are fired before the UPDATE proceeds. The UPDATE does not fail if the VERIFY clause fails to match. When the VERIFY clause is specified, only one table can be updated at a time.

Syntax 2 is intended for use with SQL Remote only. If no OLD and NEW expressions are used, it must be used inside a BEFORE trigger so that it has access to the relevant values. The purpose is to provide a full list of subscribe by values any time the list changes. It is placed in SQL Remote triggers so that the database server can compute the current list of SUBSCRIBE BY values. Both lists are placed in the transaction log.

The Message Agent uses the two lists to make sure that the row moves to any remote database that did not have the row and now needs it. The Message Agent also removes the row from any remote database that has the row and no longer needs it. A remote database that has the row and still needs it is not affected by the UPDATE statement.

Syntax 2 of the UPDATE statement allows the old SUBSCRIBE BY list and the new SUBSCRIBE BY list to be explicitly specified, which can make SQL Remote triggers more efficient. In the absence of these lists, the database server computes the old SUBSCRIBE BY list from the publication definition. Since the new SUBSCRIBE BY list is commonly only slightly different from the old SUBSCRIBE BY list, the work to compute the old list may be done twice. By specifying both the old and new lists, this extra work can be avoided.

The OLD and NEW SUBSCRIBE BY syntax is especially useful when many tables are being updated in the same trigger with the same subscribe by expressions. This can dramatically increase performance.

The SUBSCRIBE BY expression is either a value or a subquery.

Syntax 2 of the UPDATE statement is used to implement a specific SQL Remote feature, and is to be used inside a BEFORE trigger.

For publications created using a subquery in a subscription expression, you must write a trigger containing syntax 2 of the UPDATE statement to ensure that the rows are kept in their proper subscriptions.

For a full description of this feature, see “Using BEFORE UDPATE triggers” [*SQL Remote*].

Syntax 2 of the UPDATE statement makes an entry in the transaction log, but does not change the database table.

Permissions

Must have UPDATE permission for the columns being modified.

Side effects

None.

See also

- "Using BEFORE UDPATE triggers" [*SQL Remote*]

Standards and compatibility

- **SQL/2003** Vendor extension.

Example

The following example transfers employee Philip Chin (employee 129) from the sales department to the marketing department.

```
UPDATE Employees
VERIFY( DepartmentID ) VALUES( 300 )
SET DepartmentID = 400
WHERE EmployeeID = 129;
```

VALIDATE statement

Use this statement to validate the current database, or a single table or materialized view in the current database.

Syntax 1 - Validating tables and materialized views

VALIDATE {
TABLE [*owner*.]*table-name*
  | **MATERIALIZED VIEW** [*owner*.]*materialized-view-name* }
[**WITH EXPRESS CHECK**]

Syntax 2 - Validating a database

VALIDATE { **CHECKSUM** | **DATABASE** }

Syntax 3 - Validating indexes

VALIDATE {
INDEX *index-name*
| [**INDEX**] **FOREIGN KEY** *role-name*
| [**INDEX**] **PRIMARY KEY** }
ON [*owner*.]*object-name*
}

object-name :
table-name
| *materialized-view-name*

Parameters

- **WITH EXPRESS CHECK** In addition to the default checks, check that the number of rows in the table or materialized view matches the number of entries in the index. This option does not perform individual index lookups for each row, nor does it perform checksum validation. This option can significantly improve performance when validating large databases with a small cache.

Remarks

Validation of tables includes a checksum validation, and validation that the number of rows in a table matches the number of rows in each index associated with the table. If you specify WITH EXPRESS CHECK, a checksum validation is not performed.

The VALIDATE DATABASE statement validates that all table pages in the database belong to the correct object. VALIDATE DATABASE also performs a checksum validation, but does not validate the indexes, or check data correctness.

Use the VALIDATE CHECKSUM statement to perform a checksum validation on the database. The VALIDATE CHECKSUM statement ensures that database pages have not been modified on disk. When a database is created with checksums enabled, a checksum is calculated for each database page before it is written to disk. VALIDATE CHECKSUM reads each database page from disk and calculates the checksum for each page. If the calculated checksum for a page does not match the stored checksum for that page, an error occurs and information about the invalid page appears in the database server messages window. The VALIDATE CHECKSUM statement can also be useful on databases with checksums disabled, since critical database pages still include checksums.

Use the VALIDATE INDEX statement to validate an index, including index statistics, on a table or a materialized view. The VALIDATE INDEX statement ensures that every row referenced in the index actually exists. For foreign key indexes, it also ensures that the corresponding row exists in the primary table. This check complements the validity checking carried out by the VALIDATE TABLE statement. The VALIDATE INDEX statement also verifies that the statistics reported on the specified indexes are accurate. If they are not accurate, an error is generated.

Caution
Validating a table or an entire database should be performed while no connections are making changes to the database; otherwise, errors may be reported indicating some form of database corruption even though no corruption actually exists.

Permissions

Must have DBA or VALIDATE authority.

Side effects

None.

See also

- "Validation utility (dbvalid)" [*SQL Anywhere Server - Database Administration*]
- "sa_validate system procedure" on page 963
- "Validating databases" [*SQL Anywhere Server - Database Administration*]
- "CREATE DATABASE statement" on page 432
- "CREATE INDEX statement" on page 470

Standards and compatibility

- **SQL/2003** Vendor extension.

WAITFOR statement

Use this statement to delay processing for the current connection for a specified amount of time or until a given time.

Syntax

WAITFOR {
DELAY *time*
| **TIME** *time* }
[**CHECK EVERY** *integer*]
[**AFTER MESSAGE BREAK**]

time : *string*

Parameters

- **DELAY clause** If DELAY is used, processing is suspended for the given interval.

- **TIME clause** If TIME is specified, processing is suspended until the database server time reaches the time specified. If the current server time is greater than the time specified, processing is suspended until that time on the following day.

- **CHECK EVERY clause** This optional clause controls how often the WAITFOR statement wakes up. By default, it wakes up every 5 seconds. The value is in milliseconds, and the minimum value is 250 milliseconds.

- **AFTER MESSAGE BREAK clause** The WAITFOR statement can be used to wait for a message from another connection. In most cases, when a message is received it is forwarded to the application that executed the WAITFOR statement and the WAITFOR statement continues to wait. If the AFTER MESSAGE BREAK clause is specified, when a message is received from another connection, the WAITFOR statement completes. The message text is not forwarded to the application, but it can be accessed by obtaining the value of the MessageReceived connection property.

  For more information about the MessageReceived property, see "Connection properties" [*SQL Anywhere Server - Database Administration*].

Remarks

The WAITFOR statement wakes up periodically (every 5 seconds by default) to check if it has been canceled or if messages have been received. If neither of these has happened, the statement continues to wait.

WAITFOR provides an alternative to the following statement:

```
CALL java.lang.Thread.sleep( <time_to_wait_in_millisecs> );
```

Because scheduled events execute on their own connection, scheduled events are often a better choice than using WAITFOR TIME.

Permissions

None

Side effects

The implementation of this statement uses a worker thread while it is waiting. This uses up one of the threads specified by the -gn database option (the default is 20 threads).

See also

- “CREATE EVENT statement” on page 450

Standards and compatibility

- **SQL/2003** Vendor extension.

Examples

The following example waits for three seconds:

```
WAITFOR DELAY '00:00:03';
```

The following example waits for 0.5 seconds (500 milliseconds):

```
WAITFOR DELAY '00:00:00:500';
```

The following example waits until 8 PM:

```
WAITFOR TIME '20:00';
```

In the following example, connection 1's WAITFOR statement completes when it receives the message from connection 2:

```
// connection 1:
BEGIN
  DECLARE msg LONG VARCHAR;
  LOOP  // forever
    WAITFOR DELAY '00:05:00' AFTER MESSAGE BREAK;
    SET msg = CONNECTION_PROPERTY('MessageReceived');
    IF msg != '' THEN
      MESSAGE 'Msg: ' || msg TO CONSOLE;
    END IF;
  END LOOP
END
// connection 2:
MESSAGE 'here it is' FOR connection 1
```

WHENEVER statement [ESQL]

Use this statement to specify error handling in embedded SQL programs.

Syntax

WHENEVER {
SQLERROR
| **SQLWARNING**
| **NOTFOUND** }
GOTO
  label
  | **STOP**
  | **CONTINUE**
  | { *C-code*; }

label : *identifier*

Remarks

The WHENEVER statement is used to trap errors, warnings and exceptional conditions encountered by the database when processing SQL statements. The statement can be put anywhere in an embedded SQL program and does not generate any code. The preprocessor will generate code following each successive SQL statement. The error action remains in effect for all embedded SQL statements from the source line of the WHENEVER statement until the next WHENEVER statement with the same error condition, or the end of the source file.

Errors based on source position
The error conditions are in effect based on positioning in the C language source file, not based on when the statements are executed.

The default action is CONTINUE.

Note that this statement is provided for convenience in simple programs. Most of the time, checking the sqlcode field of the SQLCA (SQLCODE) directly is the easiest way to check error conditions. In this case, the WHENEVER statement would not be used. If fact, all the WHENEVER statement does is cause the preprocessor to generate an *if (SQLCODE)* test after each statement.

Permissions

None.

Side effects

None.

Standards and compatibility

- **SQL/2003** Core feature.

Example

The following are examples of the WHENEVER statement:

```
EXEC SQL WHENEVER NOTFOUND GOTO done;
EXEC SQL WHENEVER SQLERROR
   {
      PrintError( &sqlca );
      return( FALSE );
   };
```

WHILE statement [T-SQL]

Use this statement to provide repeated execution of a statement or compound statement.

Syntax

WHILE *search-condition-statement*

Remarks

The WHILE conditional affects the execution of only a single SQL statement, unless statements are grouped into a compound statement between the keywords BEGIN and END.

The BREAK statement and CONTINUE statement can be used to control execution of the statements in the compound statement. The BREAK statement terminates the loop, and execution resumes after the END keyword marking the end of the loop. The CONTINUE statement causes the WHILE loop to restart, skipping any statements after the CONTINUE.

Permissions

None.

Side effects

None.

See also

- "LOOP statement" on page 672
- "CONTINUE statement [T-SQL]" on page 431

Standards and compatibility

- **SQL/2003** Transact-SQL extension.

Example

The following code illustrates the use of WHILE:

```
WHILE ( SELECT AVG(UnitPrice) FROM Products ) < $30
BEGIN
   UPDATE Products
   SET UnitPrice = UnitPrice + 2
   IF ( SELECT MAX(UnitPrice) FROM Products ) > $50
      BREAK
END
```

The BREAK statement breaks the WHILE loop if the most expensive product has a price above $50. Otherwise, the loop continues until the average price is greater than or equal to $30.

WINDOW clause

In a SELECT statement, use the WINDOW clause to define all or part of a window for use with window functions such as AVG and RANK.

Syntax

WINDOW *window-expression*, ...

window-expression : *new-window-name* **AS (** *window-spec* **)**

window-spec :
[*existing-window-name*]
[**PARTITION BY** *expression*, ...]
[**ORDER BY** *expression* [**ASC** | **DESC**], ...]
[{ **ROWS** | **RANGE** } { *window-frame-start* | *window-frame-between* }]

window-frame-start :
{ **UNBOUNDED PRECEDING**
 | *unsigned-integer* **PRECEDING**
 | **CURRENT ROW** }

window-frame-between :
BETWEEN *window-frame-bound1* **AND** *window-frame-bound2*

window-frame-bound :
window-frame-start
| **UNBOUNDED FOLLOWING**
| *unsigned-integer* **FOLLOWING**

Parameters

- **PARTITION BY clause** The PARTITION BY clause organizes the result set into logical groups based on the unique values of the specified expression. When this clause is used with window functions, the functions are applied to each partition independently. For example, if you follow PARTITION BY with a column name, the result set is partitioned by distinct values in the column.

  If this clause is omitted, the entire result set is considered a partition.

  The PARTITION BY *expression* cannot be an integer literal.

- **ORDER BY clause** The ORDER BY clause defines how to sort the rows in each partition of the result set. You can further control the order by specifying ASC for ascending order (the default) or DESC for descending order.

  The ORDER BY *expression* cannot be an integer literal.

  If this clause is omitted, SQL Anywhere returns rows in whatever order is most efficient, and the appearance of result sets may vary depending on when you last accessed the row.

- **ROWS clause and RANGE clause** Use either a ROWS or RANGE clause to express the size of the window. The window size can be one, many, or all rows of a partition. You can express the size of the window either in terms of a range of data values offset from the value in the current row (RANGE), or in terms of the number of physical rows offset from the current row (ROWS).

When using the RANGE clause, you must also specify an ORDER BY clause because range calculations require values to be sorted. The ORDER BY clause for ranges must contain one expression, and that expression must result in either a date or a numeric value.

If you do not specify a ROWS or RANGE clause, the database server uses default window sizes based on whether an ORDER BY clause is present. For information about the defaults, see "Defining a window" [*SQL Anywhere Server - SQL Usage*].

- **PRECEDING clause** Use the PRECEDING clause to define the first row of the window using the current row as a reference point. The starting row is expressed in terms of the number of rows preceding the current row. For example, `5 PRECEDING` sets the window to start with the fifth row preceding the current row.

  Use UNBOUNDED PRECEDING to set the first row in the window to be the first row in the partition.

- **BETWEEN clause** Use the BETWEEN clause to define the first and last row of the window, using the current row as a reference point. First and last rows are expressed in terms of the number of rows preceding and following the current row, respectively. For example, `BETWEEN 3 PRECEDING AND 5 FOLLOWING` sets the window to start with the third row preceding the current row, and end with the fifth row following the current row.

  Use BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING to set the first and last rows in the window to be the first and last row in the partition, respectively. This is equivalent to the default behavior if no ROW or RANGE clause is specified.

- **FOLLOWING clause** Use the FOLLOWING clause to define the last row of the window using the current row as a reference point. The last row is expressed in terms of the number of rows following the current row.

  Use UNBOUNDED FOLLOWING to set the last row in the window to be the last row in the partition.

Remarks

The WINDOW clause must appear before the ORDER BY clause in a SELECT statement.

With the exception of the LIST function, all aggregate functions can be used as window functions. However, ranking aggregate functions (RANK, DENSE_RANK, PERCENT_RANK, CUME_DIST, and ROW_NUMBER) require an ORDER BY clause, and do not allow a ROW or RANGE clause in the WINDOW clause or inline definition. For all other window functions, you can use any of the clauses.

For more information about defining and using windows to achieve the results you want, see "Defining a window" [*SQL Anywhere Server - SQL Usage*] and "Window definition: inlining and the WINDOW clause" [*SQL Anywhere Server - SQL Usage*].

See also

- "SELECT statement" on page 728
- "OLAP support" [*SQL Anywhere Server - SQL Usage*]

Standards and compatibility

- **SQL/2003** SQL/2003 features T611, T612.

Example

The following example returns an employee's salary and the average salary for all employees in the selected state. The results are ordered by state and then by surname.

```
SELECT EmployeeID, Surname, Salary, State,
  AVG( Salary ) OVER SalaryWindow
FROM Employees
WINDOW SalaryWindow AS ( PARTITION BY State )
ORDER BY State, Surname;
```

WRITETEXT statement [T-SQL]

Permits non-logged, interactive updating of an existing text or image column.

Syntax

WRITETEXT *table-name.column-name*
text-pointer [**WITH LOG**] *data*

Remarks

Updates an existing text or image value. The update is not recorded in the transaction log, unless the WITH LOG option is supplied. You cannot carry out WRITETEXT operations on views.

Permissions

None.

Side effects

WRITETEXT does not fire triggers, and by default WRITETEXT operations are not recorded in the transaction log.

See also

- "READTEXT statement [T-SQL]" on page 702
- "TEXTPTR function [Text and image]" on page 323

Standards and compatibility

- **SQL/2003** Transact-SQL extension.

Example

The following code fragment illustrates the use of the WRITETEXT statement. The SELECT statement in this example returns a single row. The example replaces the contents of the column_name column on the specified row with the value newdata.

```
EXEC SQL create variable textpointer binary(16);
EXEC SQL set textpointer =
   (  SELECT textptr(column_name)
      FROM table_name WHERE ID = 5 );
EXEC SQL writetext table_name.column_name
   textpointer 'newdata';
```

System Objects

This section describes SQL Anywhere tables, views, and procedures.

Tables

Contents

System tables

The structure of every database is described in several system tables. System tables are owned by the user SYS. The contents of these tables can be changed only by the database server. The UPDATE, DELETE, and INSERT commands cannot be used to modify the contents of these tables. Further, the structure of these tables cannot be changed using the ALTER TABLE and DROP commands.

System tables in SQL Anywhere are exposed via their corresponding views.

DUMMY system table

Column name	Column type	Column constraint	Table constraints
dummy_col	INTEGER	NOT NULL	

The DUMMY table is provided as a read-only table that always has exactly one row. This can be useful for extracting information from the database, as in the following example that gets the current user ID and the current date from the database.

```
SELECT USER, today(*) FROM SYS.DUMMY;
```

Use of SYS.DUMMY in the FROM clause is optional. If no table is specified in the FROM clause, the table is assumed to be SYS.DUMMY. The above example could be written as follows:

```
SELECT USER, today(*);
```

dummy_col This column is not used. It is present because a table cannot be created with no columns.

The cost of reading from the SYS.DUMMY table is less than the cost of reading from a similar user-created table because there is no latch placed on the table page of SYS.DUMMY.

Access plans are not constructed with scans of the SYS.DUMMY table. Instead, references to SYS.DUMMY are replaced with a Row Constructor algorithm, which virtualizes the table reference. This eliminates contention associated with the use of SYS.DUMMY. Note that DUMMY still appears as the table and/or correlation name in short, long, and graphical plans. See “RowConstructor algorithm (ROWS)” [*SQL Anywhere Server - SQL Usage*].

ISYSARTICLE system table

Each row in the ISYSARTICLE system table describes an article in a publication. See “SYSARTICLE system view” on page 990.

ISYSARTICLECOL system table

Each row in the ISYSARTICLECOL system table identifies a column in an article. See “SYSARTICLECOL system view” on page 991.

ISYSATTRIBUTE system table

This table is for internal use only.

ISYSATTRIBUTENAME system table

This table is for internal use only.

ISYSCAPABILITY system table

Each row in the ISYSCAPABILITY system table identifies a capability of a remote server. See "SYSCAPABILITY system view" on page 991.

ISYSCHECK system table

Each row in the ISYSCHECK system table identifies a named check constraint in a table. See "SYSCHECK system view" on page 992.

ISYSCOLPERM system table

Each row in the ISYSCOLPERM system table describes an UPDATE, SELECT, or REFERENCES permission on a column. See "SYSCOLPERM system view" on page 992.

ISYSCOLSTAT system table

The ISYSCOLSTAT system table contains the column statistics used by the optimizer. See "SYSCOLSTAT system view" on page 993.

NOTE
For databases created using SQL Anywhere version 11.0.0 and later, this table is always encrypted to protect the data from unauthorized access.

ISYSCONSTRAINT system table

Each row in the ISYSCONSTRAINT system table describes a named constraint for all tables except the system tables. See "SYSCONSTRAINT system view" on page 994.

ISYSDEPENDENCY system table

Each row in the ISYSDEPENDENCY system table describes a table or view dependency. See "SYSDEPENDENCY system view" on page 997.

ISYSDBFILE system table

Each row in the ISYSDBFILE system table describes a dbspace. See "SYSDBFILE system view" on page 995.

ISYSDBSPACE system table

Each row in the ISYSDBSPACE system table describes a dbspace. See "SYSDBSPACE system view" on page 996.

ISYSDBSPACEPERM system table

Each row in the ISYSDBSPACEPERM system table describes permission on a dbspace. See "SYSDBSPACEPERM system view" on page 996.

ISYSDOMAIN system table

Each of the predefined data types (also called **domains**) is assigned a unique number. The ISYSDOMAIN table is provided for informational purposes, to show the association between these numbers and the appropriate data types. This table is never changed. See "SYSDOMAIN system view" on page 997.

ISYSEVENT system table

Each row in the ISYSEVENT system table describes an event created with CREATE EVENT. See "SYSEVENT system view" on page 998.

ISYSEXTERNLOGIN system table

Each row in the ISYSEXTERNLOGIN system table describes an external login for remote data access. See "SYSEXTERNLOGIN system view" on page 1001.

NOTE
For databases created using SQL Anywhere version 11.0.0 and later, this table is always encrypted to protect the data from unauthorized access.

ISYSFILE system table

Each row in the ISYSFILE system table describes a dbspace for a database. Every database consists of one or more dbspaces; each dbspace corresponds to an operating system file. See "SYSFILE compatibility view (deprecated)" on page 1062.

ISYSFKEY system table

Each row in the ISYSFKEY system table describes a foreign key in the database. See "SYSFKEY system view" on page 1002.

ISYSGROUP system table

Each row in the ISYSGROUP system table defines a member of a group. This table describes the many-to-many relationship between groups and members. See "SYSGROUP system view" on page 1003.

ISYSHISTORY system table

Each row in the ISYSHISTORY system table indicates a time in which the database was started with a different version of the software and/or on a different platform. See "SYSHISTORY system view" on page 1003.

ISYSIDX system table

Each row in the ISYSIDX system table describes an index in the database. See "SYSIDX system view" on page 1005.

ISYSIDXCOL system table

Each row in the ISYSIDXCOL system table describes a column in an index. See "SYSIDXCOL system view" on page 1006.

ISYSJAR system table

Each row in the ISYSJAR system table defines a JAR file in the system. See "SYSJAR system view" on page 1007.

ISYSJARCOMPONENT system table

Each row in the ISYSJAR system table defines a JAR file component. See "SYSJARCOMPONENT system view" on page 1008.

ISYSJAVACLASS system table

Each row in the ISYSJAVACLASS system table describes a Java class. See "SYSJAVACLASS system view" on page 1009.

ISYSLOGINMAP system table

The ISYSLOGINMAP system table contains all the User Profile names that can be used to connect to the database using either an integrated login or a Kerberos login. As a security measure, only users with DBA authority can view the contents of this table. See "SYSLOGINMAP system view" on page 1009.

ISYSLOGINPOLICY system table

Each row in the ISYSLOGINPOLICY system table describes a login policy. See "SYSLOGINPOLICY system view" on page 1010.

ISYSLOGINPOLICYOPTION system table

Each row in the ISYSLOGINPOLICYOPTION system table describes an option for a login policy. See "SYSLOGINPOLICYOPTION system view" on page 1010.

ISYSMVOPTION system table

Each row in the ISYSMVOPTION system table describes an option for a materialized view. See "SYSMVOPTION system view" on page 1011.

ISYSMVOPTIONNAME system table

Each row in the ISYSMVOPTIONNAME system table provides the name of a materialized view listed in ISYSMVOPTION. See "SYSMVOPTIONNAME system view" on page 1011.

ISYSOBJECT system table

Each row in the ISYSOBJECT system view describes an object in the database. Examples of database objects include tables, views, columns, indexes, and procedures. See "SYSOBJECT system view" on page 1012.

ISYSOPTION system table

Each row in the ISYSOPTION system table describes the settings for an option for one user ID. Options settings are stored in the ISYSOPTION table by the SET command, and each user can have their own setting for each option. See "SYSOPTION system view" on page 1013.

ISYSOPTSTAT system table

The ISYSOPTSTAT system table stores the cost model calibration information as computed by the ALTER DATABASE CALIBRATE statement. See "SYSOPTSTAT system view" on page 1014.

ISYSPHYSIDX system table

Each row in the ISYSPHYSIDX system table describes a physical index in the database. See "SYSPHYSIDX system view" on page 1014.

ISYSPROCEDURE system table

Each row in the ISYSPROCEDURE system table describes a procedure in the database. See "SYSPROCEDURE system view" on page 1015.

ISYSPROCPARM system table

Each row in the ISYSPROCPARM system table describes a parameter to a procedure in the database. See "SYSPROCPARM system view" on page 1016.

ISYSPROCPERM system table

Each row in the ISYSPROCPERM system table describes a user granted permission to call one procedure. See "SYSPROCPERM system view" on page 1018.

ISYSPROXYTAB system table

Each row in the ISYSPROXYTAB system table describes a proxy table. See “SYSPROXYTAB system view” on page 1018.

ISYSPUBLICATION system table

Each row in the ISYSPUBLICATION system table describes a SQL Remote or MobiLink publication. See “SYSPUBLICATION system view” on page 1019.

ISYSREMARK system table

Each row in the ISYSREMARK system table describes a remark (or comment) for an object. See “SYSREMARK system view” on page 1020.

ISYSREMOTEOPTION system table

Each row in the ISYSREMOTEOPTION system table describes the values of a SQL Remote message link parameter. See “SYSREMOTEOPTION system view” on page 1020.

ISYSREMOTEOPTIONTYPE system table

Each row in the ISYSREMOTEOPTIONTYPE system table describes one of the SQL Remote message link parameters. See “SYSREMOTEOPTIONTYPE system view” on page 1020.

ISYSREMOTETYPE system table

The ISYSREMOTETYPE system table contains information about SQL Remote. See “SYSREMOTETYPE system view” on page 1021.

ISYSREMOTEUSER system table

Each row in the ISYSREMOTEUSER system table describes a user ID with REMOTE permissions (a subscriber), together with the status of SQL Remote messages that were sent to and from that user. See “SYSREMOTEUSER system view” on page 1021.

ISYSSCHEDULE system table

Each row in the ISYSSCHEDULE system table describes a time at which an event is to fire, as specified by the SCHEDULE clause of CREATE EVENT. See “SYSSCHEDULE system view” on page 1023.

ISYSSERVER system table

Each row in the ISYSSERVER system table describes a remote server. See “SYSSERVER system view” on page 1024.

ISYSSOURCE system table

Each row in the ISYSSOURCE system view contains the source for an object listed in the ISYSOBJECT system table. See “SYSSOURCE system view” on page 1024.

ISYSSQLSERVERTYPE system table

The ISYSSQLSERVERTYPE system table contains information relating to compatibility with Adaptive Server Enterprise. See “SYSSQLSERVERTYPE system view” on page 1025.

ISYSSUBSCRIPTION system table

Each row in the ISYSSUBSCRIPTION system table describes a subscription from one user ID (which must have REMOTE permissions) to one publication. See “SYSSUBSCRIPTION system view” on page 1025.

ISYSSYNC system table

This table contains information relating to MobiLink synchronization. Some columns in this table contain potentially sensitive data. For that reason, access to this table is restricted to users with DBA authority. The SYSSYNC2 view provides public access to the data in this table except for the potentially sensitive columns. See “SYSSYNC system view” on page 1026.

ISYSSYNCSCRIPT system table

This table contains information relating to MobiLink synchronization scripts. See “SYSSYNCSCRIPT system view” on page 1027.

ISYSTAB system table

Each row in the ISYSTAB system table describes one table in the database. See "SYSTAB system view" on page 1028.

ISYSTABCOL system table

Each row in the ISYSTABCOL system table describes a column of a table in the database. See "SYSTABCOL system view" on page 1030.

ISYSTEXTCONFIG system table

Each row in the ISYSTEXTCONFIG system table describes a text configuration, for use with the full text search feature. See "SYSTEXTCONFIG system view" on page 1034.

ISYSTEXTIDX system table

Each row in the ISYSTEXTIDX system table describes a text index, for use with the full text search feature. See "SYSTEXTIDX system view" on page 1035.

ISYSTEXTIDXTAB system table

Each row in the ISYSTEXTIDXTAB system table describes a text index, for use with the full text search feature. See "SYSTEXTIDX system view" on page 1035.

ISYSTABLEPERM system table

Each row in the ISYSTABLEPERM system table corresponds to one table, one user ID granting the permission (**grantor**) and one user ID granted the permission (**grantee**). See "SYSTABLEPERM system view" on page 1032.

ISYSTRIGGER system table

Each row in the ISYSTRIGGER system table describes a trigger in the database. See "SYSTRIGGER system view" on page 1037.

ISYSTYPEMAP system table

The ISYSTYPEMAP system table contains the compatibility mapping values for the ISYSSQLSERVERTYPE system table. See "SYSTYPEMAP system view" on page 1038.

ISYSUSER system table

Each row in the ISYSUSER system table describes a user in the system. See "SYSUSER system view" on page 1039.

NOTE
For databases created using SQL Anywhere version 11.0.0 and later, this table is always encrypted to protect the data from unauthorized access.

ISYSUSERAUTHORITY system table

Each row in the ISYSUSERAUTHORITY system table describes the authority granted to a user. See "SYSUSERAUTHORITY system view" on page 1040.

ISYSUSERMESSAGE system table

Each row in the ISYSUSERMESSAGE system table holds a user-defined message for an error condition. See "SYSUSERMESSAGE system view" on page 1040.

ISYSUSERTYPE system table

Each row in the ISYSUSERTYPE system table describes a user-defined data type. See "SYSUSERTYPE system view" on page 1041.

ISYSVIEW system table

Each row in the ISYSVIEW system table describes a view in the database. See "SYSVIEW system view" on page 1041.

ISYSWEBSERVICE system table

Each row in the ISYSWEBSERVICE system table describes a web service. See "SYSWEBSERVICE system view" on page 1043.

Diagnostic tracing tables

Following are the main tables that are used for application profiling and diagnostic tracing. These tables are owned by the dbo user. For many of these tables, there exists a global shared temporary table with a similar name and schema. For example, the sa_diagnostic_blocking table has a global temporary table counterpart, sa_tmp_diagnostic_blocking table, which has the same schema. During a tracing session, diagnostic data is written to these temporary tables. Because temporary tables are not logged, they provide superior performance during a tracing session, where it is important to minimize the impact on the server.

See also

- "Application profiling" [*SQL Anywhere Server - SQL Usage*]
- "Advanced application profiling using diagnostic tracing" [*SQL Anywhere Server - SQL Usage*]

sa_diagnostic_auxiliary_catalog table

The sa_diagnostic_auxiliary_catalog table is owned by the dbo user, and is used to map database objects between the production database and tracing database. Objects include user tables, procedures, and functions. This table is used primarily by the Index Consultant and the TRACED_PLAN function.

Columns

Column name	Column type	Column constraint	Table constraints
original_object_id	UNSIGNED BIGINT	NOT NULL	Primary key.
local_object_id	UNSIGNED BIGINT	NOT NULL	Unique.
pages_if_table	UNSIGNED INT		
rows_if_table	UNSIGNED BIGINT		

original_object_id The object ID of this object in the main tracing database.

local_object_id The object ID of this object in the auxiliary tracing database.

pages_if_table If the object is a table, this is the number of pages in the table. If the object is not a table, this value is NULL.

rows_if_table If the object is a table, this is the number of rows in the table. If the object is not a table, this value is NULL.

See also

- "TRACED_PLAN function [Miscellaneous]" on page 326
- "Index Consultant" [*SQL Anywhere Server - SQL Usage*]

sa_diagnostic_blocking table

The sa_diagnostic_blocking table is owned by the dbo user, and records blocking events. If logging of blocking events is enabled, a row is inserted in this table each time a connection is blocked while trying to access a resource. Typically, this is caused by either a table or a row lock. A large number of blocks may indicate that you should examine the concurrency in your application to reduce contention for tables and rows.

There are two versions of this table: sa_diagnostic_blocking, and sa_tmp_diagnostic_blocking.

Columns

Column name	Column type	Column constraint	Table constraints
logging_session_id	UNSIGNED INT	NOT NULL	Foreign key references sa_diagnostic_cursor. Foreign key references sa_diagnostic_request.
lock_id	UNSIGNED BIGINT	NOT NULL	
request_id	UNSIGNED BIGINT		Foreign key references sa_diagnostic_request.
cursor_id	UNSIGNED BIGINT		Foreign key references sa_diagnostic_cursor.
original_table_object_id	UNSIGNED BIGINT		
rowid	UNSIGNED BIGINT		
block_time	TIMESTAMP	NOT NULL	
unblock_time	TIMESTAMP		
blocked_by	UNSIGNED INT	NOT NULL	

logging_session_id A number uniquely identifying the logging session during which the diagnostic information was gathered.

lock_id The ID of the lock that caused the blocking if a row or table lock caused the block, otherwise NULL.

request_id The ID of the request that was blocked if the block did not occur because of a cursor, otherwise NULL. This value corresponds to the ID assigned to the request in sa_diagnostic_request.

cursor_id The ID of the cursor if the block occurred because of a cursor, otherwise NULL. This value corresponds to the ID assigned to the cursor in sa_diagnostic_cursor.

original_table_object_id If the block occurred because of a table lock, the ID of the table on which the block occurred, otherwise NULL.

rowid If the block occurred because of a row lock, the ID of the row on which the block occurred, otherwise NULL.

block_time The time at which the block occurred.

unblock_time The time at which the block ended.

blocked_by The ID of the connection that held the lock, causing the block.

See also

- "Transaction blocking and deadlock" [*SQL Anywhere Server - SQL Usage*]
- "How locking works" [*SQL Anywhere Server - SQL Usage*]

sa_diagnostic_cachecontents table

The sa_diagnostic_cachecontents table is owned by the dbo user. When diagnostic tracing is enabled, periodic snapshots of the cache contents are taken. The sa_diagnostic_cachecontents table records the number of table pages for each table in the cache at the time the snapshot was taken, and the number of rows in each table. The optimizer can use this information to recreate the conditions under which a query was originally optimized, and then make optimization decisions.

Data in the sa_diagnostic_cachecontents table is updated every 20 seconds, as long as there is query activity.

There are two versions of this table: sa_diagnostic_cachecontents, and sa_tmp_diagnostic_cachecontents.

Columns

Column name	Column type	Column constraint	Table constraints
logging_session_id	UNSIGNED INT	NOT NULL	
"time"	TIMESTAMP	NOT NULL	Primary key.
original_table_object_id	UNSIGNED BIGINT	NOT NULL	Primary key.
pages_in_cache	UNSIGNED INT	NOT NULL	
num_table_pages	UNSIGNED INT	NOT NULL	
num_table_rows	UNSIGNED BIGINT	NOT NULL	

logging_session_id A number uniquely identifying the logging session during which the diagnostic information was gathered.

"time" The time at which the snapshot of the cache was taken.

original_table_object_id The object ID of each table represented in the snapshot.

pages_in_cache For a specified table in the snapshot, the total number of pages in cache at the moment of the snapshot.

num_table_pages For a specified table in the snapshot, the total number of pages for the table.

num_table_rows For a specified table in the snapshot, the total number of rows in the table.

sa_diagnostic_connection table

The sa_diagnostic_connection table is owned by the dbo user, and has one row for every database connection that is active during the logging session. Connect and disconnect times, if they occur within the logging session, can be derived from the sa_diagnostic_request table.

Most of the values in this table mirror values of connection properties.

There are two versions of this table: sa_diagnostic_connection, and sa_tmp_diagnostic_connection.

Columns

Column name	Column type	Column constraint	Table constraints
logging_session_id	UNSIGNED INT	NOT NULL	Primary key.
connection_number	UNSIGNED INT		Primary key.
connection_name	LONG VARCHAR		
user_name	LONG VARCHAR		
comm_link	CHAR(40)		
node_address	LONG VARCHAR		
appinfo	LONG VARCHAR		

logging_session_id A number uniquely identifying the logging session during which the diagnostic information was gathered.

connection_number A number assigned by the database server to identify the user's connection to the database. This value reflects the value of the Number connection property.

connection_name Optional name property for the connection. This value reflects the value of the Name connection property.

user_name The name of the user connected to the database.

comm_link Specifies the client-side network protocol options. This value reflects the value of the CommLinks connection property.

node_address The node for the client in a client/server connection. This value reflects the value of the NodeAddress connection property.

appinfo Information about the client process, such as the IP address of the client computer, the operating system it is running on, and so on. This value reflects the value of the AppInfo connection property.

See also

- "Connection properties" [*SQL Anywhere Server - Database Administration*]

sa_diagnostic_cursor table

The sa_diagnostic_cursor table is owned by the dbo user. Each row describes either an internal or external cursor opened during the logging session.

There are two versions of this table: sa_diagnostic_cursor, and sa_tmp_diagnostic_cursor.

Columns

Column name	Column type	Column constraint	Table constraints
logging_session_id	UNSIGNED INT	NOT NULL	Primary key. Foreign key references sa_diagnostic_query.
cursor_id	UNSIGNED BIGINT	NOT NULL	Primary key.
query_id	UNSIGNED BIGINT	NOT NULL	Foreign key references sa_diagnostic_query.
isolation_level	TINYINT		
flags	UNSIGNED INT		
forward_fetches	UNSIGNED INT		
reverse_fetches	UNSIGNED INT		
absolute_fetches	UNSIGNED INT		
first_fetch_time_ms	UNSIGNED INT		
total_fetch_time_ms	UNSIGNED INT		
plan_xml	LONG VARCHAR		

logging_session_id A number uniquely identifying the logging session during which the diagnostic information was gathered.

cursor_id A unique number identifying the cursor.

query_id Identifies the query over which this cursor ranges.

isolation_level Isolation level at which this cursor was opened.

flags Internal use.

forward_fetches Number of forward fetches, including prefetches, done on the cursor.

reverse_fetches Number of reverse fetches, including prefetches, done on the cursor.

absolute_fetches Number of absolute fetches done on the cursor.

first_fetch_time_ms Duration of time spent fetching the first row.

total_fetch_time_ms Duration of time spent fetching. This value does not include application processing time between actual fetches (think time).

plan_xml Detailed plan for cursors that were dumped at the time the cursor was closed. These plans contain detailed statistics where appropriate.

See also

- "Introduction to cursors" [*SQL Anywhere Server - Programming*]

sa_diagnostic_deadlock table

The sa_diagnostic_deadlock table is owned by the dbo user. When diagnostic tracing is enabled and is set to include tracing of deadlock events, a set of rows is inserted into this table every time a deadlock occurs (one row for each connection that was part of the deadlock is inserted). The set of all rows that comprise a single deadlock event is uniquely identified by a snapshot_id.

Columns

Column name	Column type	Column constraint	Table constraints
logging_session_id	UNSIGNED INT	NOT NULL	
snapshot_id	UNSIGNED BIGINT	NOT NULL	
snapshot_at	TIMESTAMP	NOT NULL	
waiter	UNSIGNED INT	NOT NULL	
request_id	UNSIGNED BIGINT		
original_table_object_id	UNSIGNED BIGINT		
rowid	UNSIGNED BIGINT		

Column name	Column type	Column constraint	Table constraints
owner	UNSIGNED INT	NOT NULL	
rollback_operation_count	UNSIGNED INT	NOT NULL	

logging_session_id A number uniquely identifying the logging session during which the diagnostic information was gathered.

snapshot_id A number identifying which deadlock event this row is a part of. Note that this column has nothing to do with snapshot isolation.

snapshot_at The time at which the deadlock occurred.

waiter The connection number of the connection that this row represents.

request_id The ID of the request that this connection was processing when the deadlock occurred.

original_table_object_id The object ID of the table on which this connection was blocked.

rowid The record ID of the row on which this connection was blocked.

owner The connection number of the connection that locked the desired row.

rollback_operation_count The number of uncommitted operations.

See also

- "Transaction blocking and deadlock" [*SQL Anywhere Server - SQL Usage*]

sa_diagnostic_hostvariable table

The sa_diagnostic_hostvariable table is owned by the dbo user, and contains the values of host variables used by the specified cursor.

There are two versions of this table: sa_diagnostic_hostvariable, and sa_tmp_diagnostic_hostvariable.

Columns

Column name	Column type	Column constraint	Table constraints
logging_session_id	UNSIGNED INT	NOT NULL	Primary key. Foreign key references sa_diagnostic_request.
request_id	UNSIGNED BIGINT	NOT NULL	Primary key. Foreign key references sa_diagnostic_request.
cursor_id	UNSIGNED BIGINT		

Column name	Column type	Column constraint	Table constraints
hostvar_num	UNSIGNED SMALLINT	NOT NULL	Primary key.
hostvar_type	UNSIGNED TI-NYINT	NOT NULL	
hostvar_value	LONG VAR-CHAR		

logging_session_id A number uniquely identifying the logging session during which the diagnostic information was gathered.

request_id The ID of the request to which the host variables belong.

cursor_id The ID of the cursor to which the host variables pertain.

hostvar_num The ordinal position of the host variable in the SQL statement.

hostvar_type The domain number of the host variable, typically a string, integer, or a float.

hostvar_value A string representing the value of the host variable. Even if the host variable is an integer or a float, the value is still represented here as a string.

See also

- "Using host variables" [*SQL Anywhere Server - Programming*]

sa_diagnostic_internalvariable table

The sa_diagnostic_internalvariable table is owned by the dbo user, and contains the values of internal (local) variables used by a given statement. This table is primarily used by the Index Consultant, and the traced_plan function.

There are two versions of this table: sa_diagnostic_internalvariable, and sa_tmp_diagnostic_internalvariable.

Columns

Column name	Column type	Column constraint	Table constraints
logging_session_id	UNSIGNED INT	NOT NULL	
request_id	UNSIGNED BIGINT		
rowvariable_id	UNSIGNED INT		
variable_domain	UNSIGNED SMALLINT		

Column name	Column type	Column constraint	Table constraints
variable_name	CHAR(128)		
variable_value	LONG VARCHAR		

logging_session_id A number uniquely identifying the logging session during which the diagnostic information was gathered.

request_id The ID of the request that contains the internal variable.

rowvariable_id The column number in the row variable of this value.

variable_domain The data type of the internal variable.

variable_name The name of the internal variable.

variable_value A string representing the value of the internal variable.

See also

- “Local variables” on page 66

sa_diagnostic_query table

The sa_diagnostic_query table is owned by the dbo user, and stores optimization information for queries, especially the context in which they were optimized. A row in this table represents an invocation of the optimizer for a query. Plans captured at optimization time are stored here.

Some of the values in this table mirror database option values.

There are two versions of this table: sa_diagnostic_query, and sa_tmp_diagnostic_query.

Columns

Column name	Column type	Column constraint	Table constraints
logging_session_id	UNSIGNED INT	NOT NULL	Primary key. Foreign key references sa_diagnostic_statement
query_id	UNSIGNED BIGINT	NOT NULL	Primary key. Foreign key references sa_diagnostic_statement.
statement_id	UNSIGNED BIGINT	NOT NULL	
user_object_id	UNSIGNED BIGINT	NOT NULL	
start_time	TIMESTAMP	NOT NULL	

Column name	Column type	Column constraint	Table constraints
cache_size_bytes	UNSIGNED BIGINT		
optimization_goal	TINYINT		
optimization_level	TINYINT		
user_estimates	TINYINT		
optimization_workload	TINYINT		
available_requests	TINYINT		
active_requests	TINYINT		
max_tasks	TINYINT		
used_bypass	TINYINT		
estimated_cost_ms	TINYINT		
plan_explain	LONG VARCHAR		
plan_xml	LONG VARCHAR		
sql_rewritten	LONG VARCHAR		

logging_session_id The ID of the logging session during which the query or request occurred.

query_id A number uniquely identifying the query.

statement_id A number uniquely identifying a statement in a query.

user_object_id The object ID of the user under which this query was executed. If the query was run from a procedure, this would be the user ID of the procedure owner.

start_time The time at which this query was optimized.

cache_size_bytes The size, in bytes, of the cache at the time this query was optimized.

optimization_goal Determines whether query processing is optimized towards returning the first row quickly, or minimizing the cost of returning the complete result set. This value reflects the value of the optimization_goal database option.

To see possible values for this column, see "optimization_goal option [database]" [*SQL Anywhere Server - Database Administration*].

optimization_level Controls the amount of effort made by the SQL Anywhere query optimizer to find an access plan for a SQL statement. This value reflects the value of the optimization_level database option.

To see possible values for this column, see "optimization_level option [database]" [*SQL Anywhere Server - Database Administration*].

user_estimates Controls whether user selectivity estimates in query predicates are respected or ignored by the query optimizer. This value reflects the value of the user_estimates database option.

To see possible values for this column, see "user_estimates option [database]" [*SQL Anywhere Server - Database Administration*].

optimization_workload Determines whether query processing is optimized towards a workload that is a mix of updates and reads or a workload that is predominantly read-based. This value reflects the value of the optimization_workload database option.

To see possible values for this column, see "optimization_workload option [database]" [*SQL Anywhere Server - Database Administration*].

available_requests Used internally to compute the level of intra-query parallelism.

active_requests Used internally to compute the level of intra-query parallelism.

max_tasks Used internally to compute the level of intra-query parallelism.

used_bypass Whether a simple query bypass was used. A value of 1 indicates a bypass was used; a value of 0 indicates that the query was fully optimized.

estimated_cost_ms The estimated cost, in milliseconds.

plan_explain A text plan representation of this query.

plan_xml A graphical plan representation of the query (if one was recorded).

sql_rewritten Text of a query after applying optimizations. A value will only be present in this column if optimization logging is enabled.

See also

- "Database options" [*SQL Anywhere Server - Database Administration*]
- "How the optimizer works" [*SQL Anywhere Server - SQL Usage*]

sa_diagnostic_request table

The sa_diagnostic_request table is owned by the dbo user, and is the master table for all requests. A request is an event related to query processing and generally includes:

- connect or disconnect events
- statement executions
- statement preparations
- open or drop cursor events

There are two versions of this table: sa_diagnostic_request and sa_tmp_diagnostic_request.

Columns

Column name	Column type	Column con-straint	Table constraints
logging_ses-sion_id	UNSIGNED INT	NOT NULL	Primary key. Foreign key references sa_diagnostic_connection. Foreign key references sa_diagnostic_cursor. Foreign key references sa_diagnostic_query. Foreign key references sa_diagnostic_statement.
request_id	UNSIGNED BI-GINT	NOT NULL	Primary key.
start_time	TIMESTAMP	NOT NULL	
finish_time	TIMESTAMP	NOT NULL	
duration_ms	UNSIGNED INT	NOT NULL	
connection_num-ber	UNSIGNED INT		Foreign key references sa_diagnostic_connection.
request_type	UNSIGNED SMALLINT		
statement_id	UNSIGNED BI-GINT		Foreign key references sa_diagnostic_statement.
query_id	UNSIGNED BI-GINT		Foreign key references sa_diagnostic_query.
cursor_id	UNSIGNED BI-GINT		Foreign key references sa_diagnostic_cursor.
sql_code	SMALLINT		

logging_session_id The logging session during which the request occurred.

request_id A number uniquely identifying the request.

start_time The time at which the event started.

finish_time For statement execution, the time when the statement completed; otherwise, NULL.

duration_ms The duration of the event in milliseconds.

connection_number The ID of the connection that caused the event to happen.

request_type The type of request. Values include:

Value	Description
1	Start of new tracing session
2	Statement execution
3	Cursor open
4	Cursor close
5	Connect
6	Disconnect

statement_id If the event was statement-related, the ID assigned to the statement for tracing purposes.

query_id If the event was query-related, the ID assigned to the query for tracing purposes.

cursor_id If the event was cursor-related, the ID assigned to the cursor for tracing purposes.

sql_code Since rows in this table represent operations on statements, cursors, or queries, most return a SQL code. This column contains the SQL code returned. If a SQL code of 0 is returned, the column contains NULL.

sa_diagnostic_statement table

The sa_diagnostic_statement table is owned by the dbo user, and stores the text of statements. A row in this table represents a SQL statement that was executed by the server. Such statements may have been issued by an external source, such as a client request, or by an internal source such as a procedure, trigger, or user-defined function. Internal statements only appear here once per session.

There are two versions of this table: sa_diagnostic_statement, and sa_tmp_diagnostic_statement.

Columns

Column name	Column type	Column constraint	Table constraints
logging_session_id	UNSIGNED INT	NOT NULL	Primary key.
statement_id	UNSIGNED BIGINT	NOT NULL	Primary key.
database_object	UNSIGNED BIGINT		
line_number	UNSIGNED SMALLINT		
signature	UNSIGNED INT		

Column name	Column type	Column constraint	Table constraints
statement_text	LONG VARCHAR	NOT NULL	

logging_session_id The logging session during which the statement was submitted.

statement_id A unique number assigned to the statement for tracing purposes.

database_object If the statement came from a procedure, trigger, or function, this is the ID as specified in the ISYSOBJECT system table.

line_number If the statement formed part of a compound statement, this reflects the ordinal position of the statement within the compound statement.

signature Used internally to group similar queries.

statement_text The statement text.

sa_diagnostic_statistics table

The sa_diagnostic_statistics table is owned by the dbo user, and contains a history of performance counters maintained in the server. Each row represents the value of a given performance counter at a given moment in time.

There are two versions of this table: sa_diagnostic_statistics, and sa_tmp_diagnostic_statistics.

Columns

Column name	Column type	Column constraint	Table constraints
logging_session_id	UNSIGNED INT	NOT NULL	
"time"	TIMESTAMP	NOT NULL	
counter_id	UNSIGNED SMALLINT	NOT NULL	
type	TINYINT	NOT NULL	
connection_number	UNSIGNED INT	NOT NULL	
counter_value	UNSIGNED INT	NOT NULL	

logging_session_id A number uniquely identifying the logging session during which the diagnostic information was gathered.

"time" The time at which the performance counter value was captured.

counter_id A number uniquely identifying the performance counter. You can get the name of the property that this counter_id represents using the PROPERTY_NAME function.

type Indicates whether this is a database, server, or connection statistic. Possible values are 0 for server, 1 for database, 2 for connection, and 4 for external database.

connection_number In the case of a connection statistic, the connection number from which this property was captured. In the case of an extended database statistic, the file number for the file from which this property was captured. Otherwise, the value is 0.

counter_value The value of the performance counter.

See also

- "PROPERTY_NAME function [System]" on page 268

sa_diagnostic_tracing_level table

The sa_diagnostic_tracing_level table is owned by the dbo user, and each row in this table is a condition that determines what kind of diagnostic information to send to the tracing database. If a piece of logging data meets the conditions of one or more rows in this table, then the corresponding data is logged.

Data in this table is populated using the CONNECT TRACING or REFRESH TRACING LEVELS statements.

Columns

Column name	Column type	Column constraint	Table constraints
id	UNSIGNED INT	NOT NULL	Primary key.
scope	CHAR(32)	NOT NULL	
identifier	CHAR(128)		
trace_type	CHAR(32)	NOT NULL	
trace_condition	CHAR(32)		
value	UNSIGNED INT		
enabled	BIT	NOT NULL	

scope The scope of the diagnostic tracing, as listed below. To see the description for each scope, see "Diagnostic tracing scopes" [*SQL Anywhere Server - SQL Usage*].

- DATABASE
- ORIGIN
- USER
- CONNECTION_NAME
- CONNECTION_NUMBER
- FUNCTION
- PROCEDURE
- EVENT
- TRIGGER
- TABLE

id For internal use only.

identifier The identifier for the scope. This value changes, depending on the specified scope. For example:

- if *scope* is DATABASE, *identifier* may not be present.
- if *scope* is ORIGIN, *identifier* must be either Internal or External.
- if *scope* is USER, *identifier* is the ID of the user.
- if *scope* is CONNECTION_NAME, or CONNECTION_NUMBER, *identifier* is the name or number, respectively, for the connection.
- if *scope* is FUNCTION, PROCEDURE, EVENT, TRIGGER, or TABLE, *identifier* is the fully qualified identifier for the object.

trace_type The type of data to trace for the specified scope, as listed below. To see the description for each trace type, see "Diagnostic tracing types" [*SQL Anywhere Server - SQL Usage*].

- VOLATILE_STATISTICS
- NONVOLATILE_STATISTICS
- CONNECTION_STATISTICS
- BLOCKING
- PLANS
- PLANS_WITH_STATISTICS
- STATEMENTS
- STATEMENTS_WITH_VARIABLES
- OPTIMIZATION_LOGGING
- OPTIMIZATION_LOGGING_WITH_PLANS

condition Applies only to plans, and controls whether to trace large, expensive queries, or queries for which the optimizer did not make optimal choices. Possible values are listed below. To see a description of each condition, see "Diagnostic tracing conditions" [*SQL Anywhere Server - SQL Usage*].

- NONE, or NULL
- SAMPLE_EVERY
- ABSOLUTE_COST
- RELATIVE_COST_DIFFERENCE

condition_value The value associated with the *condition*. For example, if *condition* is SAMPLE_EVERY, the *condition_value* would be a positive integer reflecting time in milliseconds. Additional rules are as follows:

- If *condition* is NULL or NONE, there is no *condition_value*.
- If *condition* is ABSOLUTE_COST, *condition_value* reflects the total actual cost of executing the statement, in milliseconds.
- If *condition* is RELATIVE_COST_DIFFERENCE, *condition_value* reflects the cost of executing, as a percentage of the estimated cost.

enabled Whether the row is enabled. That is, whether the tracing settings in the row are active. 1 is enabled; 0 is disabled.

See also

- "ATTACH TRACING statement" on page 406
- "REFRESH TRACING LEVEL statement" on page 708

Other tables

Following is information about other tables such as system tables used by Java in the database and SQL Remote.

RowGenerator table (dbo)

The dbo.RowGenerator table is provided as a read-only table that has 255 rows. This table can be useful for queries which produce small result sets and which need a range of numeric values.

The RowGenerator table is used by system procedures and views, and should not be modified in any way.

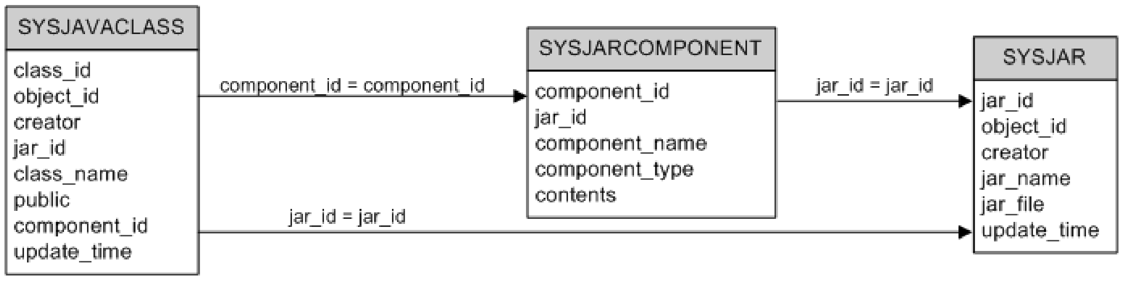
You can also use the sa_rowgenerator system procedure to generate a range of numeric values. For more information about using the sa_rowgenerator system procedure, including examples, see "sa_rowgenerator system procedure" on page 930.

Column name	Column type	Column constraint	Underlying table constraints
row_num	SMALLINT	NOT NULL	

row_num A value between 1 and 255.

Java system tables

The system tables that are used for Java are listed below. Foreign key relations between tables are indicated by arrows: the arrow leads from the foreign table to the primary table.



MobiLink system tables

For information about the MobiLink system tables, see "MobiLink server system tables" [*MobiLink - Server Administration*].

SQL Remote system tables

For information about the SQL Remote system tables, see "SQL Remote system tables" [*SQL Remote*].

UltraLite system tables

For information about the UltraLite system tables, see "UltraLite system tables" [*UltraLite - Database Management and Reference*].

System procedures

Contents

Introduction to system procedures

The following sections document the system procedures included with SQL Anywhere. A few system procedures, such as sa_get_table_definition, are implemented as functions. However, because they are used in the same context and manner as system procedures, they are included with the system procedures, and their naming is similar to the system procedures (sa_xxx).

SQL Anywhere includes the following kinds of system procedures:

- System procedures, for displaying system information in tabular form.
- SOAP and HTTP services system procedures, for supporting web services.
- MAPI and SMTP system procedures, for sending electronic mail.
- Transact-SQL system and catalog procedures. See "Adaptive Server Enterprise system and catalog procedures" on page 833.

To view details about system procedures and functions

1. Connect to a database with DBA authority.
2. Right-click the database and then choose **Configure Owner Filter**.
3. Click **DBO** and then click **OK**.
4. In the left pane, double-click **Procedures & Functions**.
5. In the left pane, select the procedure and in the right pane click the **SQL** tab.

Web services system procedures

The following system procedures are for use with web services:

- "sa_http_header_info system procedure" on page 883
- "sa_http_php_page system procedure" on page 884
- "sa_http_php_page_interpreted system procedure" on page 885
- "sa_http_variable_info system procedure" on page 886
- "sa_set_http_header system procedure" on page 944
- "sa_set_http_option system procedure" on page 946
- "sa_set_soap_header system procedure" on page 948

There are also many functions available for web services. See "Web services functions" on page 130.

MAPI and SMTP procedures

SQL Anywhere includes system procedures for sending electronic mail using the Microsoft Messaging API standard (MAPI) or the Internet standard Simple Mail Transfer Protocol (SMTP). These system procedures are implemented as extended system procedures: each procedure calls a function in an external DLL.

These procedures are owned by the dbo user ID. Users must be granted EXECUTE permission before they can use these procedures, unless they already have DBA authority.

To use the MAPI or SMTP system procedures, a MAPI or SMTP email system must be accessible from the database server computer.

The MAPI and SMTP system procedures are:

- **xp_startmail** Starts a mail session in a specified mail account by logging onto the MAPI message system. See "xp_startmail system procedure" on page 983.
- **xp_startsmtp** Starts a mail session in a specified mail account by logging onto the SMTP message system. See "xp_startsmtp system procedure" on page 984.
- **xp_sendmail** Sends a mail message to specified users. See "xp_sendmail system procedure" on page 978.
- **xp_stopmail** Closes the MAPI mail session. See "xp_stopmail system procedure" on page 985.
- **xp_stopsmtp** Closes the SMTP mail session. See "xp_stopsmtp system procedure" on page 986.

Example

The following procedure notifies a set of people that a backup has been completed.

```
CREATE PROCEDURE notify_backup( )
BEGIN
   CALL xp_startmail( mail_user='ServerAccount',
                      mail_password='ServerPassword'
                   );
   CALL xp_sendmail( recipient='IS Group',
                     subject='Backup',
                     "message"='Backup completed'
                   );
   CALL xp_stopmail( )
END;
```

Return codes for MAPI and SMTP system procedures

The following codes may be returned by either the MAPI or the SMTP system procedures:

Return code	Meaning
0	Success
2	xp_startmail or xp_startsmtp failed
3	xp_stopmail or xp_stopsmtp failed
5	xp_sendmail failed
12	Attachment not found
15	Insufficient memory

Return code	Meaning
20	Unknown recipient
25	Mail session failed to start

In addition, the following codes may be returned by the MAPI system procedures:

Return code	Meaning
11	Ambiguous recipient
12	Attachment not found
13	Disk full
14	Failure
15	Insufficient memory
16	Invalid session
17	Text too large
18	Too many files
19	Too many recipients
20	Unknown recipient
21	Login failure
22	Too many sessions
23	User abort
24	No MAPI

In addition, the following codes may be returned by the SMTP system procedures:

Return code	Meaning
100	Socket error.
101	Socket timeout.
102	Unable to resolve the SMTP server hostname.
103	Unable to connect to the SMTP server.

Return code	Meaning
104	Server error; response not understood. For example, the message is poorly formatted, or the server is not SMTP.
421	*domain* service not available, closing transmission channel.
450	Requested mail action not taken: mailbox unavailable.
451	Requested action not taken: local error in processing.
452	Requested action not taken: insufficient system storage.
500	Syntax error, command unrecognized. (This may include errors such as a command that is too long).
501	Syntax error in parameters or arguments.
502	Command not implemented.
503	Bad sequence of commands.
504	Command parameter not implemented.
550	Requested action not taken: mailbox unavailable. For example, the mailbox is not found, there is no access, or no relay is allowed.
551	User not local; please try *forward-path*
552	Request mail action aborted: exceeded storage allocation.
553	Requested action not taken: mailbox name not allowed. For example, the mailbox syntax is incorrect.
554	Transaction failed.

Adaptive Server Enterprise system and catalog procedures

Adaptive Server Enterprise provides system and catalog procedures to carry out many administrative functions and to obtain system information. System procedures are built-in stored procedures used for getting reports from and updating system tables; catalog stored procedures retrieve information from the system tables in tabular form.

SQL Anywhere has implemented support for some of these Adaptive Server Enterprise procedures. However, for information about using these procedures, refer to your Adaptive Server Enterprise documentation.

Adaptive Server Enterprise system procedures

The following list describes the Adaptive Server Enterprise system procedures that are provided in SQL Anywhere.

While these procedures perform the same functions as they do in Adaptive Server Enterprise and pre-Version 12 Adaptive Server IQ, they are not identical. If you have preexisting scripts that use these procedures, you may want to examine the procedures. To see the text of a stored procedure, you can open it in Sybase Central or, in Interactive SQL, run the following command.

```
sp_helptext 'dbo.procedure_name'
```

You may need to reset the width of your Interactive SQL output to see the full text, by choosing **Tools** » **Options** » **SQL Anywhere** » **Truncate Length**, and entering a new value.

System procedure name	Description
sp_addgroup	Adds a group to a database
sp_addlogin	Adds a new login ID to a database
sp_addmessage	Adds a user-defined message to ISYSUSERMESSAGE, for use by stored procedure PRINT and RAISERROR calls
sp_addtype	Creates a user-defined data type
sp_adduser	Adds a new user ID to a database
sp_changegroup	Changes a user's group or adds a user to a group
sp_dropgroup	Drops a group from a database
sp_droplogin	Drops a login ID from a database
sp_dropmessage	Drops a user-defined message
sp_droptype	Drops a user-defined data type
sp_dropuser	Drops a user ID from a database
sp_getmessage	Retrieves a stored message string from ISYSUSERMESSAGE, for PRINT and RAISERROR statements.
sp_helptext	Displays the text of a system procedure, trigger, or view
sp_password	Adds or changes a password for a user ID

Adaptive Server Enterprise catalog procedures

SQL Anywhere implements a subset of the Adaptive Server Enterprise catalog procedures. The implemented catalog procedures are described in the following table.

Catalog procedure name	Description
sp_column_privileges	Unsupported
sp_columns	Returns the data types of the specified columns
sp_fkeys	Returns foreign key information about the specified table
sp_pkeys	Returns primary key information about the specified table
sp_special_columns	Returns the optimal set of columns that uniquely identify a row in the specified table
sp_sproc_columns	Returns information about a stored procedure's input and return parameters
sp_statistics	Returns information about tables and their indexes
sp_stored_procedures	Returns information about one or more stored procedures
sp_tables	Returns a list of objects that can appear in a FROM clause for the specified table

Alphabetical list of system procedures

System procedures are owned by the user ID dbo. Some of these procedures are for internal system use. This section documents only those not intended solely for system and internal use. You cannot call external functions on Windows Mobile.

openxml system procedure

Generates a result set from an XML document.

Syntax

openxml(*xml-data*,
 xpath [**,** *flags* [**,** *namespaces*]] **)**
WITH (*column-name column-type* [*xpath*],... **)**

Arguments

- **xml_data** The XML on which the result set is based. This can be any string expression, such as a constant, variable, or column.
- **xpath** A string containing an XPath query. XPath allows you to specify patterns that describe the structure of the XML document you are querying. The XPath pattern included in this argument selects the nodes from the XML document. Each node that matches the XPath query in the second *xpath* argument generates one row in the table.

  Metaproperties can only be specified in WITH clause *xpath* arguments. A metaproperty is accessed within an XPath query as if it was an attribute. If a *namespaces* is not specified, then by default the prefix mp is bound to the Uniform Resource Identifier (URI) urn:ianywhere-com:sa-xpath-metaprop. If a *namespaces* is specified, this URI must be bound to mp or some other prefix to access metaproperties in the query. Metaproperty names are case sensitive. The openxml statement supports the following metaproperties:

  - **@mp:id** returns an ID for a node that is unique within the XML document. The ID for a given node in a given document may change if the database server is restarted. The value of this metaproperty increases with document order.
  - **@mp:localname** returns the local part of the node's name, or NULL if the node does not have a name.
  - **@mp:prefix** returns the prefix part of the node's name, or NULL if the node does not have a name or if the name is not prefixed.
  - **@mp:namespaceuri** returns the URI of the namespace that the node belongs to, or NULL if the node is not in a namespace.
  - **@mp:xmltext** returns a subtree of the XML document in XML form. For example, when you match an internal node, you can use this metaproperty to return an XML string, rather than the concatenated values of the descendant text nodes.
- **flags** Indicates the mapping that should be used between the XML data and the result set when an XPath query is not specified in the WITH clause. If the *flags* parameter is not specified, the default

behavior is to map attributes to columns in the result set. The *flags* parameter can have one of the following values:

Value	Description
1	XML attributes are mapped to columns in the result set (the default).
2	XML elements are mapped to columns in the result set.

- **namespace-declaration** An XML document. The in-scope namespaces for the query are taken from the root element of the document. If namespaces are specified, then you must include a *flags* argument, even if all the *xpath* arguments are specified.
- **WITH clause** Specifies the schema of the result set and how the value is found for each column in the result set. WITH clause *xpath* arguments are matched relative to the matches for the *xpath* in the second argument. If a WITH clause expression matches more than one node, then only the first node in the document order is used. If the node is not a text node, then the result is found by appending all the text node descendants. If a WITH clause expression does not match any nodes, then the column for that row is NULL.

  The openxml WITH clause syntax is similar to the syntax for selecting from a stored procedure.

  For information about selecting from a stored procedure, see "FROM clause" on page 613.
- **column-name** The name of the column in the result set.
- **column-type** The data type of the column in the result set. The data type must be compatible with the values selected from the XML document. See "SQL data types" on page 77.

Usage

The openxml system procedure parses the *xml_data* and models the result as a tree. The tree contains a separate node for each element, attribute, and text node, or other XML construct. The XPath queries supplied to the openxml system procedure are used to select nodes from the tree, and the selected nodes are then mapped to the result set.

The XML parser used by the openxml system procedure is non-validating, and does not read the external DTD subset or external parameter entities.

When there are multiple matches for a column expression, the first match in the document order (the order of the original XML document before it was parsed) is used. NULL is returned if there are no matching nodes. When an internal node is selected, the result is all the descendant text nodes of the internal node concatenated together.

Columns of type BINARY, LONG BINARY, IMAGE, and VARBINARY are assumed to be in base64-encoded format and are decoded automatically. If you generate XML using the FOR XML clause, these types are base64-encoded, and can be decoded using the openxml system procedure. See "FOR XML and binary data" [*SQL Anywhere Server - SQL Usage*].

The openxml system procedure supports a subset of the XPath syntax, as follows:

- The child, self, attribute, descendant, descendant-or-self, and parent axes are fully supported.

- Both abbreviated and unabbreviated syntax can be used for all supported features. For example, `'a'` is equivalent to `'child::a'` and `'..'` is equivalent to `'parent::node()'`.
- Name tests can use wildcards. For example, `'a/*/b'`.
- The following kind tests are supported: node(), text(), processing-instruction(), and comment().
- Qualifiers of the form *expr1*[*expr2*] and *expr1*[*expr2*="*string*"] can be used, where *expr2* is any supported XPath expression. A qualifier evaluates TRUE if *expr2* matches one or more nodes. For example, `'a[b]'` finds `a` nodes that have at least one `b` child, and `a[b="I"]` finds `a` nodes that have at least one `b` child with a text value of `I`.

See also

- "Using XPath expressions" [*SQL Anywhere Server - SQL Usage*]
- "Importing XML using openxml" [*SQL Anywhere Server - SQL Usage*]
- XPath query language: http://www.w3.org/TR/xpath.

Example

The following query generates a result set from the XML document supplied as the first argument to the openxml system procedure:

```
SELECT * FROM openxml( '<products>
                  <ProductType ID="301">Tee Shirt</ProductType>
                  <ProductType ID="401">Baseball Cap</ProductType>
                  </products>',
                  '/products/ProductType' )
WITH ( ProductName LONG VARCHAR 'text()', ProductID CHAR(3) '@ID');
```

This query generates the following result:

ProductName	ProductID
Tee Shirt	301
Baseball Cap	401

In the following example, the first <ProductType> element contains an entity. When you execute the query, this node is parsed as an element with four children: Tee, &amp;, Sweater, and Set. You can use `.` to concatenate the children together in the result set.

```
SELECT * FROM openxml( '<products>
                  <ProductType ID="301">Tee &amp; Sweater Set</ProductType>
                  <ProductType ID="401">Baseball Cap</ProductType>
                  </products>',
                  '/products/ProductType' )
WITH ( ProductName LONG VARCHAR '.', ProductID CHAR(3) '@ID');
```

This query generates the following result:

ProductName	ProductID
Tee Shirt & Sweater Set	301

ProductName	ProductID
Baseball Cap	401

The following query uses an equality predicate to generate a result set from the supplied XML document.

```
SELECT * FROM openxml('<EmployeeDirectory>
   <Employee>
      <column name="EmployeeID">105</column>
      <column name="GivenName">Matthew</column>
      <column name="Surname">Cobb</column>
      <column name="Street">7 Pleasant Street</column>
      <column name="City">Grimsby</column>
      <column name="State">UT</column>
      <column name="PostalCode">02154</column>
      <column name="Phone">6175553840</column>
   </Employee>
  <Employee>
      <column name="EmployeeID">148</column>
      <column name="GivenName">Julie</column>
      <column name="Surname">Jordan</column>
      <column name="Street">1244 Great Plain Avenue</column>
      <column name="City">Woodbridge</column>
      <column name="State">AZ</column>
      <column name="PostalCode">01890</column>
      <column name="Phone">6175557835</column>
   </Employee>
  <Employee>
      <column name="EmployeeID">160</column>
      <column name="GivenName">Robert</column>
      <column name="Surname">Breault</column>
      <column name="Street">358 Cherry Street</column>
      <column name="City">Milton</column>
      <column name="State">PA</column>
      <column name="PostalCode">02186</column>
      <column name="Phone">6175553099</column>
   </Employee>
  <Employee>
      <column name="EmployeeID">243</column>
      <column name="GivenName">Natasha</column>
      <column name="Surname">Shishov</column>
      <column name="Street">151 Milk Street</column>
      <column name="City">Grimsby</column>
      <column name="State">UT</column>
      <column name="PostalCode">02154</column>
      <column name="Phone">6175552755</column>
   </Employee>
</EmployeeDirectory>', '/EmployeeDirectory/Employee')
WITH ( EmployeeID INT 'column[@name="EmployeeID"]',
       GivenName    CHAR(20) 'column[@name="GivenName"]',
       Surname      CHAR(20) 'column[@name="Surname"]',
       PhoneNumber  CHAR(10) 'column[@name="Phone"]');
```

This query generates the following result set:

EmployeeID	GivenName	Surname	PhoneNumber
105	Matthew	Cobb	6175553840

EmployeeID	GivenName	Surname	PhoneNumber
148	Julie	Jordan	6175557835
160	Robert	Breault	6175553099
243	Natasha	Shishov	6175552755

The following query uses the XPath @attribute expression to generate a result set:

```
SELECT * FROM openxml( '<Employee
      EmployeeID="105"
      GivenName="Matthew"
      Surname="Cobb"
      Street="7 Pleasant Street"
      City="Grimsby"
      State="UT"
      PostalCode="02154"
      Phone="6175553840"
/>', '/Employee' )
WITH ( EmployeeID INT '@EmployeeID',
       GivenName    CHAR(20) '@GivenName',
       Surname      CHAR(20) '@Surname',
       PhoneNumber  CHAR(10) '@Phone');
```

The following query operates on an XML document like the one used in the above query, except that an XML namespace has been introduced. It demonstrates the use of wildcards in the name test for the XPath query, and generates the same result set as the above query.

```
SELECT * FROM openxml( '<Employee  xmlns="http://www.iAnywhere.com/
EmployeeDemo"
      EmployeeID="105"
      GivenName="Matthew"
      Surname="Cobb"
      Street="7 Pleasant Street"
      City="Grimsby"
      State="UT"
      PostalCode="02154"
      Phone="6175553840"
/>', '/*:Employee' )

WITH ( EmployeeID INT '@EmployeeID',
       GivenName    CHAR(20) '@GivenName',
       Surname      CHAR(20) '@Surname',
       PhoneNumber  CHAR(10) '@Phone');
```

Alternatively, you could specify a namespace declaration:

```
SELECT * FROM openxml( '<Employee  xmlns="http://www.iAnywhere.com/
EmployeeDemo"
      EmployeeID="105"
      GivenName="Matthew"
      Surname="Cobb"
      Street="7 Pleasant Street"
      City="Grimsby"
      State="UT"
      PostalCode="02154"
      Phone="6175553840"
/>', '/prefix:Employee', 1, '<r xmlns:prefix="http://www.iAnywhere.com/
EmployeeDemo"/>' )
```

```
WITH ( EmployeeID INT '@EmployeeID',
       GivenName    CHAR(20) '@GivenName',
       Surname      CHAR(20) '@Surname',
       PhoneNumber  CHAR(10) '@Phone');
```

For more examples of using the openxml system procedure, see "Importing XML using openxml" [*SQL Anywhere Server - SQL Usage*].

sa_ansi_standard_packages system procedure

Returns information about the non-core SQL extensions used in a SQL statement.

Syntax

sa_ansi_standard_packages(*sql-standard-string*, *sql-statement-string* **)**

Arguments

- **sql-standard-string** The standard to use for the core extensions. One of SQL:1999 or SQL:2003.
- **sql-statement-string** The SQL statement to evaluate.

Remarks

If there are no non-core extensions used for the statement, the result set is empty.

Permissions

None

Side effects

None

See also

- "SQL preprocessor" [*SQL Anywhere Server - Programming*]
- "SQLFLAGGER function [Miscellaneous]" on page 310
- "sql_flagger_error_level option [compatibility]" [*SQL Anywhere Server - Database Administration*]
- "sql_flagger_warning_level option [compatibility]" [*SQL Anywhere Server - Database Administration*]

Example

Following is an example call to the sa_ansi_standard_packages system procedure:

```
CALL sa_ansi_standard_packages( 'SQL:2003',
'SELECT *
   FROM ( SELECT o.SalesRepresentative,
                 o.Region,
                 SUM( s.Quantity * p.UnitPrice ) AS total_sales,
                 DENSE_RANK() OVER ( PARTITION BY o.Region,
                                     GROUPING( o.SalesRepresentative )
                                     ORDER BY total_sales DESC ) AS
sales_rank
            FROM Product p, SalesOrderItems s, SalesOrders o
            WHERE p.ID = s.ProductID AND s.ID = o.ID
            GROUP BY GROUPING SETS( ( o.SalesRepresentative, o.Region ),
    o.Region ) ) AS DT
       WHERE sales_rank <= 3
       ORDER BY Region, sales_rank');
```

The query generates the following result set:

package_id	package_name
T612	Advanced OLAP operations
T611	Elementary OLAP operations
F591	Derived tables
T431	Extended grouping capabilities

sa_audit_string system procedure

Adds a string to the transaction log.

Syntax

sa_audit_string(*string* **)**

Arguments

- **string**   A string of characters to add to the transaction log.

Remarks

If auditing is turned on, this system procedure adds a comment to the auditing information stored in the transaction log. The string can be a maximum of 200 bytes.

Permissions

DBA authority required

Side effects

None

See also

- "auditing option [database]" [*SQL Anywhere Server - Database Administration*]
- "Auditing database activity" [*SQL Anywhere Server - Database Administration*]

Example

The following example uses sa_audit_string to add a comment into the audit log:

```
CALL sa_audit_string( 'Auditing test' );
```

sa_char_terms system procedure

Breaks a CHAR string into terms and returns each term as a row along with its position.

Syntax

sa_char_terms('*char-string*' [**,** '*text-config-name*' [**,** '*owner*']]]
)

Arguments

- **char-string** The CHAR string you are parsing.
- **text-config-name** The text configuration object to apply when processing the string. The default value is 'default_char'.
- **owner** The owner of the specified text configuration object. The default value is DBA.

Remarks

You can use this system procedure to find out how a string is interpreted when the settings for a text configuration object are applied. This can be helpful when you want to know what terms would be dropped during indexing or from a query string.

Permissions

None

Side effects

None

See also

- "Full text searching" [*SQL Anywhere Server - SQL Usage*]
- "Text configuration objects" [*SQL Anywhere Server - SQL Usage*]
- "sa_nchar_terms system procedure" on page 914

Example

The following statement returns the terms in the CHAR string 'the quick brown fox jumped over the fence':

```
CALL sa_char_terms ( 'the quick brown fox jumped over the fence' );
```

term	position
the	1
quick	2
brown	3
fox	4
jumped	5

term	position
over	6
the	7
fence	8

sa_check_commit system procedure

Checks for outstanding referential integrity violations before a commit.

Syntax

sa_check_commit(
tname,
keyname
)

Arguments

- **tname** A VARCHAR(128) parameter containing the name of a table with a row that is currently violating referential integrity.
- **keyname** A VARCHAR(128) parameter containing the name of the corresponding foreign key index.

Remarks

If the database option wait_for_commit is On, or if a foreign key is defined using CHECK ON COMMIT in the CREATE TABLE statement, you can update the database and cause a referential integrity violation if the violations are resolved before the changes are committed.

You can use the sa_check_commit system procedure to check whether there are any outstanding referential integrity violations before attempting to commit your changes.

The returned parameters indicate the name of a table containing a row that is currently violating referential integrity, and the name of the corresponding foreign key index.

Permissions

None

Side effects

None

See also

- "wait_for_commit option [database]" [*SQL Anywhere Server - Database Administration*]
- "CREATE TABLE statement" on page 523

Example

The following set of commands can be executed from Interactive SQL. Rows are deleted from the Departments table in the sample database and a referential integrity violation occurs. The call to the sa_check_commit system procedure checks which tables and keys have outstanding violations, and the rollback cancels the change:

```
SET TEMPORARY OPTION wait_for_commit='On'
go
DELETE FROM Departments
go
CREATE VARIABLE tname VARCHAR( 128 );
CREATE VARIABLE keyname VARCHAR( 128 )
go
CALL sa_check_commit( tname, keyname )
go
SELECT tname, keyname
go
ROLLBACK
go
```

sa_clean_database system procedure

Starts the database cleaner and sets the maximum length of time for which it can run.

Syntax

sa_clean_database([*duration*] **)**

Arguments

- **duration** The number of seconds that the clean operation is allowed to run. If no argument is specified, or if 0 is specified, the database cleaner runs until all pages in all dbspaces have been cleaned.

Remarks

The database cleaner is an internal task that runs on a default schedule. You can use this system procedure to force the database cleaner to run immediately and to specify how long the cleaner can run each time it is invoked.

Some database tasks, such as processing snapshot isolation transactions, index maintenance, and deleting rows, can execute more efficiently if some portions of the request are deferred to a later time. These deferrable activities typically involve cleanup by removing deleted, historical, and otherwise unnecessary entries from database pages, or reorganizing database pages for more efficient access.

Postponing some of these activities not only allows the current request to finish more quickly, it potentially allows cleanup to occur when the database server is less active. These unnecessary entries are identified so that they are not visible to other transactions; however, they do take up space on a page, and must be removed at some point.

The database cleaner performs any deferred cleanup activities. It is scheduled to run every 20 seconds. When it is invoked, the database cycles sequentially through the database's dbspaces, examining and cleaning each cleanable page before moving on to the next one. When invoked automatically by the database server, the database cleaner is a self-tuning process. The amount of work that the database cleaner performs, and the duration for which it executes, depend on several factors, including the fraction of outstanding cleanable pages in a dbspace, the current amount of activity in the database server, and the amount of time that the database cleaner has already spent cleaning. If, after running for 0.5 seconds, the cleaner detects active requests in the server, it stops and reschedules itself to execute at its regular interval. The database cleaner attempts to process pages when there are no other requests executing in the server, and therefore takes advantage of periods of server inactivity.

Database cleaner statistics are available through four database properties:

- **CleanablePagesAdded** returns the number of pages that need to be cleaned
- **CleanablePagesCleaned** returns the number of pages that have already been cleaned
- **CleanableRowsAdded** returns the number of rows that need to be cleaned
- **CleanabledRowsCleaned** returns the number of rows that have already been cleaned

The difference between the values of CleanablePagesAdded and CleanablePagesCleaned indicates how many database pages still require cleaning.

You can use the sa_clean_database system procedure to configure the database cleaner to run until all the pages in a database are cleaned, or to specify a maximum duration for the database cleaner to run.

To further customize the behavior of the database cleaner, you can set up an event that starts the database cleaner if the number of pages or rows that need to be cleaned exceed a specified threshold. See "CREATE EVENT statement" on page 450.

Permissions

DBA authority required

Side effects

None

See also

- "CREATE EVENT statement" on page 450
- CleanablePagesAdded, CleanablePagesCleaned, CleanableRowsAdded, and CleanableRowsCleaned properties: "Database properties" [*SQL Anywhere Server - Database Administration*]

Example

The following example sets the duration of the database cleaner to 10 seconds:

```
CALL sa_clean_database( 10 );
```

The following example creates a scheduled event that runs daily to allow the database cleaner to run until all pages in the database are cleaned:

```
CREATE EVENT DailyDatabaseCleanup
SCHEDULE
   START TIME '6:00 pm'
   ON ( 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday' )
   HANDLER
      BEGIN
      CALL sa_clean_database( );
   END;
```

The following example forces the database cleaner to run when 20% or more of the pages in the database need to be cleaned:

```
CREATE EVENT PERIODIC_CLEANER
SCHEDULE
BETWEEN '9:00 am' and '5:00 pm'
EVERY 1 HOURS
HANDLER
BEGIN
     DECLARE @num_db_pages INTEGER;
     DECLARE @num_dirty_pages INTEGER;

     -- Get the number of database pages
     SELECT (SUM( DB_EXTENDED_PROPERTY( 'FileSize', t.dbspace_id ) -
                  DB_EXTENDED_PROPERTY( 'FreePages', t.dbspace_id ) ))
     INTO @num_db_pages
     FROM (SELECT dbspace_id FROM SYSDBSPACE) AS t;

     -- Get the number of dirty pages to be cleaned
     SELECT (DB_PROPERTY( 'CleanablePagesAdded' ) -
                  DB_PROPERTY( 'CleanablePagesCleaned' ))
     INTO @num_dirty_pages;

     -- Check whether the number of dirty pages exceeds 20% of
     -- the size of the database
     IF @num_dirty_pages > @num_db_pages * 0.20 THEN
       -- Start cleaning the database for a maximum of 60 seconds
       CALL sa_clean_database( 60 );
     END IF;
END;
```

sa_column_stats system procedure

Returns various statistics about the specified column(s). These statistics are not related to the column statistics maintained for use by the optimizer.

Syntax

sa_column_stats (
[*tab_name*]
[**,** *col_name*]
[**,** *tab_owner*]
[**,** *max_rows*]
)

Arguments

- **tab_name** This optional CHAR(128) parameter specifies the name of the table. If this parameter is not specified, statistics are calculated for all columns in all table(s).
- **col_name** This optional CHAR(128) parameter specifies the columns for which to calculate statistics. If this parameter is not specified, statistics are calculated for all columns in the specified table(s).

- **tab_owner** This optional CHAR(128) parameter specifies the owner of the table. If this parameter is not specified, the database server uses the owner of the first table that matches the *tab_name* specified.
- **max_rows** This optional INTEGER parameter specifies the number of rows to use for the calculations. If this parameter is not specified, 1000 rows are used by default. Specifying 0 instructs the database server to calculate the ratio based on all the rows in the table.

Result set

With the exception of table_owner, table_name, and column_name, all values in the result set are NULL for non-string columns. Also, for empty tables, num_rows_processed and num_values_compressed are 0, while all other values are NULL.

Column name	Data type	Description
table_owner	CHAR(128)	The owner of the table.
table_name	CHAR(128)	The table name.
column_name	CHAR(128)	The column name.
num_rows_processed	INTEGER	The total number of rows read to calculate the statistics.
num_values_compressed	INTEGER	The number of values in the column that are compressed. If the column is not compressed, the value is 0.
avg_compression_ratio	DOUBLE	The average compression ratio, expressed as a percentage reduction in size, for compressed values in the column. If the column is not compressed, the value is NULL.
avg_length	DOUBLE	The average length of all non-NULL strings in the column.
stddev_length	DOUBLE	The standard deviation of the lengths of all non-NULL strings in the column.
min_length	INTEGER	The minimum length of non-NULL strings in the column.
max_length	INTEGER	The maximum length of strings in the column.
avg_uncompressed_length	DOUBLE	The average length of all uncompressed, non-NULL strings in the column.
stddev_uncompressed_length	DOUBLE	The standard deviation of the lengths of all uncompressed, non-NULL strings in the column.
min_uncompressed_length	INTEGER	The minimum length of all uncompressed, non-NULL strings in the column.

Column name	Data type	Description
max_uncompressed_length	INTEGER	The maximum length of all uncompressed, non-NULL strings in the column.

Remarks

The database server determines the columns that match the owner, table, and column names specified, and then for each one, calculates statistics for the data in each specified column. By default, the database server only uses the first 1000 rows of data.

For avg_compression_ratio, values cannot be greater than, or equal to 100, however, they can be less than 0 if highly incompressible data (for example, data that is already compressed) is inserted into a compressed column. Higher values indicate better compression. For example, if the number returned is 80, then the size of the compressed data is 80% less than the size of the uncompressed data.

Permissions

DBA authority required

Side effects

None

See also

- "Choosing whether to compress columns" [*SQL Anywhere Server - SQL Usage*]

Example

In this example, you use the sa_column_stats system procedure in a SELECT statement to determine which columns in the database are benefitting most from column compression:

```
SELECT * FROM sa_column_stats()
  WHERE num_values_compressed > 0
  ORDER BY avg_compression_ratio desc;
```

In this example, you narrow your selection from the previous example to tables owned by bsmith:

```
SELECT * FROM sa_column_stats( tab_owner='bsmith' )
  WHERE num_values_compressed > 0
  ORDER BY avg_compression_ratio desc;
```

sa_conn_activity system procedure

Returns the most recently-prepared SQL statement for each connection to the specified database on the server.

Syntax

sa_conn_activity([*connidparm*] **)**

Arguments

- **connidparm** Use this optional INTEGER parameter to specify the ID number of a connection.

Result set

Column name	Data type	Description
Number	INT	The ID number of the connection.
Name	VARCHAR(255)	The name of the connection.
Userid	VARCHAR(255)	The user ID for the connection.
DBNumber	INT	The ID number of the database.
LastReqTime	VARCHAR(255)	The time at which the last request for the specified connection started.
LastStatement	LONG VARCHAR	The most recently-prepared SQL statement for the connection.

Remarks

The sa_conn_activity system procedure returns a result set consisting of the most recently-prepared SQL statement for each connection, if the server has been told to collect the information. Recording of statements must be enabled for the database server prior to calling sa_conn_activity. To do this, specify the -zl option when starting the database server, or execute the following:

```
CALL sa_server_option('RememberLastStatement','ON');
```

This procedure is useful when the database server is busy and you want to obtain information about the last SQL statement prepared for each connection. This feature can be used as an alternative to request logging.

For information about the LastStatement property, from which these values are derived, see "Connection properties" [*SQL Anywhere Server - Database Administration*].

If *connidparm* is not specified, then information is returned for all connections to all databases running on the database server. If *connidparm* is less than zero, option values for the current connection are returned.

Permissions

None

Side effects

None

See also

- "-zl server option" [*SQL Anywhere Server - Database Administration*]
- "sa_server_option system procedure" on page 935

sa_conn_compression_info system procedure

Summarizes communication compression rates.

Syntax

sa_conn_compression_info([*connidparm*] **)**

Arguments

- **connidparm** Use this optional INTEGER parameter to specify the ID number of a connection.

Result set

Column name	Data type	Description
Type	VARCHAR(20)	A string identifying whether the compression statistics that follow represent either one Connection, or all connections to the Server.
ConnNumber	INTEGER	An INTEGER representing a connection ID. Returns NULL if the Type is Server.
Compression	VARCHAR(10)	A string representing whether compression is enabled for the connection. Returns NULL if Type is Server, or ON/OFF if Type is Connection.
TotalBytes	INTEGER	An INTEGER representing the total number of actual bytes both sent and received.
TotalBytesUnComp	INTEGER	An INTEGER representing the number of bytes that would have been sent and received if compression was disabled.
CompRate	NUMERIC(5,2)	A NUMERIC (5,2) representing the overall compression rate. For example, a value of 0 indicates that no compression occurred. A value of 75 indicates that the data was compressed by 75%, or down to one quarter of its original size.
CompRateSent	NUMERIC(5,2)	A NUMERIC (5,2) representing the compression rate for data sent to the client.
CompRateReceived	NUMERIC(5,2)	A NUMERIC (5,2) representing the compression rate for data received from the client.
TotalPackets	INTEGER	An INTEGER representing the total number of actual packets both sent and received.
TotalPacketsUnComp	INTEGER	An INTEGER representing the total number of packets that would have been sent and received if compression was disabled.

Column name	Data type	Description
CompPktRate	NUMERIC(5,2)	A NUMERIC (5,2) representing the overall compression rate of packets.
CompPktRateSent	NUMERIC(5,2)	A NUMERIC (5,2) representing the compression rate of packets sent to the client.
CompPktRateReceived	NUMERIC(5,2)	A NUMERIC (5,2) representing the compression rate of packets received from the client.

Remarks

If you specify the connection ID number, the sa_conn_compression_info system procedure returns a result set consisting of compression properties for the supplied connection. If no *connection-id* is supplied, this system procedure returns information for all current connections to databases on the server.

For information about the properties these values are derived from, see "Connection properties" [*SQL Anywhere Server - Database Administration*].

Permissions

None

Side effects

None

Example

The following example uses the sa_conn_compression_info system procedure to return a result set summarizing compression properties for all connections to the server.

```
CALL sa_conn_compression_info( );
```

Type	ConnNumber	Compression	TotalBytes	...
Connection	79	Off	7841	...
Server	(NULL)	(NULL)	2737761	...
...	...	...	...	...

sa_conn_info system procedure

Reports connection property information.

Syntax

sa_conn_info([*connidparm*] **)**

Arguments

- **connidparm** This optional INTEGER parameter specifies the ID number of a connection.

Result set

Column name	Data type	Description
Number	INTEGER	The ID number of the connection.
Name	VARCHAR(255)	The name of the connection.
Userid	VARCHAR(255)	The user ID for the connection.
DBNumber	INTEGER	The ID number of the database.
LastReqTime	VARCHAR(255)	The time at which the last request for the specified connection started.
ReqType	VARCHAR(255)	A string for the type of the last request.
CommLink	VARCHAR(255)	The communication link for the connection. This is one of the network protocols supported by SQL Anywhere, or local for a same-computer connection.
NodeAddr	VARCHAR(255)	The address of the client in a client/server connection.
ClientPort	INTEGER	The port number on which the client application communicates using TCP/IP.
ServerPort	INTEGER	The port number on which the server communicates using TCP/IP.
BlockedOn	INTEGER	If the current connection is not blocked, this is zero. If it is blocked, the connection number on which the connection is blocked because of a locking conflict.
LockTable	VARCHAR(255)	If the connection is currently waiting for a lock, LockTable will be the name of the table associated with that lock. Otherwise, LockTable will be the empty string.
UncommitOps	INTEGER	The number of uncommitted operations.
LockRowID	UNSIGNED BIGINT	If the connection is waiting on a lock that is associated with a particular row identifier, LockRowID contains that row identifier. LockRowID is NULL if the connection is not waiting on a lock associated with a row (that is, it is not waiting on a lock, or it is waiting on a lock that has no associated row).

Column name	Data type	Description
LockIndexID	INTEGER	If the connection is waiting on a lock that is associated with a particular index, LockIndexID contains the identifier of that index (or -1 if the lock is associated with all indexes on the table in LockTable). LockIndexID is NULL if the connection is not waiting on a lock associated with an index (that is, it is not waiting on a lock, or it is waiting on a lock that has no associated index).

Remarks

If you specify the connection ID number, the sa_conn_info system procedure returns a result set consisting of connection properties for the supplied connection. If no *connidparm* is supplied, this system procedure returns information for all current connections to databases on the server. If *connidparm* is less than zero, option values for the current connection are returned.

In a block situation, the BlockedOn value returned by this procedure allows you to check which users are blocked, and who they are blocked on. The sa_locks system procedure can be used to display the locks held by the blocking connection.

For more information based on any of these properties, you can execute something similar to the following:

```
SELECT *, DB_NAME( DBNumber ),
   CONNECTION_PROPERTY( 'LastStatement', Number )
   FROM sa_conn_info( );
```

The value of LockRowID can be used to look up a lock in the output of the sa_locks procedure.

The value in LockIndexID can be used to look up a lock in the output of the sa_locks procedure. Also, the value in LockIndexID corresponds to the primary key of the ISYSIDX system table, which can be viewed using the SYSIDX system view.

Every lock has an associated table, so the value of LockTable can be used to unambiguously determine whether a connection is waiting on a lock.

Permissions

None

Side effects

None

See also

- “Connection properties” [*SQL Anywhere Server - Database Administration*]
- “sa_locks system procedure” on page 893
- “SYSIDX system view” on page 1005

Example

The following example uses the sa_conn_info system procedure to return a result set summarizing connection properties for all connections to the server.

```
CALL sa_conn_info( );
```

Number	Name	Userid	DBNumber	...
79		DBA	0	...
46	Sybase Central 1	DBA	0	...
...	...	...	...	...

sa_conn_list system procedure

Returns a result set containing connection IDs.

Syntax

sa_conn_list(
[*connidparm*]
[**,** *dbidparm*]
)

Arguments

- **connidparm** Use this optional INTEGER parameter to specify the ID number of a connection.
- **dbidparm** Use this optional INTEGER parameter to specify the ID number of a database.

Result set

Column name	Data type	Description
Number	INTEGER	The ID number of the connection.

Remarks

If you do not specify any parameters, or if both parameters are NULL, the connection IDs for all connections to all databases running on the database server are returned. If *connidparm* is less than 0, only the connection ID for the current connection is returned. If *connidparm* is NULL and *dbidparm* is less than 0, the connection IDs for just the current database are returned. If *connidparm* is NULL, and *dbidparm* is not NULL and its value is greater than or equal to 0, the connection IDs for only that database are returned.

Permissions

None

Side effects

None

See also

- "sa_db_list system procedure" on page 861
- "sa_conn_options system procedure" on page 856

sa_conn_options system procedure

Returns property information for connection properties that correspond to database options.

Syntax

sa_conn_options([*connidparm*] **)**

Arguments

- **connidparm** Use this optional INTEGER parameter to specify the ID number of a connection.

Result set

Column name	Data type	Description
Number	INTEGER	The ID number of the connection.
PropNum	INTEGER	The connection property number.
OptionName	VARCHAR(255)	The option name.
OptionDescription	VARCHAR(255)	The option description.
Value	LONG VARCHAR	The option value.

Remarks

Returns the connection ID as Number, and the PropNum, OptionName, OptionDescription, and Value for each available connection property that corresponds to a database option.

If you do not specify *connidparm*, then option values for all connections to the current database are returned. If *connidparm* is less than zero, option values for the current connection are returned.

Permissions

None

Side effects

None

See also

- "sa_db_list system procedure" on page 861
- "sa_conn_list system procedure" on page 855

sa_conn_properties system procedure

Reports connection property information.

Syntax

sa_conn_properties([*connidparm*] **)**

Arguments

- **connidparm** Use this optional INTEGER parameter to specify the ID number of a connection.

Result set

Column name	Data type	Description
Number	INTEGER	The ID number of the connection.
PropNum	INTEGER	The connection property number.
PropName	VARCHAR(255)	The connection property name.
PropDescription	VARCHAR(255)	The connection property description.
Value	LONG VARCHAR	The connection property value.

Remarks

Returns the connection ID as Number, and the PropNum, PropName, PropDescription, and Value for each available connection property. Values are returned for all connection properties, database option settings related to connections, and statistics related to connections. Valid properties with NULL values are also returned.

If no *connidparm* is supplied, properties for all connections to the current database are returned. If *connidparm* is less than zero, option values for the current connection are returned.

Permissions

None

Side effects

None

See also

- "sa_conn_list system procedure" on page 855
- "sa_conn_options system procedure" on page 856
- "System functions" on page 133
- "Connection properties" [*SQL Anywhere Server - Database Administration*]

Examples

The following example uses the sa_conn_properties system procedure to return a result set summarizing connection property information for all connections.

```
CALL sa_conn_properties( );
```

Number	PropNum	PropName	...
79	37	CacheHits	...
79	38	CacheRead	...
...	...	...	...

This example uses the sa_conn_properties system procedure to return a list of all connections, in decreasing order by CPU time*:

```
SELECT Number AS connection_number,
    CONNECTION_PROPERTY ( 'Name', Number ) AS connection_name,
    CONNECTION_PROPERTY ( 'Userid', Number ) AS user_id,
  CAST ( Value AS NUMERIC ( 30, 2 ) ) AS approx_cpu_time
  FROM sa_conn_properties()
  WHERE PropName = 'ApproximateCPUTime'
  ORDER BY approx_cpu_time DESC;
```

*Example courtesy of Breck Carter, RisingRoad Professional Services (http://www.risingroad.com).

sa_convert_ml_progress_to_timestamp system procedure

For MobiLink scripted uploads only. This converts the progress value for scripted upload from a 64-bit INTEGER to a TIMESTAMP.

Syntax

sa_convert_ml_progress_to_timestamp(*progress* **)**

Arguments

- **progress** The function takes one parameter which is an UNSIGNED BIGINT.

Remarks

The function returns the TIMESTAMP that is represented by the value passed in. This procedure is the inverse of sa_convert_timestamp_to_ml_progress.

Permissions

None

Side effects

None

See also

- “sa_convert_timestamp_to_ml_progress system procedure” on page 859
- “Scripted upload” [*MobiLink - Client Administration*]

Example

```
SELECT sa_convert_timestamp_to_ml_progress( 3600000 );
```

sa_convert_timestamp_to_ml_progress system procedure

For MobiLink scripted uploads only. This converts the progress value for scripted upload from a TIMESTAMP to a 64-bit UNISIGNED BIGINT.

Syntax

sa_convert_timestamp_to_ml_progress([*t1*] **)**

Arguments

- **t1** Use this optional TIMESTAMP parameter to specify the progress value to convert to 64-bit UNISIGNED BIGINT.

Remarks

The function returns an UNISIGNED BIGINT that represents the timestamp passed in as a parameter. This procedure is the inverse of sa_convert_ml_progress_to_timestamp.

Permissions

None

Side effects

None

See also

- “sa_convert_ml_progress_to_timestamp system procedure” on page 858
- “Scripted upload” [*MobiLink - Client Administration*]

Examples

```
SELECT sa_convert_timestamp_to_ml_progress( CURRENT TIMESTAMP );
```

```
SELECT sa_convert_timestamp_to_ml_progress( '1900/01/01 1:00' );
```

sa_db_info system procedure

Reports database property information.

Syntax

sa_db_info([*dbidparm*] **)**

Arguments

- **dbidparm** Use this optional INTEGER parameter to specify the ID number of a database.

Result set

Column name	Data type	Description
Number	INTEGER	The ID number of the connection.
Alias	VARCHAR(255)	The database name.
File	VARCHAR(255)	The file name of the database root file, including path.
ConnCount	INTEGER	The number of connections to the database.
PageSize	INTEGER	The page size of the database, in bytes.
LogName	VARCHAR(255)	The file name of the transaction log, including path.

Remarks

If you specify a database ID, sa_db_info returns a single row containing the Number, Alias, File, ConnCount, PageSize, and LogName for the specified database.

If no *dbidparm* is supplied, properties for all databases are returned.

Permissions

None

Side effects

None

See also

- "sa_db_properties system procedure" on page 861
- "Database properties" [*SQL Anywhere Server - Database Administration*]

Example

The following statement returns a row for each database that is running on the server:

```
CALL sa_db_info( );
```

Property	Value
Number	0
Alias	demo
File	C:\Documents and Settings\All Users\Documents\SQL Anywhere 11\Samples\demo.db

Property	Value
ConnCount	1
PageSize	4096
LogName	C:\Documents and Settings\All Users\Documents\SQL Anywhere 11\Samples\demo.log

sa_db_list system procedure

Returns a database ID.

Syntax

sa_db_list([*dbidparm*] **)**

Arguments

- **dbidparm** Use this optional INTEGER parameter to specify the ID number of a database.

Result set

Column name	Data type	Description
Number	INTEGER	The ID number of the database.

Remarks

If you do not specify a *dbidparm*, or if *dbidparm* is NULL, the IDs for all databases running on the database server are returned. If *dbidparm* is less than 0, then only the ID for the current database is returned.

Permissions

None

Side effects

None

See also

- "sa_conn_list system procedure" on page 855
- "sa_conn_options system procedure" on page 856

sa_db_properties system procedure

Reports database property information.

Syntax

sa_db_properties([*dbidparm*] **)**

Arguments

- **dbidparm** Use this optional INTEGER parameter to specify the ID number of a database.

Result set

Column name	Data type	Description
Number	INTEGER	The ID number of the database.
PropNum	INTEGER	The database property number.
PropName	VARCHAR(255)	The database property name.
PropDescription	VARCHAR(255)	The database property description.
Value	LONG VARCHAR	The database property value.

Remarks

If you specify a database ID, the sa_db_properties system procedure returns the database ID number and the PropNum, PropName, PropDescription, and Value for each available database property. Values are returned for all database properties and statistics related to databases. Valid properties with NULL values are also returned.

If *dbidparm* is not specified, properties for all databases are returned.

Permissions

None

Side effects

None

See also

- "sa_db_info system procedure" on page 859
- "Database properties" [*SQL Anywhere Server - Database Administration*]

Example

The following example uses the sa_db_properties system procedure to return a result set summarizing database property information for all databases.

```
CALL sa_db_properties( );
```

Number	PropNum	PropName	...
0	0	ConnCount	...

Number	PropNum	PropName	...
0	1	IdleCheck	...
0	2	IdleWrite	...
...	...	...	...

sa_dependent_views system procedure

Returns the list of all dependent views for a given table or view.

Syntax

sa_dependent_views('*tbl_name* **' [, '** *owner_name* **')**

Arguments

- **tbl_name** Use this CHARACTER parameter to specify the name of the table or view.
- **owner_name** Use this optional CHARACTER parameter to specify the owner for *tbl_name*.

Result set

Column name	Data type	Description
table_id	UNSIGNED INTEGER	The object ID of the table or view.
dep_view_id	UNSIGNED INTEGER	The object ID of the dependent views.

Remarks

Use this procedure to obtain the list of IDs of dependent views. Alternatively, you can use the procedure in a statement that returns more information about the views, such as their names.

No errors are generated if no existing tables satisfy the specified criteria for table and owner names. Also:

- *table_name* is optional and has a default value of null.
- If both *owner* and *table_name* are null, information is returned on all tables that have dependent views.
- If *table_name* is null but *owner* is specified, information is returned on all tables owned by the specified owner.
- If *table_name* is specified but *owner* is null, information is returned on any one of the tables with the specified name.

By default, execution of the procedure does not require any permissions and assumes that PUBLIC has access to the catalog. DBAs can control access as needed on the view and/or the catalog.

Permissions

None

Side effects

None

See also

- “SYSDEPENDENCY system view” on page 997
- “View dependencies” [*SQL Anywhere Server - SQL Usage*]

Examples

In this example, the sa_dependent_views system procedure is used to obtain the list of IDs for the views that are dependent on the SalesOrders table. The procedure returns the table_id for SalesOrders, and the dep_view_id for the dependent view, ViewSalesOrders.

```
sa_dependent_views( 'SalesOrders' );
```

In this example, the sa_dependent_views system procedure is used in a SELECT statement to obtain the list of names of views dependent on the SalesOrders table. The procedure returns the ViewSalesOrders view.

```
SELECT t.table_name FROM SYSTAB t,
sa_dependent_views( 'SalesOrders' ) v
WHERE t.table_id = v.dep_view_id;
```

sa_describe_query system procedure

Describes the result set for a query with one row describing each output column of the query.

Syntax

sa_describe_query(
query
[, *add_keys*]
)

Arguments

- **query** Use this LONG VARCHAR parameter to specify the text of the SQL statement being described.
- **add_keys** Use this optional BIT parameter to specify whether to determine a set of columns that uniquely identify rows in the result set for the query being described. The default is 0; the database server does not attempt to identify the columns. See the Remarks section below for a full explanation of this parameter.

Result Set

Column name	Data type	Description
column_number	INTEGER	The ordinal position of the column described by this row, starting at 1.

Column name	Data type	Description
name	VAR-CHAR(128)	The name of the column.
domain_id	SMALLINT	The data type of the column. See "SYSDOMAIN system view" on page 997.
domain_name	VAR-CHAR(128)	The data type name. See "SYSDOMAIN system view" on page 997.
domain_name_with_size	VAR-CHAR(160)	The data type name, including size and precision (as used in CREATE TABLE or CAST functions).
width	INTEGER	The length of a string parameter, the precision of a numeric parameter, or the number of bytes of storage for any other data type.
scale	INTEGER	The number of digits after the decimal point for numeric data type columns, and zero for all other data types.
declared_width	INTEGER	The length of a string parameter, the precision of a numeric parameter, or the number of bytes of storage for any other data type.
user_type_id	SMALLINT	The type_id of the user-defined data type if there is one, otherwise NULL. See "SYSUSERTYPE system view" on page 1041.
user_type_name	VAR-CHAR(128)	The name of the user-defined data type if there is one, otherwise NULL. See "SYSUSERTYPE system view" on page 1041.
correlation_name	VAR-CHAR(128)	The correlation name associated with the expression if one is available, otherwise NULL.
base_table_id	UNSIGNED INTEGER	The table_id if the expression is a field, otherwise NULL. See "SYSTAB system view" on page 1028.
base_column_id	UNSIGNED INTEGER	The column_id if the expression is a field, otherwise NULL. See "SYSTABCOL system view" on page 1030.
base_owner_name	VAR-CHAR(128)	The owner name if the expression is a field, otherwise NULL. See "SYSUSER system view" on page 1039.
base_table_name	VAR-CHAR(128)	The table name if the expression is a field, otherwise NULL.

Column name	Data type	Description
base_column_name	VAR-CHAR(128)	The column name if the expression is a field, otherwise NULL.
nulls_allowed	BIT	An indicator that is 1 if the expression can be NULL, otherwise 0.
is_autoincrement	BIT	An indicator that is 1 if the expression is a column declared to be autoincrement, otherwise 0.
is_key_column	BIT	An indicator that is 1 if the expression is part of a key for the result set, otherwise 0. See the Remarks section below for more information.
is_added_key_column	BIT	An indicator that is 1 if the expression is an added key column, otherwise 0. See the Remarks section below for more information.

Remarks

The sa_describe_query procedure provides an API-independent mechanism to describe the name and type information for the expressions in the result set of a query.

When 1 is specified for *add_keys*, the sa_describe_query procedure attempts to find a set of columns from the objects being queried that, when combined, can be used as a key to uniquely identify rows in result set of the query being described. The key takes the form of one or more columns from the objects being queried, and may include columns that are not explicitly referenced in the query. If the optimizer finds a key, the column or columns used in the key are identified in the results by an is_key_column value of 1. If no key is found, an error is returned.

For any column that is included in the key but that is not explicitly referenced in the query, the is_added_key_column value is set to 1 to indicate that the column has been added to the results for the procedure; otherwise, the value of is_added_key_column is 0.

If you do not specify *add_keys*, or you specify a value of 0, the optimizer does not attempt to find a key for the result set, and the is_key_column and is_added_key_column columns contain NULL.

The declared_width and width values both describe the size of a column. The declared_width describes the size of the column as defined by the CREATE TABLE statement or by the query, while the width value gives the size of the field when fetched to the client. The client representation of a type may be different from the database server. For example, date and time types are converted to strings if the return_date_time_as_string option is on. For strings, fields declared with character-length semantics have a declared_width value that matches the CREATE TABLE size, while the width value gives the maximum number of bytes needed to store the returned string. For example:

Declaration	width	declared_width
CHAR(10)	10	10

Declaration	width	declared_width
CHAR(10 CHAR)	40	10
TIMESTAMP	depends on the length of the timestamp format string	8
NUMERIC(10, 3)	10 (precision)	10 (precision)

Permissions

None

Side effects

None

See also

- "EXPRTYPE function [Miscellaneous]" on page 205
- "Character data types" on page 78
- "return_date_time_as_string option [database]" [*SQL Anywhere Server - Database Administration*]

Examples

The following example describes the information returned when querying all columns in the Departments table:

```
SELECT *
FROM sa_describe_query( 'SELECT * FROM Departments DEPT' );
```

The results show the values of the is_key_column and is_added_key_column as NULL because the *add_keys* parameter was not specified.

The following example describes the information returned by querying the DepartmentName and Surname columns of the Employees table, joined with the Departments table:

```
SELECT  *
FROM  sa_describe_query( 'SELECT DepartmentName, Surname
 FROM Employees E JOIN Departments D ON E.EmployeeID = D.DepartmentHeadId',
 add_keys = 1 );
```

The results shows a 1 in rows 3 and 4 of the result set, indicating that the columns needed to uniquely identify rows in the result set for the query are Employees.EmployeeID and Departments.DepartmentID. Also, a 1 is present in the is_added_key_column for rows 3 and 4 because Employees.EmployeeID and Departments.DepartmentID were not explicitly referenced in the query being described.

sa_disable_auditing_type system procedure

Disables auditing of specific events.

Syntax

sa_disable_auditing_type(' *types* **')**

Arguments

- **types** Use this VARCHAR(128) parameter to specify a comma-delimited string containing one or more of the following values:
  - **all** disables all types of auditing.
  - **connect** disables auditing of both successful and failed connection attempts.
  - **connectFailed** disables auditing of failed connection attempts.
  - **DDL** disables auditing of DDL statements.
  - **options** disables auditing of public options.
  - **permission** disables auditing of permission checks, user checks, and SETUSER statements.
  - **permissionDenied** disables auditing of failed permission and user checks.
  - **triggers** disables auditing in response to trigger events.

Remarks

You can use the sa_disable_auditing_type system procedure to disable auditing of one or more categories of information.

Setting this option to all disables all auditing. You can also disable auditing by setting the PUBLIC.auditing option to Off.

Permissions

DBA authority required

Side effects

None

See also

- "sa_enable_auditing_type system procedure" on page 870
- "Auditing database activity" [*SQL Anywhere Server - Database Administration*]
- "auditing option [database]" [*SQL Anywhere Server - Database Administration*]

Example

To disable all auditing:

```
CALL sa_disable_auditing_type( 'all' );
```

sa_disk_free_space system procedure

Reports information about space available for a dbspace, transaction log, transaction log mirror, and/or temporary file.

Syntax

sa_disk_free_space([*p_dbspace_name*] **)**

Arguments

- **p_dbspace_name** Use this VARCHAR(128) parameter to specify the name of a dbspace, transaction log file, transaction log mirror file, or temporary file.

  If there is a dbspace called log, mirror, or temp, you can prefix the keyword with an underscore. For example, use _log to get information about the log file if a dbspace called log exists.

  Specify SYSTEM to get information about the main database file, TEMPORARY or TEMP to get information about the temporary file, TRANSLOG to get information about the transaction log, or TRANSLOGMIRROR to get information about the transaction log mirror. See "Pre-defined dbspaces" [*SQL Anywhere Server - Database Administration*].

Result set

Column name	Data type	Description
dbspace_name	VARCHAR(128)	This is the dbspace name, transaction log file, transaction log mirror file, or temporary file.
free_space	UNSIGNED BIGINT	The number of free bytes on the volume.
total_space	UNSIGNED BIGINT	The total amount of disk space available on the drive where the dbspace resides.

Remarks

If the *p_dbspace_name* parameter is not specified or is NULL, then the result set contains one row for each dbspace, plus one row for each of the transaction log, transaction log mirror, and temporary file, if they exist. If *p_dbspace_name* is specified, then exactly one or zero rows are returned (zero if no such dbspace exists, or if log or mirror is specified and there is no log or mirror file).

For a list of the names of the pre-defined dbspaces for SQL Anywhere databases, see "Pre-defined dbspaces" [*SQL Anywhere Server - Database Administration*].

Permissions

DBA authority required

Side effects

None

Example

The following example uses the sa_disk_free_space system procedure to return a result set containing information about available space.

```
CALL sa_disk_free_space( );
```

dbspace_name	free_space	total_space
system	10952101888	21410402304

dbspace_name	free_space	total_space
translog	10952101888	21410402304
temporary	10952101888	21410402304

sa_enable_auditing_type system procedure

Enables auditing and specifies which events to audit.

Syntax

sa_enable_auditing_type('*types* ')

Arguments

- **types** Use this VARCHAR(128) parameter to specify a comma-delimited string containing one or more of the following values:
  - **all** enables all types of auditing.
  - **connect** enables auditing of both successful and failed connection attempts.
  - **connectFailed** enables auditing of failed connection attempts.
  - **DDL** enables auditing of DDL statements.
  - **options** enables auditing of public options.
  - **permission** enables auditing of permission checks, user checks, and SETUSER statements.
  - **permissionDenied** enables auditing of failed permission and user checks.
  - **triggers** enables auditing after a trigger event.

Remarks

sa_enable_auditing_type works in conjunction with the PUBLIC.auditing option to enable auditing of specific types of information.

If you set the PUBLIC.auditing option to On, and do not specify which type of information to audit, the default setting (all) takes effect. In this case, all types of auditing information are recorded.

If you set the PUBLIC.auditing option to On, and disable all types of auditing using sa_disable_auditing_type, no auditing information is recorded. To re-establish auditing, you must use sa_enable_auditing_type to specify which type of information you want to audit.

If you set the PUBLIC.auditing option to Off, then no auditing information is recorded, regardless of the sa_enable_auditing_type setting.

Permissions

DBA authority required

Side effects

None

See also

- "sa_disable_auditing_type system procedure" on page 867
- "Auditing database activity" [*SQL Anywhere Server - Database Administration*]
- "auditing option [database]" [*SQL Anywhere Server - Database Administration*]

Example

To enable only option auditing:

```
CALL sa_enable_auditing_type( 'options' );
```

sa_eng_properties system procedure

Reports database server property information.

Syntax

sa_eng_properties( )

Result set

Column name	Data type	Description
PropNum	INTEGER	The database server property number.
PropName	VARCHAR(255)	The database server property name.
PropDescription	VARCHAR(255)	The database server property description.
Value	LONG VARCHAR	The database server property value.

Remarks

Returns the PropNum, PropName, PropDescription, and Value for each available server property. Values are returned for all database server properties and statistics related to database servers. For a list of available database server properties, see "System functions" on page 133.

Permissions

None

Side effects

None

See also

- "Database server properties" [*SQL Anywhere Server - Database Administration*]

Example

The following statement returns a set of available server properties

```
CALL sa_eng_properties( );
```

PropNum	PropName	...
1	IdleWrite	...
2	IdleChkPt	...
...	...	...

sa_flush_cache system procedure

Empties all pages for the current database in the database server cache.

Syntax

sa_flush_cache( )

Remarks

Database administrators can use this procedure to empty the contents of the database server cache for the current database. This is useful in performance measurement to ensure repeatable results.

Permissions

DBA authority required

Side effects

None

sa_flush_statistics system procedure

Saves all cost model statistics in the database server cache.

Syntax

sa_flush_statistics( )

Remarks

Use this procedure to flush current cost model statistics in the database, currently cached, to disk. You can then retrieve the statistics using the sa_get_histogram system procedure, or the Histogram utility (dbhist). When this system procedure runs, the ISYSCOLSTAT system table is updated. Under normal operation it should not be necessary to execute this procedure because the server automatically writes out statistics to disk on a periodic basis.

Permissions

DBA authority required

Side effects

None

See also

- “sa_get_histogram system procedure” on page 876
- “SYSCOLSTAT system view” on page 993
- “Histogram utility (dbhist)” [*SQL Anywhere Server - Database Administration*]

sa_get_bits system procedure

Takes a bit string and returns a row for each bit in the string. By default, only rows with a bit value of 1 are returned.

Syntax

sa_get_bits(*bit_string* [**,** *only_on_bits*])

Arguments

- **bit_string** Use this LONG VARBIT parameter to specify the bit string from which to get the bits. If the *bit_string* parameter is NULL, no rows are returned.
- **only_on_bits** Use this optional BIT to specify whether to return only rows with on bits (bits with the value of 1). Specify 1 (the default) to return only rows with on bits; specify 0 to return rows for all bits in the bit string.

Result set

Column	Data type	Description
bitnum	UNSIGNED INT	The position of the bit described by this row. For example, the first bit in the bit string has bitnum of 1.
bit_val	BIT	The value of the bit at position bitnum. If *only_on_bits* is set to 1, this value is always 1.

Remarks

The sa_get_bits system procedure decodes a bit string, returning one row for each bit in the bit string, indicating the value of the bit. If *only_on_bits* is set to 1 (the default) or NULL, then only rows corresponding to on bits are returned. An optimization allows this case to be processed efficiently for long bit strings that have few on bits. If *only_on_bits* is set to 0, then a row is returned for each bit in the bit string.

For example, the statement `CALL sa_get_bits( '1010' )` returns the following result set, indicating on bits in positions 1 and 3 of the bit string.

bitnum	bit_val
1	1
3	1

The sa_get_bits system procedure can be used to convert a bit string into a relation. This can be used to join a bit string with a table, or to retrieve a bit string as a result set instead of as a single binary value. It can be more efficient to retrieve a bit string as a result set if there are a large number of 0 bits, as these do not need to be retrieved.

Permissions

None

Side effects

None

See also

- "sa_split_list system procedure" on page 951
- "SET_BIT function [Bit array]" on page 300
- "SET_BITS function [Aggregate]" on page 301
- "GET_BIT function [Bit array]" on page 208

Examples

The following example shows how to use the sa_get_bits system procedure to encode a set of integers as a bit string, and then decode it for use in a join:

```
CREATE VARIABLE @s_depts LONG VARBIT;

SELECT  SET_BITS( DepartmentID )
 INTO @s_depts
 FROM Departments
 WHERE DepartmentName like 'S%';

SELECT *
 FROM sa_get_bits( @s_depts ) B
  JOIN Departments D ON B.bitnum = D.DepartmentID;
```

sa_get_dtt system procedure

Reports the current value of the Disk Transfer Time (DTT) model, which is part of the cost model.

Syntax

sa_get_dtt(*dbspace_id* **)**

Arguments

- **dbspace_id** Use this UNSIGNED SMALLINT parameter to specify the database file ID.

Remarks

You can obtain the *dbspace_id* from the SYSDBSPACE system view.

This procedure, intended for internal diagnostic purposes, retrieves data from the ISYSOPTSTAT system table.

Result set

Column name	Data type	Description
BandSize	UNSIGNED INTEGER	Size, in pages, of disk over which random access takes place.
ReadTime	UNSIGNED INTEGER	Amortized cost, in microseconds, of reading one page.
WriteTime	UNSIGNED INTEGER	Amortized cost, in microseconds, of writing one page.

Permissions

None

Side effects

None

See also

- "SYSDBSPACE system view" on page 996
- "SYSOPTSTAT system view" on page 1014
- "sa_get_dtt_groupreads system procedure" on page 875

sa_get_dtt_groupreads system procedure

Estimates and reports the cost of issuing group reads on the database server.

Syntax

sa_get_dtt_groupreads(*dbspace_id* **)**

Arguments

- **dbspace_id** Use this UNSIGNED SMALLINT parameter to specify the database file ID.

Remarks

You can obtain the *dbspace_id* from the SYSDBSPACE system view. The estimates returned by the sa_get_dtt_groupreads system procedure are part of the cost model, and are used to select group reads of appropriate sizes during operations such as sorting.

This procedure, intended for internal diagnostic purposes, retrieves data from the ISYSOPTSTAT system table. Rows are not returned if the specified dbspace does not have any estimates recorded in SYSOPTSTAT. To tailor estimates for specific hardware devices, execute the following statement:

```
ALTER DATABASE CALIBRATE GROUP READ;
```

Result set

Column name	Data type	Description
GroupSize	UNSIGNED INTEGER	Size, in pages, of disk over which random access takes place.
ReadTime	FLOAT	Amortized cost, in microseconds, of reading one page.

Permissions

None

Side effects

None

See also

- "SYSDBSPACE system view" on page 996
- "SYSOPTSTAT system view" on page 1014
- "ALTER DATABASE statement" on page 358
- "sa_get_dtt system procedure" on page 874

sa_get_histogram system procedure

Retrieves the histogram for a column.

Syntax

sa_get_histogram(
 col_name,
 tbl_name
 [, *owner_name*]
)

Arguments

- **col_name** Use this CHAR(128) parameter to specify the column for which to retrieve the histogram.
- **tbl_name** Use this CHAR(128) parameter to specify the table in which *col_name* is found.
- **owner_name** Use this optional CHAR(128) parameter to specify the owner of *tbl_name*.

Result set

Column name	Data type	Description
StepNumber	SMALLINT	Histogram bucket number. The frequency of the first bucket (StepNumber = 0) indicates the selectivity of NULLs.
Low	CHAR(128)	Lowest (inclusive) column value in the bucket.

Column name	Data type	Description
High	CHAR(128)	Highest (exclusive) column value in the bucket.
Frequency	DOUBLE	Selectivity of values in the bucket.

Remarks

This procedure, intended for internal diagnostic purposes, retrieves column statistics from the database server for the specified columns. Note that while these statistics are permanently stored in the ISYSCOLSTAT system table, they are maintained in memory while the server is running, and written to ISYSCOLSTAT periodically. As such, the statistics returned by the sa_get_histogram system procedure may differ from those obtained by selecting from ISYSCOLSTAT at any given point of time.

You can manually update ISYSCOLSTAT with the latest statistics held in memory using the sa_flush_statistics system procedure, however, this is not recommended in a production environment, and should be reserved for diagnostic purposes. See "sa_flush_statistics system procedure" on page 872.

A singleton bucket is indicated by a Low value in the result set being equal to the corresponding High value.

It is recommended that you use the Histogram utility to view histograms. See "Histogram utility (dbhist)" [*SQL Anywhere Server - Database Administration*].

To determine the selectivity of a predicate over a string column, use the ESTIMATE or ESTIMATE_SOURCE functions. For string columns, both sa_get_histogram and the Histogram utility retrieve nothing from the ISYSCOLSTAT system table. Attempting to retrieve string data generates an error. See "ESTIMATE function [Miscellaneous]" on page 195, and "ESTIMATE_SOURCE function [Miscellaneous]" on page 196.

Statistics (including histograms) may not be present for a table or materialized view, for example, if statistics were recently dropped. In this case, the result set for the sa_get_histogram system procedure is empty. To create statistics for a table or materialized view, execute a CREATE STATISTICS statement. See "CREATE STATISTICS statement" on page 516.

Permissions

DBA authority required

Side effects

None

See also

- "Optimizer estimates and column statistics" [*SQL Anywhere Server - SQL Usage*]
- "Histogram utility (dbhist)" [*SQL Anywhere Server - Database Administration*]
- "SYSCOLSTAT system view" on page 993

Example

For example, the following statement retrieves the histogram for the ProductID column of the SalesOrderItems table:

```
CALL sa_get_histogram( 'ProductID', 'SalesOrderItems' );
```

sa_get_request_profile system procedure

Analyzes the request log to determine the execution times of similar statements.

Syntax

sa_get_request_profile(
  [*filename*
  [**,** *conn_id*
  [**,** *first_file*
  [**,** *num_files*]]]]
)

Arguments

- **filename** Use this optional LONG VARCHAR parameter to specify the request logging file name.
- **conn_id** Use this optional UNSIGNED INTEGER parameter to specify the ID number of a connection.
- **first_file** Use this optional INTEGER parameter to specify the first request log file to analyze.
- **num_files** Use this optional INTEGER parameter to specify the number of request log files to analyze.

Remarks

This procedure calls sa_get_request_times to process a request log file, and then summarizes the results into the global temporary table satmp_request_profile. This table contains the statements from the log along with how many times each was executed, and their total, average, and maximum execution times. The table can be sorted in various ways to identify targets for performance optimization efforts.

If you do not specify a log file (*filename*), the default is the current log file that is specified in the command with -zo, or that has been specified by

```
sa_server_option( 'RequestLogFile', filename )
```

If a connection ID is specified, it is used to filter information from the log so that only requests for that connection are retrieved.

Permissions

DBA authority required

Side effects

Automatic commit

Example

The following command obtains the request times for the requests in the files *req.out.3*, *req.out.4*, and *req.out.5*.

```
CALL sa_get_request_profile('req.out',0,3,3);
```

See also

- "sa_get_request_times system procedure" on page 879
- "sa_statement_text system procedure" on page 953
- "sa_server_option system procedure" on page 935

sa_get_request_times system procedure

Analyzes the request log to determine statement execution times.

Syntax

sa_get_request_times(*filename*
 [**,** *conn_id*
 [**,** *first_file*
 [**,** *num_files*]]]
)

Arguments

- **filename** Use this optional LONG VARCHAR parameter to specify the request logging file name.
- **conn_id** Use this optional UNSIGNED INTEGER parameter to specify the ID number of a connection.
- **first_file** Use this optional INTEGER parameter to specify the first file to analyze.
- **num_files** Use this optional INTEGER parameter to specify the number of request log files to analyze.

Remarks

This procedure reads the specified request log and populates the global temporary table satmp_request_time with the statements from the log and their execution times.

For statements such as inserts and updates, the execution time is straightforward. For queries, the time is calculated from preparing the statement to dropping it, including describing it, opening a cursor, fetching rows, and closing the cursor. For most queries, this is an accurate reflection of the amount of time taken. In cases where the cursor is left open while other events take place, such as operator interaction or client processing, the time appears as a large value but is not a true indication that the query is costly.

This procedure recognizes host variables in the request log and populates the global temporary table satmp_request_hostvar with their values. For older databases where this temporary table does not exist, host variable values are ignored.

If you do not specify a log file, the default is the current log file that is specified in the command with -zo, or that has been specified by

```
sa_server_option( 'RequestLogFile', filename )
```

If a connection ID is specified, it is used to filter information from the log so that only requests for that connection are retrieved.

Permissions

DBA authority required

Side effects

Automatic commit

Example

The following command obtains the execution times for the requests in the files *req.out.3*, *req.out.4*, and *req.out.5*.

```
CALL sa_get_request_times('req.out',0,3,3);
```

See also

- "sa_get_request_profile system procedure" on page 878
- "sa_statement_text system procedure" on page 953
- "sa_server_option system procedure" on page 935

sa_get_server_messages system procedure [deprecated]

Allows you to return constants from the database server messages window as a result set.

This system procedure is deprecated. Use sa_server_messages instead. See "sa_server_messages system procedure" on page 933.

Syntax

sa_get_server_messages(*first_line* **)**

Arguments

- **first_line** Use this INTEGER parameter to specify the line number from which to start displaying server messages.

Result set

Column name	Data type	Description
line_num	INTEGER	The line number of a server message.
message_text	VARCHAR(255)	The server message text.
message_time	TIMESTAMP	The time of the message.

Remarks

This procedure takes an INTEGER parameter that specifies the starting line number to display, and returns a row for that line and for all subsequent lines. If the starting line is negative, the result set starts at the first available line. The result set includes the line number, message text, and message time.

Permissions

None

Side effects

None

Example

The following example uses the sa_get_server_messages system procedure to return a result set containing the content of the database server messages window, starting from line 16.

```
CALL sa_get_server_messages( 16 );
```

line_num	message_text	...
16	Running on Windows 2000 Build 2195...	...
17	2132K of memory used for caching	...
...	...	...

sa_get_table_definition system procedure

Returns a LONG VARCHAR string containing the SQL statements required to create the specified table and its indexes, foreign keys, triggers and granted permissions.

Syntax

sa_get_table_definition(*@table_owner*, *@table_name* **)**

Arguments

- **@table_owner** Use this CHAR(128) parameter to specify the owner of *@table_name*.
- **@table_name** Use this CHAR(128) parameter to specify the name of the table.

Remarks

To create a new table with the same definition, use the string returned by the sa_get_table_definition system procedure with the EXECUTE IMMEDIATE statement and the LOCATE, SUBSTRING, and REPLACE functions.

Permissions

DBA authority required

Side effects

None

See also

- "sa_split_list system procedure" on page 951
- "EXECUTE IMMEDIATE statement [SP]" on page 600
- "LOCATE function [String]" on page 240
- "SUBSTRING function [String]" on page 318
- "REPLACE function [String]" on page 289

Example

The following statement uses the sa_get_table_definition system procedure to display the string containing the SQL statements required to create the Departments table.

```
SELECT row_value
FROM sa_split_list( sa_get_table_definition( 'GROUPO', 'Departments'),
CHAR(10)) ;
```

sa_get_user_status system procedure

Allows you to determine the current status of users.

Syntax

sa_get_user_status

Arguments

None

Result set

Column name	Data type	Description
user_id	UNSIGNED INT	A unique number identifying the user.
user_name	CHAR(128)	The name of the user.
connections	INT	The current number of connections by this user.
failed_logins	UNSIGNED INT	The number of failed login attempts made by the user.
last_login_time	TIMESTAMP	The time the user last logged in.
locked	TINYINT	Indicates if the user account is locked.
reason_locked	LONG VARCHAR(255)	The reason the account is locked.

Remarks

This procedure returns a result set that shows the current status of users. In addition to basic user information, the procedure includes a column indicating if the user has been locked out and a column with a reason for the lockout. Users can be locked out for the following reasons: locked due to policy, password expiry, or too many failed attempts.

Permissions

DBA authority required to view information about all users. Users without DBA authority can only view their own information.

Side effects

None

See also

- "Managing login policies overview" [*SQL Anywhere Server - Database Administration*]
- "Creating a new login policy" [*SQL Anywhere Server - Database Administration*]
- "Creating a user and assigning a login policy" [*SQL Anywhere Server - Database Administration*]
- "Assigning a login policy to an existing user" [*SQL Anywhere Server - Database Administration*]
- "Altering a login policy" [*SQL Anywhere Server - Database Administration*]
- "Dropping a login policy" [*SQL Anywhere Server - Database Administration*]

Example

The following example uses the sa_get_user_status system procedure to return the status of database users.

```
CALL sa_get_user_status;
```

sa_http_header_info system procedure

Returns HTTP header names and values.

Syntax

sa_http_header_info([*header_parm*] **)**

Arguments

- **header_parm** Use this optional VARCHAR(255) parameter to specify an HTTP header name.

Result set

Column name	Data type	Description
Name	VARCHAR(255)	The HTTP header name.
Value	LONG VARCHAR	The HTTP header value.

Remarks

The sa_http_header_info system procedure returns header names and values. If you do not specify the header name using the optional parameter, the result set contains values for all headers.

This procedure returns a non-empty result set if it is called while processing an HTTP request within a web service.

Permissions

None

Side effects

None

See also

- “SQL Anywhere web services” [*SQL Anywhere Server - Programming*]
- “-xs server option” [*SQL Anywhere Server - Database Administration*]
- “sa_http_variable_info system procedure” on page 886
- “sa_set_http_header system procedure” on page 944
- “sa_set_http_option system procedure” on page 946
- “Web services functions” on page 130
- “Web services system procedures” on page 830

sa_http_php_page system procedure

Returns the result of passing the PHP code that is to be interpreted through a PHP interpreter using the current HTTP request for context information such as headers, GET/POST data, protocol version, request URL, method, and so on.

Syntax

sa_http_php_page(*php_page* **)**

Arguments

- **php_page** This LONG VARCHAR parameter contains the entire PHP code that is to be interpreted, including the starting and ending markers (`<?php` and `?>`).

Remarks

To use this system procedure, the PHP external environment must already be installed. See “The PHP external environment” [*SQL Anywhere Server - Programming*].

The owner of this system procedure is DBO. However, for improved security, the sa_http_php_page system procedure is executed as the invoker.

Permissions

None

Side effects

None

See also

- "SQL Anywhere web services" [*SQL Anywhere Server - Programming*]
- "-xs server option" [*SQL Anywhere Server - Database Administration*]
- "HTTP_BODY function [HTTP]" on page 223
- "sa_http_php_page_interpreted system procedure" on page 885
- "sa_http_header_info system procedure" on page 883
- "sa_set_http_header system procedure" on page 944
- "sa_set_http_option system procedure" on page 946
- "Web services functions" on page 130
- "Web services system procedures" on page 830

sa_http_php_page_interpreted system procedure

Returns the result of passing the PHP code that is to be interpreted through a PHP interpreter using the specified parameters for context information such as headers, GET/POST data, protocol version, request URL, method, and so on.

Syntax

sa_http_php_page_interpreted(
*php_page***,**
*method***,**
*url***,**
*version***,**
*headers***,**
request_body
)

Arguments

- **php_page** This LONG VARCHAR parameter contains the entire PHP code that is to be interpreted, including the starting and ending markers (`<?php` and `?>`).

- **method** This LONG VARCHAR parameter contains the HTTP request method (for example, GET, POST, PUT, or one of the other standard request methods). The value for *method* can be determined using the value of @HttpMethod in the current HTTP request.

- **url** This LONG VARCHAR parameter contains the full HTTP request URL, including the query string, if present. The value for *url* can be determined using the value of @HttpURI in the current HTTP request.

- **version** This LONG VARCHAR parameter contains the HTTP request protocol version (for example, HTTP/1.1). The value for *version* can be determined using the value of @HttpVersion in the current HTTP request.

- **headers** This LONG BINARY parameter contains the HTTP request headers in the standard HTTP header format: `Field-Name: Value\r\n`. The value for headers can be retrieved from the current HTTP request using the following SELECT statement:

  ```
  SELECT LIST( name || ': ' || value, CHAR(13) || CHAR(10) )
     FROM sa_http_header_info();
  ```

- **request_body** This LONG BINARY parameter contains the HTTP request body in binary form. The value for *request_body* can be retrieved from the current HTTP request using the HTTP_BODY function. See "HTTP_BODY function [HTTP]" on page 223.

Remarks

To use this system procedure, the PHP external environment must already be installed. See "The PHP external environment" [*SQL Anywhere Server - Programming*].

To use this system procedure outside web services requests, you must provide request information. Any headers set within the PHP code are lost.

The owner of this system procedure is DBO. However, for improved security, the sa_http_php_page_interpreted system procedure is executed as the invoker.

Permissions

None

Side effects

None

See also

- "SQL Anywhere web services" [*SQL Anywhere Server - Programming*]
- "-xs server option" [*SQL Anywhere Server - Database Administration*]
- "HTTP_BODY function [HTTP]" on page 223
- "sa_http_php_page system procedure" on page 884
- "sa_http_header_info system procedure" on page 883
- "sa_set_http_header system procedure" on page 944
- "sa_set_http_option system procedure" on page 946
- "Web services functions" on page 130
- "Web services system procedures" on page 830

sa_http_variable_info system procedure

Returns HTTP variable names and values.

Syntax

sa_http_variable_info([*variable_parm*] **)**

Arguments

- **variable_parm** Use this optional VARCHAR(255) parameter to specify an HTTP variable name.

Result set

Column name	Data type	Description
Name	VARCHAR(255)	The HTTP variable name.

Column name	Data type	Description
Value	LONG VARCHAR	The HTTP variable value.

Remarks

The sa_http_variable_info system procedure returns variable names and values. If you do not specify the variable name using the optional parameter, the result set contains values for all variables.

This procedure returns a non-empty result set if it is called while processing an HTTP request within a web service.

Permissions

None

Side effects

None

See also

- "SQL Anywhere web services" [*SQL Anywhere Server - Programming*]
- "-xs server option" [*SQL Anywhere Server - Database Administration*]
- "sa_http_header_info system procedure" on page 883
- "sa_set_http_header system procedure" on page 944
- "sa_set_http_option system procedure" on page 946
- "Web services functions" on page 130
- "Web services system procedures" on page 830

sa_index_density system procedure

Reports information about the amount of fragmentation and skew within indexes.

Syntax

sa_index_density(
  [*tbl_name*
  [, *owner_name*]]
)

Arguments

- **tbl_name** Use this optional CHAR(128) parameter to specify the table name.
- **owner_name** Use this optional CHAR(128) parameter to specify the owner name.

Result set

Column name	Data type	Description
Table-Name	CHAR(128)	The name of a table.
TableId	UNSIGNED INTEGER	The table ID.
Index-Name	CHAR(128)	The name of an index.
IndexId	UNSIGNED INTEGER	The index ID. This column contains one of the following values: ● **0** for primary keys ● **SYSFKEY.*foreign_key_id*** for foreign keys ● **SYSIDX.*index_id*** for all other indexes
Index-Type	CHAR(4)	The index type. This column contains one of the following values: ● **PKEY** for primary keys ● **FKEY** for foreign keys ● **UI** for unique indexes ● **UC** for unique constraints ● **NUI** for non-unique indexes
LeafPages	UNSIGNED INTEGER	The number of leaf pages.
Density	NUMERIC(8,6)	A fraction between 0 and 1 that provides an indication of how full each index page is (on average).
Skew	NUMERIC(8,6)	A number that provides an indication of the level of unbalance in an index. A value of 1 indicates a perfectly balanced index. Larger values indicate a higher degree of skew.

Remarks

Use the sa_index_density system procedure to obtain information about the degree of fragmentation and skew in indexes. For indexes with a high number of leaf pages, higher density values and lower skew values are desirable.

Index density reflects the average fullness of the index pages, as a percentage. A density of 0.7 indicates that index pages are, on average, 70% full with index data. Index skew reflects the typical deviation from the average density. The amount of skew is important to the optimizer when making selectivity estimates.

When the number of leaf pages is low, you do not need to be concerned about density and skew values. Density and skew values become important only when the number of leaf pages are high. When the number of leaf pages is high, a low density value can indicate fragmentation, and a high skew value can indicate that indexes are not well balanced. Both of these can be factors in poor performance. Executing a REORGANIZE TABLE statement addresses both of these issues. See "REORGANIZE TABLE statement" on page 712.

If you do not specify a table when calling this procedure, the information for all indexes on all tables in the database is returned.

You can also use the **Application Profiling Wizard** to determine whether index density and skew are at acceptable levels. See "Application Profiling Wizard" [*SQL Anywhere Server - SQL Usage*].

Permissions

DBA authority required

Side effects

None

See also

- "Reducing index fragmentation and skew" [*SQL Anywhere Server - SQL Usage*]

Example

The following example uses the sa_index_density system procedure to return a result set summarizing the amount of fragmentation and skew within all the indexes in the database.

```
CALL sa_index_density( );
```

sa_index_levels system procedure

Assists in performance tuning by reporting the number of levels in an index.

Syntax

sa_index_levels(
[*tbl_name*
[**,** *owner_name*]]
)

Arguments

- **tbl_name** Use this optional CHAR(128) parameter to specify the table name.
- **owner_name** Use this optional CHAR(128) parameter to specify the owner name.

Result set

Column name	Data type	Description
TableName	CHAR(128)	The name of a table.
TableId	UNSIGNED INTEGER	The table ID.
IndexName	CHAR(128)	The name of an index.
IndexId	UNSIGNED INTEGER	The index ID. This column contains one of the following: ● **0** for primary keys ● **SYSFKEY.*foreign_key_id*** for foreign keys ● **SYSIDX.*index_id*** for all other indexes
IndexType	CHAR(4)	The index type. This column contains one of the following values: ● **PKEY** for primary keys ● **FKEY** for foreign keys ● **UI** for unique indexes ● **UC** for unique constraints ● **NUI** for non-unique indexes
Levels	INTEGER	The number of levels in the index.

Remarks

The number of levels in the index tree determines the number of I/O operations needed to access a row using the index. Indexes with a few levels are more efficient than indexes with a large number of levels.

The procedure returns a result set containing the table name, the table ID, the index name, the index ID, the index type, and the number of levels in the index.

If no arguments are supplied, levels are returned for all indexes in the database. If only *tbl_name* is supplied, levels for all indexes on that table are supplied. If *tbl_name* is NULL and an *owner_name* is given, only levels for indexes on tables owned by that user are returned.

Permissions

DBA authority required

Side effects

None

See also

- "CREATE INDEX statement" on page 470
- "Using indexes" [*SQL Anywhere Server - SQL Usage*]

Example

The following example uses the sa_index_levels system procedure to return the number of levels in the Products index.

```
CALL sa_index_levels( );
```

TableName	TableId	IndexName	...	Levels
Products	436	Products	...	1
...	...	...	...	...

sa_java_loaded_classes system procedure

Lists the classes currently loaded by the database server into a Java Virtual Machine.

Syntax

sa_java_loaded_classes( )

Result set

Column name	Data type	Description
class_name	VARCHAR(512)	The name of a class currently loaded by the database server into a Java Virtual Machine.

Remarks

Returns a result set containing all the names of the Java classes currently loaded by the database server into a Java Virtual Machine.

The procedure can be useful to diagnose missing classes. It can also be used to identify which classes from a particular jar are used by a given application.

Permissions

DBA authority required

Side effects

None

See also

- "Installing Java classes into a database" [*SQL Anywhere Server - Programming*]

sa_load_cost_model system procedure

Replaces the current cost model with the cost model stored in the specified file.

Syntax

sa_load_cost_model (*file_name* **)**

Arguments

- ***file_name*** Use this CHAR(1024) parameter to specify the name of the cost model file to load.

Remarks

The optimizer uses cost models to determine optimal access plans for queries. The database server maintains a cost model for each database. The cost model for a database can be recalibrated at any time using the CALIBRATE SERVER clause of the ALTER DATABASE statement. For example, you might decide to recalibrate the cost model if you move the database onto non-standard hardware.

The sa_load_cost_model system procedure allows you to load a cost model that has been saved to file (*file_name*). Loading a cost model replaces the current cost model for the database.

Note
The sa_unload_cost_model system procedure does not include CALIBRATE PARALLEL READ information in the file that sa_load_cost_model loads.

Using the sa_load_cost_model system procedure can eliminate repetitive, time-consuming recalibration activities when there is a large number of identical hardware installations.

Exclusive use of the database is required when loading the new cost model.

When loading a cost model, consider whether it was generated for a database that is located on similar hardware. Loading a cost model from a database that is stored on significantly different hardware may cause poor performance due to inefficient access plans.

Cost models are saved to file using the sa_unload_cost_model system procedure. See "sa_unload_cost_model system procedure" on page 962.

Permissions

Must have DBA authority.

Side effects

The database server performs a COMMIT after loading the new cost model.

See also

- "ALTER DATABASE statement" on page 358
- "sa_unload_cost_model system procedure" on page 962
- "Query optimization and execution" [*SQL Anywhere Server - SQL Usage*]

Example

The following example loads the cost model from a file called costmodel8:

```
CALL sa_load_cost_model( 'costmodel8' );
```

sa_locks system procedure

Displays all locks in the database.

Syntax

sa_locks(
  [*connection*
  [, *creator*
  [, *table_name*
  [, *max_locks*]]]]
)

Arguments

- **connection** Use this INTEGER parameter to specify a connection ID. The procedure returns lock information only about the specified connection. The default value is 0 (or NULL), in which case information is returned about all connections.
- **creator** Use this CHAR(128) parameter to specify a user ID. The procedure returns information only about the tables owned by the specified user. The default value for the creator parameter is NULL. When this parameter is set to NULL, sa_locks returns the following information:
  - if the table_name parameter is unspecified, locking information is returned for all tables in the database
  - if the table_name parameter is specified, locking information is returned for tables with the specified name that were created by the current user
- **table_name** Use this CHAR(128) parameter to specify a table name. The procedure returns information only about the specified tables. The default value is NULL, in which case information is returned about all tables.
- **max_locks** Use this INTEGER parameter to specify the maximum number of locks for which to return information. The default value is 1000. The value -1 means return all lock information.

Result set

Column name	Data type	Description
conn_name	VARCHAR(128)	The name of the current connection.
conn_id	INTEGER	The ID number of the connection
user_id	CHAR(128)	The user connected through connection ID.

Column name	Data type	Description
table_type	CHAR(6)	The type of table. This type is either BASE for a table, GLBTMP for global temporary table, or MVIEW for a materialized view.
creator	VARCHAR(128)	The owner of the table.
table_name	VARCHAR(128)	The table on which the lock is held.
index_id	INTEGER	The index ID or NULL.
lock_class	CHAR(8)	The lock class. One of Schema, Row, Table, or Position. See "Objects that can be locked" [*SQL Anywhere Server - SQL Usage*].
lock_duration	CHAR(11)	The duration of the lock. One of Transaction, Position, or Connection.
lock_type	CHAR(9)	The lock type (this is dependent on the lock class).
row_identifier	UNSIGNED BIGINT	The identifier for the row. This is either an 8-byte row identifier or NULL.

Remarks

The sa_locks procedure returns a result set containing information about all the locks in the database.

The value in the lock_type column depends on the lock classification in the lock_class column. The following values can be returned:

Lock class	Lock types	Comments
Schema	Shared (shared schema lock) Exclusive (exclusive schema lock)	For schema locks, the row_identifier and index ID values are NULL. See "Schema locks" [*SQL Anywhere Server - SQL Usage*].

Lock class	Lock types	Comments
Row	Read (read lock) Intent (intent lock) Write (write lock) Surrogate (surrogate lock)	Row read locks can be short-term locks (scans at isolation level 1) or can be long-term locks at higher isolation levels. The lock_duration column indicates whether the read lock is of short duration because of cursor stability (Position) or long duration, held until COMMIT/ROLL-BACK (Transaction). Row locks are always held on a specific row, whose 8-byte row identifier is reported as a 64-bit integer value in the row_identifier column. A surrogate lock is a special case of a row lock. Surrogate locks are held on surrogate entries, which are created when referential integrity checking is delayed. See "Locking during inserts" [*SQL Anywhere Server - SQL Usage*]. There is not a unique surrogate lock for every surrogate entry created in a table. Rather, a surrogate lock corresponds to the set of surrogate entries created for a given table by a given connection. The row_identifier value is unique for the table and connection associated with the surrogate lock. See "Row locks" [*SQL Anywhere Server - SQL Usage*].
Table	Shared (shared table lock) Intent (intent to up-date table lock) Exclusive (exclusive table lock)	See "Table locks" [*SQL Anywhere Server - SQL Usage*].
Posi-tion	Phantom (phantom lock) Insert (insert lock)	In most cases, a position lock is also held on a specific row, and that row's 64-bit row identifier appears in the row_identifier column in the result set. However, Position locks can be held on entire scans (index or sequential), in which case the row_identifier column is NULL. See "Position locks" [*SQL Anywhere Server - SQL Usage*].

A position lock can be associated with a sequential table scan, or an index scan. The index_id column indicates whether the position lock is associated with a sequential scan. If the position lock is held because of a sequential scan, the index_id column is NULL. If the position lock is held as the result of a specific index scan, the index identifier of that index is listed in the index_id column. The index identifier corresponds to the primary key of the ISYSIDX system table, which can be viewed using the SYSIDX view. If the position lock is held for scans over all indexes, the index ID value is -1.

Permissions

DBA authority required

Side effects

None

See also

- “How locking works” [*SQL Anywhere Server - SQL Usage*]
- “SYSIDX system view” on page 1005

Example

For an example of this system procedure, and tips to augment the amount of information you can return, see “Obtaining information about locks” [*SQL Anywhere Server - SQL Usage*].

sa_make_object system procedure

Ensures that a skeletal instance of an object exists before executing an ALTER statement.

Syntax

sa_make_object(
 objtype,
 objname
 [**,** *owner*
 [**,** *tabname*]]
)

objtype:
'procedure'
| **'function'**
| **'view'**
| **'trigger'**
| **'service'**
| **'event'**

Arguments

- **objtype** Use this CHAR(30) parameter to specify the type of object being created. If objtype is **'trigger'**, this argument specifies the owner of the table on which the trigger is to be created.

- **objname** Use this CHAR(128) parameter to specify the name of the object to be created.

- **owner** Use this optional CHAR(128) parameter to specify the owner of the object to be created. The default value is CURRENT USER.

- **tabname** This CHAR(128) parameter is required only if objtype is 'trigger', in which case you use it to specify the name of the table on which the trigger is to be created.

Remarks

This procedure is useful in scripts or command files that are run repeatedly to create or modify a database schema. A common problem in such scripts is that the first time they are run, a CREATE statement must be executed, but subsequent times an ALTER statement must be executed. This procedure avoids the necessity of querying the system views to find out whether the object exists.

To use the procedure, follow it by an ALTER statement that contains the entire object definition.

Permissions

Resource authority is required to create or modify database objects

Side effects

Automatic commit

See also

- "ALTER EVENT statement" on page 365
- "ALTER FUNCTION statement" on page 369
- "ALTER PROCEDURE statement" on page 376
- "ALTER TRIGGER statement" on page 401
- "ALTER VIEW statement" on page 403
- "ALTER SERVICE statement" on page 382

Examples

The following statements ensure that a skeleton procedure definition is created, define the procedure, and grant permissions on it. A command file containing these instructions could be run repeatedly against a database without error.

```
CALL sa_make_object( 'procedure','myproc' );
ALTER PROCEDURE myproc( in p1 INT, in p2 CHAR(30) )
BEGIN
    // ...
END;
GRANT EXECUTE ON myproc TO public;
```

The following example uses the sa_make_object system procedure to add a skeleton web service.

```
CALL sa_make_object( 'service','my_web_service' );
```

sa_materialized_view_can_be_immediate system procedure

Returns whether the specified materialized view can be defined as immediate.

Syntax

sa_materialized_view_can_be_immediate(
view_name
, *owner_name*
)

Arguments

- **view_name** Use this CHAR(128) parameter to specify the name of the materialized view. If *view_name* is NULL, a `Table...not found` error is returned.

- **owner_name** Use this CHAR(128) parameter to specify the owner of the materialized view. If *owner_name* is NULL, a `Table...not found` error is returned.

Remarks

There are restrictions on whether the specified manual view can be changed to an immediate view. Use this system procedure to determine whether the change is permitted. For a list of the additional restrictions for creating immediate views, see "Additional restrictions for immediate views" [*SQL Anywhere Server - SQL Usage*].

The sa_materialized_view_can_be_immediate system procedure returns the following information for the specified materialized view.

Column name	Data type	Description
SQLStateVal	CHAR(6)	The SQLSTATE returned.
ErrorMessage	LONG VARCHAR	The error message corresponding to the SQLSTATE.

Each row in the result set corresponds to a single SQLSTATE returned for a view. So, if the materialized view definition violates more than one restriction, the results include multiple rows for the view.

You can combine the output of this system procedure with the output of the sa_materialized_view_info system procedure to get information on the status of views and whether they can be made immediate. See the Example section of "sa_materialized_view_info system procedure" on page 899.

Permissions

DBA authority, or execute permissions on DBO owned procedures.

See also

- "Change a manual view to an immediate view" [*SQL Anywhere Server - SQL Usage*]
- "Additional restrictions for immediate views" [*SQL Anywhere Server - SQL Usage*]
- "sa_materialized_view_info system procedure" on page 899

Side effects

All metadata for the specified materialized view, and all dependencies, are loaded into the server cache.

Example

Execute the following statements to create a manual view, view10, and refresh it:

```
CREATE MATERIALIZED VIEW view10
   AS (SELECT C.ID, C.Surname, sum(P.UnitPrice) as revenue, C.CompanyName,
SO.OrderDate
         FROM Customers C, SalesOrders SO, SalesOrderItems SOI, Products P
         WHERE C.ID = SO.CustomerID
         AND SO.ID = SOI.ID
         AND P.ID = SOI.ProductID
         GROUP BY C.ID, C.Surname, C.CompanyName, SO.OrderDate);
REFRESH MATERIALIZED VIEW view10;
```

Use the following query to find the reasons why view10 cannot be changed to an immediate view:

```
SELECT SQLStateVal AS "SQLstate", ErrorMessage AS Description
   FROM sa_materialized_view_can_be_immediate( 'view10', 'DBA' )
   ORDER BY SQLSTATE;
```

SQLstate	Description
42WC3	The materialized view view10 cannot be changed to immediate because it has already been initialized.
42WCA	The materialized view view10 cannot be changed to immediate because it does not have a unique index on non-nullable columns.
42WC6	The materialized view cannot be changed to immediate because COUNT(*) is required to be part of the SELECT list.
42WC7	The materialized view cannot be changed to immediate because it does not have a unique index on non-aggregate non-nullable columns.

sa_materialized_view_info system procedure

Returns information about the specified materialized views.

Syntax

sa_materialized_view_info(
[*view_name*
[, *owner_name*]]
)

Arguments

- **view_name** Use this optional CHAR(128) parameter to specify the name of the materialized view for which to return information.
- **owner_name** Use this optional CHAR(128) parameter to specify the owner of the materialized view.

Remarks

If neither *view_name* nor *owner_name* are specified, information about all materialized views in the database is returned.

If *owner_name* is not specified, information about all materialized views named *view_name* is returned.

The sa_materialized_view_info system procedure returns the following information for a materialized view:

Column name	Data type	Description
OwnerName	CHAR(128)	The owner of the view.
ViewName	CHAR(128)	The name of the view.

Column name	Data type	Description
Status	CHAR(1)	Status information about the view. Possible values are: ● **D** disabled ● **E** enabled
DataStatus	CHAR(1)	Status information about data in the view. Possible values are: ● **E** An error occurred during the last refresh attempt. The view is enabled, but uninitialized. ● **F** The underlying tables have not changed since the last refresh, and the view is considered fresh. The view is enabled and initialized. ● **N** The view is uninitialized. This occurs when one of the following is true: ○ the view has not been refreshed since it was created ○ the data has been truncated from the view ○ the view is disabled ● **S** An underlying table has changed since the last refresh, and the view is considered stale. The view is enabled and initialized.
ViewLastRefreshed	TIMESTAMP	The time when the view was last refreshed. If the value of ViewLastRefreshed is NULL, the view is uninitialized.
DataLastModified	TIMESTAMP	For a stale view, the last time that underlying data was modified. The value is NULL for views that are not initialized, or for views that are not considered stale.

Column name	Data type	Description
AvailForOptimization	CHAR(1)	Information about the availability of the view for use by the optimizer. Possible values are: ● **D** Use by the optimizer is disabled. The owner of the view doesn't allow the view to be used by the optimizer. ● **I** The view cannot be used by the optimizer for some internal reason, for example its definition doesn't meet the conditions required. However, the owner has not explicitly disallowed its use by the optimizer. ● **N** The view contains no data because a refresh has not been done or has failed. The view is allowed to be used by the optimizer by the owner of the view, but it is not initialized. ● **O** There is an incompatible option value for current connection. The view is allowed to be used by the optimizer and its definition meets all the required conditions, but the current option settings are not compatible with the options settings used to create the view. ● **Y** The view can be used by the optimizer. The owner of the view allows the view to be used by the optimizer and the view definition meets all the conditions needed to be used by the optimizer. For more information about how, and whether, a materialized view is selected by the optimizer, see "Improving performance with materialized views" [*SQL Anywhere Server - SQL Usage*].
RefreshType	CHAR(1)	The refresh type for the view. Possible values are: ● **I** The view is an immediate view. Immediate views are refreshed immediately when changes to the data in an underlying table impact the data in the materialized view. ● **M** The view is a manual view. Manual views are refreshed manually, for example using the REFRESH MATERIALIZED VIEW statement, or the sa_refresh_materialized_views system procedure. For more information about manual and immediate views, see "Manual and immediate materialized views" [*SQL Anywhere Server - SQL Usage*].

This procedure can be useful for determining the list of materialized views that will never be considered by the optimizer because of a problem with the view definition. The AvailForOptimization value is I for these materialized views. To learn more about the restrictions for materialized view definitions, see "Restrictions on materialized views" [*SQL Anywhere Server - SQL Usage*].

The following table shows how the AvailForOptimization property is determined. Starting from the left column, you read across the row to see the conditions that must be in place to result in the value found in the AvailForOptimization column.

User allows view to be used in optimization?	The view definition satisfies all the conditions required for use?	The connection options match those required for use of the view?	The view is initialized?	AvailForOptimization value
Yes	Yes	Yes	Yes	Y
No	N/A	N/A	N/A	D
Yes	No	N/A	Yes	I
Yes	N/A	N/A	No	N
Yes	Yes	No	Yes	O

An initialized materialized view can be empty. This occurs when there is no data in the underlying tables that meets the materialized view definition. An empty view is not considered the same as an unitialized materialized view, which also has no data in it. The value of the ViewLastRefreshed property allows you to distinguish between whether the view is uninitialized (NULL), or empty because of data in the underlying tables (non-NULL).

Permissions

DBA authority, or execute permissions on DBO owned procedures.

Side effects

All metadata for the specified materialized views, and all dependencies, are loaded into the database server cache.

See also

- "Improving performance with materialized views" [*SQL Anywhere Server - SQL Usage*]
- "Change a manual view to an immediate view" [*SQL Anywhere Server - SQL Usage*]
- "Additional restrictions for immediate views" [*SQL Anywhere Server - SQL Usage*]
- "sa_materialized_view_can_be_immediate system procedure" on page 897

Example

The following statement returns information on all materialized views in the database:

```
SELECT *
   FROM sa_materialized_view_info();
```

The results of the sa_materialized_view_info system procedure can be combined with the results of the sa_materialized_view_can_be_immediate system procedure to return status information, and whether the view is eligible for being an immediate view. Execute the following statements to create materialized views that are examined for this example:

```
CREATE MATERIALIZED VIEW view0 AS (
   SELECT ID, Name, Description, Size
   FROM Products
   WHERE Quantity > 0 );
CREATE UNIQUE INDEX u_view0
   ON view0( ID );
ALTER MATERIALIZED VIEW view0
   IMMEDIATE REFRESH;
CREATE MATERIALIZED VIEW view00 AS (
   SELECT ID, Name, Description, Size
   FROM Products
   WHERE Quantity <= 0 );
CREATE UNIQUE INDEX u_view00
   ON view00( ID );
CREATE MATERIALIZED VIEW view1 AS (
   SELECT ID, Name, Description, Size
   FROM Products
   WHERE Quantity = 0 );
ALTER MATERIALIZED VIEW view1
   DISABLE;
CREATE MATERIALIZED VIEW view100
   AS (SELECT C.ID, C.Surname, sum(P.UnitPrice) as revenue, C.CompanyName,
SO.OrderDate
         FROM Customers C, SalesOrders SO, SalesOrderItems SOI, Products P
         WHERE C.ID = SO.CustomerID
         AND SO.ID = SOI.ID
         AND P.ID = SOI.ProductID
         GROUP BY C.ID, C.Surname, C.CompanyName, SO.OrderDate);
REFRESH MATERIALIZED VIEW view100;
```

Execute the following statement to return the status and eligibility information for the views you just created:

```
SELECT ViewName, Status, ViewLastRefreshed, AvailForOptimization,
RefreshType, CanBeImmediate
FROM sa_materialized_view_info() AS V,
   LATERAL( SELECT LIST( ErrorMessage )
            FROM sa_materialized_view_can_be_immediate( V.ViewName,
V.OwnerName ) ) AS I( CanBeImmediate );
```

View-Name	Status	ViewLast-Refreshed	AvailFor-Optimization	RefreshType	CanBeImmediate
view0	E	(NULL)	N	I	
view00	E	(NULL)	N	M	
view1	D	(NULL)	N	M	Cannot use view 'view1' because it has been disabled

View-Name	Status	ViewLast-Refreshed	AvailFor-Optimization	RefreshType	CanBeImmediate
view100	E	2008-02-12 16:47:00.000	Y	M	The materialized view view10 cannot be changed to immediate because it has already been initialized. The materialized view view10 cannot be changed to immediate because it does not have a unique index on non-nullable columns. The materialized view cannot be changed to immediate because COUNT(*) is required to be part of the select list. The materialized view cannot be changed to immediate because it does not have a unique index on non-aggregate non-nullable columns'

From the results you can see that:

- view0 was never refreshed and is an immediate view.
- view00 was never refreshed and is a manual view.
- view1 is disabled
- view100 is a manual view that was last refreshed at 2008-02-12 16:47:00.000.
- view00 can be changed to an immediate view because there are no error messages in the CanBeImmediate column.
- view1 and view100 cannot be changed to immediate views for the reasons listed in the CanBeImmediate column.

sa_migrate system procedure

Migrates a set of remote tables to a SQL Anywhere database.

Syntax

sa_migrate(
base_table_owner,
server_name
[, *table_name*]
[, *owner_name*]
[, *database_name*]
[, *migrate_data*]
[, *drop_proxy_tables*]
[, *migrate_fkeys*]
)

Arguments

- **base_table_owner** Use this VARCHAR(128) parameter to specify the user on the target SQL Anywhere database who owns the migrated tables. Use the GRANT CONNECT statement to create this user. A value is required for this parameter. See "GRANT statement" on page 627.
- **server_name** Use this VARCHAR(128) parameter to specify the name of the remote server that is being used to connect to the remote database. Use the CREATE SERVER statement to create this server. A value is required for this parameter. See "CREATE SERVER statement" on page 506.
- **table_name** If you are migrating a single table, use this VARCHAR(128) parameter to specify the table name. Otherwise, you should specify NULL (the default) for this parameter. Do not specify NULL for both the table_name and owner_name parameters.
- **owner_name** If you are migrating only tables that belong to one owner, use this VARCHAR(128) parameter to specify the owner's name. Otherwise, you should enter NULL (the default) for this parameter. Do not specify NULL for both the table_name and owner_name parameters.
- **database_name** Use this VARCHAR(128) parameter to specify the name of the remote database. You must specify the database name if you want to migrate tables from only one database on the remote server. Otherwise, enter NULL (the default) for this parameter.
- **migrate_data** Use this optional BIT parameter to specify whether the data in the remote tables is migrated. This parameter can be 0 (do not migrate data) or 1 (migrate data). By default, data is migrated. (1)
- **drop_proxy_tables** Use this optional BIT parameter to specify whether the proxy tables created for the migration process are dropped once the migration is complete. This parameter can be 0 (proxy tables are not dropped) or 1 (proxy tables are dropped). By default, the proxy tables are dropped (1).
- **migrate_fkeys** Use this optional BIT parameter to specify whether the foreign key mappings are migrated. This parameter can be 0 (do not migrate foreign key mappings) or 1 (migrate foreign key mappings). By default, the foreign key mappings are migrated (1).

Remarks

You can use this procedure to migrate tables to SQL Anywhere from a remote Oracle, DB2, SQL Server, Adaptive Server Enterprise, or SQL Anywhere database. This procedure allows you to migrate in one step a set of remote tables, including their foreign key mappings, from the specified server. The sa_migrate system procedure calls the following system procedures:

- sa_migrate_create_remote_table_list
- sa_migrate_create_tables
- sa_migrate_data
- sa_migrate_create_remote_fks_list
- sa_migrate_create_fks
- sa_migrate_drop_proxy_tables

You might want to use these system procedures instead of sa_migrate if you need more flexibility. For example, if you are migrating tables with foreign key relationships that are owned by different users, you cannot retain the foreign key relationships if you use sa_migrate.

Before you can migrate any tables, you must first create a remote server to connect to the remote database using the CREATE SERVER statement. You may also need to create an external login to the remote database using the CREATE EXTERNLOGIN statement. See "CREATE SERVER statement" on page 506 and "CREATE EXTERNLOGIN statement" on page 457.

You can migrate all the tables from the remote database to a SQL Anywhere database by specifying only the *base_table_owner* and *server_name* parameters. However, if you specify only these two parameters, all the tables that are migrated will belong to one owner in the target SQL Anywhere database. If tables have different owners on the remote database and you want them to have different owners on the SQL Anywhere database, then you must migrate the tables for each owner separately, specifying the *base_table_owner* and *owner_name* parameters each time you call the sa_migrate procedure.

Caution
Do not specify NULL for both the *table_name* and *owner_name* parameters. Supplying NULL for both the *table_name* and *owner_name* parameters migrates all the tables in the database, including system tables. As well, tables that have the same name, but different owners in the remote database all belong to one owner in the target database. It is recommended that you migrate tables associated with one owner at a time.

Permissions

None

Side effects

None

See also

- "Migrating databases to SQL Anywhere" [*SQL Anywhere Server - SQL Usage*]
- "sa_migrate_create_remote_table_list system procedure" on page 909
- "sa_migrate_create_tables system procedure" on page 911
- "sa_migrate_data system procedure" on page 912
- "sa_migrate_create_remote_fks_list system procedure" on page 908
- "sa_migrate_create_fks system procedure" on page 907
- "sa_migrate_drop_proxy_tables system procedure" on page 913

Examples

The following statement migrates all the tables belonging to user p_chin from the remote database, including foreign key mappings; migrates the data in the remote tables; and drops the proxy tables when migration is complete. In this example, all the tables that are migrated belong to local_user in the target SQL Anywhere database.

```
CALL sa_migrate( 'local_user', 'server_a', NULL, 'p_chin', NULL, 1, 1, 1 );
```

The following statement migrates only the tables that belong to user remote_a from the remote database. In the target SQL Anywhere database, these tables belong to the user local_a. Proxy tables created during the migration are not dropped at completion.

```
CALL sa_migrate( 'local_a', 'server_a', NULL, 'remote_a', NULL, 1, 0, 1 );
```

sa_migrate_create_fks system procedure

Creates foreign keys for each table listed in the dbo.migrate_remote_fks_list table.

Syntax

sa_migrate_create_fks(*i_table_owner* **)**

Arguments

- **i_table_owner** Use this VARCHAR(128) parameter to specify the user on the target SQL Anywhere database who owns the migrated foreign keys. If you want to migrate tables that belong to different user, you must execute this procedure for each user whose tables you want to migrate. The *i_table_owner* is created using the GRANT CONNECT statement. A value is required for this parameter. See "GRANT statement" on page 627.

Remarks

This procedure creates foreign keys for each table that is listed in the dbo.migrate_remote_fks_list table. The user specified by the *i_table_owner* argument owns the foreign keys in the target database.

If the tables in the target SQL Anywhere database do not all have the same owner, you must execute this procedure for each user who owns tables for which you need to migrate foreign keys.

Note
This system procedure is used in conjunction with several other migration system procedures, which must be executed in sequence as listed below:

1. sa_migrate_create_remote_table_list
2. sa_migrate_create_tables
3. sa_migrate_data
4. sa_migrate_create_remote_fks_list
5. sa_migrate_create_fks
6. sa_migrate_drop_proxy_tables

As an alternative, you can migrate all tables in one step using the sa_migrate system procedure.

Permissions

None

Side effects

None

See also

- "Migrating databases to SQL Anywhere" [*SQL Anywhere Server - SQL Usage*]
- "sa_migrate system procedure" on page 905
- "sa_migrate_create_remote_table_list system procedure" on page 909
- "sa_migrate_create_tables system procedure" on page 911
- "sa_migrate_data system procedure" on page 912
- "sa_migrate_create_remote_fks_list system procedure" on page 908
- "sa_migrate_drop_proxy_tables system procedure" on page 913

Example

The following statement creates foreign keys based on the dbo.migrate_remote_fks_list table. The foreign keys belong to the user local_a on the local SQL Anywhere database.

```
CALL sa_migrate_create_fks( 'local_a' );
```

sa_migrate_create_remote_fks_list system procedure

Populates the dbo.migrate_remote_fks_list table.

Syntax

sa_migrate_create_remote_fks_list(*server_name* **)**

Arguments

- **server_name** Use this VARCHAR(128) parameter to specify the name of the remote server that is being used to connect to the remote database. The remote server is created with the CREATE SERVER statement. A value is required for this parameter. See "CREATE SERVER statement" on page 506.

Remarks

This procedure populates the dbo.migrate_remote_fks_list table with a list of foreign keys that can be migrated from the remote database. You can delete rows from this table for foreign keys that you do not want to migrate.

As an alternative, you can migrate all tables in one step using the sa_migrate system procedure.

This system procedure is used in conjunction with several other migration system procedures. The note in the Remarks section of the sa_migrate_create_fks system procedure contains the list of migrate procedures, and the order in which you must execute them. See "sa_migrate_create_fks system procedure" on page 907.

Permissions

None

Side effects

None

See also

- "Migrating databases to SQL Anywhere" [*SQL Anywhere Server - SQL Usage*]
- "sa_migrate system procedure" on page 905
- "sa_migrate_create_remote_table_list system procedure" on page 909
- "sa_migrate_create_tables system procedure" on page 911
- "sa_migrate_data system procedure" on page 912
- "sa_migrate_create_fks system procedure" on page 907
- "sa_migrate_drop_proxy_tables system procedure" on page 913

Example

The following statement creates a list of foreign keys that are in the remote database.

```
CALL sa_migrate_create_remote_fks_list( 'server_a' );
```

sa_migrate_create_remote_table_list system procedure

Populates the dbo.migrate_remote_table_list table.

Syntax

sa_migrate_create_remote_table_list(
 i_server_name
 [**,** *i_table_name*
 [**,** *i_owner_name*

```
  [, i_database_name ] ] ]
)
```

Arguments

- **i_server_name** Use this VARCHAR(128) parameter to specify the name of the remote server that is being used to connect to the remote database. The remote server is created with the CREATE SERVER statement. A value is required for this parameter. See "CREATE SERVER statement" on page 506.
- **i_table_name** Use this optional VARCHAR(128) parameter to specify the name(s) of the tables that you want to migrate, or NULL to migrate all the tables. The default is NULL. Do not specify NULL for both the *i_table_name* and *i_owner_name* parameters.
- **i_owner_name** Use this optional VARCHAR(128) parameter to specify the user who owns the tables on the remote database that you want to migrate, or NULL to migrate all the tables. The default is NULL. Do not specify NULL for both the *i_table_name* and *i_owner_name* parameters
- **i_database_name** Use this optional VARCHAR(128) parameter to specify the name of the remote database from which you want to migrate tables. This parameter is NULL by default. When migrating tables from Adaptive Server Enterprise and Microsoft SQL Server databases, you must specify the database name.

Remarks

This procedure populates the dbo.migrate_remote_table_list table with a list of tables that can be migrated from the remote database. You can delete rows from this table for remote tables that you do not want to migrate.

If you do not want all the migrated tables to have the same owner on the target SQL Anywhere database, you must execute this procedure for each user whose tables you want to migrate.

As an alternative, you can migrate all tables in one step using the sa_migrate system procedure.

Caution
Do not specify NULL for both the *i_table_name* and *i_owner_name* parameters. Supplying NULL for both the *i_table_name* and *i_owner_name* parameters migrates all the tables in the database, including system tables. As well, tables that have the same name, but different owners in the remote database all belong to one owner in the target database. It is recommended that you migrate tables associated with one owner at a time.

This system procedure is used in conjunction with several other migration system procedures. The note in the Remarks section of the sa_migrate_create_fks system procedure contains the list of migrate procedures, and the order in which you must execute them. See "sa_migrate_create_fks system procedure" on page 907.

Permissions

None

Side effects

None

See also

- “Migrating databases to SQL Anywhere” [*SQL Anywhere Server - SQL Usage*]
- “sa_migrate system procedure” on page 905
- “sa_migrate_create_tables system procedure” on page 911
- “sa_migrate_data system procedure” on page 912
- “sa_migrate_create_remote_fks_list system procedure” on page 908
- “sa_migrate_create_fks system procedure” on page 907
- “sa_migrate_drop_proxy_tables system procedure” on page 913

Example

The following statement creates a list of tables that belong to the user remote_a on the remote database.

```
CALL sa_migrate_create_remote_table_list( 'server_a', NULL, 'remote_a',
NULL );
```

sa_migrate_create_tables system procedure

Creates a proxy table and base table for each table listed in the dbo.migrate_remote_table_list table.

Syntax

sa_migrate_create_tables(*i_table_owner* **)**

Arguments

- **i_table_owner** Use this VARCHAR(128) parameter to specify the user on the target SQL Anywhere database who owns the migrated tables. This user is created using the GRANT CONNECT statement. A value is required for this parameter. See “GRANT statement” on page 627.

Remarks

This procedure creates a base table and proxy table for each table listed in the dbo.migrate_remote_table_list table (created using the sa_migrate_create_remote_table_list procedure). These proxy tables and base tables are owned by the user specified by the *i_table_owner* argument. This procedure also creates the same primary key indexes and other indexes for the new table that the remote table has in the remote database.

If you do not want all the migrated tables to have the same owner on the target SQL Anywhere database, you must execute the sa_migrate_create_remote_table_list procedure and the sa_migrate_create_tables procedure for each user who will own migrated tables.

As an alternative, you can migrate all tables in one step using the sa_migrate system procedure.

This system procedure is used in conjunction with several other migration system procedures. The note in the Remarks section of the sa_migrate_create_fks system procedure contains the list of migrate procedures, and the order in which you must execute them. See “sa_migrate_create_fks system procedure” on page 907.

Permissions

None

Side effects

None

See also

- “Migrating databases to SQL Anywhere” [*SQL Anywhere Server - SQL Usage*]
- “sa_migrate system procedure” on page 905
- “sa_migrate_create_remote_table_list system procedure” on page 909
- “sa_migrate_data system procedure” on page 912
- “sa_migrate_create_remote_fks_list system procedure” on page 908
- “sa_migrate_create_fks system procedure” on page 907
- “sa_migrate_drop_proxy_tables system procedure” on page 913

Example

The following statement creates base tables and proxy tables on the target SQL Anywhere database. These tables belong to the user local_a.

```
CALL sa_migrate_create_tables( 'local_a' );
```

sa_migrate_data system procedure

Migrates data from the remote database tables to the target SQL Anywhere database.

Syntax

sa_migrate_data(*i_table_owner* **)**

Arguments

- **i_table_owner** Use this VARCHAR(128) parameter to specify the user on the target SQL Anywhere database who owns the migrated tables. This user is created using the GRANT CONNECT statement. A value is required for this parameter. See “GRANT statement” on page 627.

Remarks

This procedure migrates the data from the remote database to the SQL Anywhere database for tables belonging to the user specified by the *i_table_owner* argument.

When the tables on the target SQL Anywhere database do not all have the same owner, you must execute this procedure for each user whose tables have data that you want to migrate.

As an alternative, you can migrate all tables in one step using the sa_migrate system procedure.

This system procedure is used in conjunction with several other migration system procedures. The note in the Remarks section of the sa_migrate_create_fks system procedure contains the list of migrate procedures, and the order in which you must execute them. See “sa_migrate_create_fks system procedure” on page 907.

Permissions

None

Side effects

None

See also

- "Migrating databases to SQL Anywhere" [*SQL Anywhere Server - SQL Usage*]
- "sa_migrate system procedure" on page 905
- "sa_migrate_create_remote_table_list system procedure" on page 909
- "sa_migrate_create_tables system procedure" on page 911
- "sa_migrate_create_remote_fks_list system procedure" on page 908
- "sa_migrate_create_fks system procedure" on page 907
- "sa_migrate_drop_proxy_tables system procedure" on page 913

Example

The following statement migrates data to the target SQL Anywhere database for tables that belong to the user local_a.

```
CALL sa_migrate_data( 'local_a' );
```

sa_migrate_drop_proxy_tables system procedure

Drops the proxy tables that were created for migration purposes.

Syntax

sa_migrate_drop_proxy_tables(*i_table_owner* **)**

Arguments

- **i_table_owner** Use this VARCHAR(128) parameter to specify the user on the target SQL Anywhere database who owns the proxy tables. This user is created using the GRANT CONNECT statement. A value is required for this parameter. See "GRANT statement" on page 627.

Remarks

This procedure drops the proxy tables that were created for the migration. The user who owns these proxy tables is specified by the *i_table_owner* argument.

If the migrated tables are not all owned by the same user on the target SQL Anywhere database, you must call this procedure for each user to drop all the proxy tables.

As an alternative, you can migrate all tables in one step using the sa_migrate system procedure.

This system procedure is used in conjunction with several other migration system procedures. The note in the Remarks section of the sa_migrate_create_fks system procedure contains the list of migrate procedures, and the order in which you must execute them. See "sa_migrate_create_fks system procedure" on page 907.

Permissions

None

Side effects

None

See also

- "Migrating databases to SQL Anywhere" [*SQL Anywhere Server - SQL Usage*]
- "sa_migrate system procedure" on page 905
- "sa_migrate_create_remote_table_list system procedure" on page 909
- "sa_migrate_create_tables system procedure" on page 911
- "sa_migrate_data system procedure" on page 912
- "sa_migrate_create_remote_fks_list system procedure" on page 908
- "sa_migrate_create_fks system procedure" on page 907

Example

The following statement drops the proxy tables on the target SQL Anywhere database that belong to the user local_a.

```
CALL sa_migrate_drop_proxy_tables( 'local_a' );
```

sa_nchar_terms system procedure

Breaks an NCHAR string into terms and returns each term as a row along with its position.

Syntax

sa_nchar_terms('*char-string*' [**,** '*text-config-name*' [**,** '*owner*']]]
)

Arguments

- **char-string** The NCHAR string you are parsing.
- **text-config-name** The text configuration object to apply when processing the string. The default is 'default_nchar'.
- **owner** The owner of the specified text configuration object. The default is DBA.

Remarks

You can use this system procedure to find out how a string is interpreted when the settings for a text configuration object are applied. This can be helpful when you want to know what terms would be dropped during indexing or from a query string.

Permissions

None

Side effects

None

See also

- “Full text searching” [*SQL Anywhere Server - SQL Usage*]
- “Text configuration objects” [*SQL Anywhere Server - SQL Usage*]
- “sa_char_terms system procedure” on page 843

Example

The syntax for this system procedure is similar to the sa_char_terms system procedure. See the Example section in “sa_char_terms system procedure” on page 843.

sa_performance_diagnostics system procedure

Returns a summary of request timing information for all connections when the database server has request timing logging enabled.

Syntax

sa_performance_diagnostics( )

Result set

Column name	Data type	Description
Number	INT	The ID number of the connection.
Name	VARCHAR(255)	The name of the connection.
Userid	VARCHAR(255)	The user ID for the connection.
DBNumber	INT	The ID number of the database.
LoginTime	TIMESTAMP	The date and time the connection was established.
TransactionStart-Time	TIMESTAMP	The time the database was first modified after a COMMIT or ROLLBACK, or an empty string if no modifications have been made to the database since the last COMMIT or ROLLBACK.
LastReqTime	TIMESTAMP	The time at which the last request for the specified connection started.
ReqType	VARCHAR(255)	The type of the last request.

Column name	Data type	Description
ReqStatus	VARCHAR(255)	The status of the request. It can be one of the following values: ● **Idle** The connection is not currently processing a request. ● **Unscheduled** The connection has work to do and is waiting for a worker thread. ● **BlockedIO** The connection is blocked waiting for an I/O. ● **BlockedContention** The connection is blocked waiting for access to shared database server data structures. ● **BlockedLock** The connection is blocked waiting for a locked object. ● **Executing** The connection is executing a request.
ReqTimeUnscheduled	DOUBLE	The time spent unscheduled.
ReqTimeActive	DOUBLE	The time spent waiting to process requests.
ReqTimeBlockIO	DOUBLE	The time spent waiting for I/O to complete.
ReqTimeBlockLock	DOUBLE	The time spent waiting for a lock.
ReqTimeBlockContention	DOUBLE	The time spent waiting for atomic access.
ReqCountUnscheduled	INT	The number of times waited for scheduling.
ReqCountActive	INT	The number of requests processed.
ReqCountBlockIO	INT	The number of times waited for I/O to complete.
ReqCountBlockLock	INT	The number of times waited for a lock.
ReqCountBlockContention	INT	The number of times waited for atomic access.
LastIdle	INT	The number of ticks between requests.

Column name	Data type	Description
BlockedOn	INT	If the current connection isn't blocked, this is zero. If it is blocked, the connection number on which the connection is blocked due to a locking conflict.
UncommitOp	INT	The number of uncommitted operations.
CurrentProcedure	VARCHAR(255)	The procedure that a connection is currently executing. If the connection is executing nested procedure calls, the name is the name of the current procedure. If there is no procedure executing, an empty string is returned
EventName	VARCHAR(255)	The name of the associated event if the connection is running an event handler. Otherwise, the result is NULL.
CurrentLineNumber	INT	The current line number of the procedure or compound statement a connection is executing. The procedure can be identified using the CurrentProcedure property. If the line is part of a compound statement from the client, an empty string is returned.
LastStatement	LONG VARCHAR	The most recently prepared SQL statement for the current connection.
LastPlanText	LONG VARCHAR	The long text plan of the last query executed on the connection.
AppInfo	LONG VARCHAR	Information about the client that made the connection. For HTTP connections, this includes information about the browser. For connections using older versions of jConnect or Open Client, the information may be incomplete. The API value can be DBLIB, ODBC, OLEDB, or ADO.NET.
LockCount	INT	The number of locks held by the connection.
SnapshotCount	INT	The number of snapshots associated with the connection.

Remarks

The sa_performance_diagnostics system procedure returns a result set consisting of a set of request timing properties and statistics if the server has been told to collect the information. Recording of request timing information must be turned on the database server prior to calling sa_performance_diagnostics. To do this, specify the -zt option when starting the database server or execute the following:

```
CALL sa_server_option( 'RequestTiming','ON' );
```

Permissions

DBA authority required

Side effects

None

See also

- "-zt server option" [*SQL Anywhere Server - Database Administration*]
- "sa_performance_statistics system procedure" on page 918
- "sa_server_option system procedure" on page 935

Examples

You can execute the following query to identify connections that have spent a long time waiting for database server requests to complete.

```
SELECT  Number, Name,
      CAST( DATEDIFF( second, LoginTime, CURRENT TIMESTAMP ) AS DOUBLE ) AS
T,
      ReqTimeActive / T AS PercentActive
FROM  dbo.sa_performance_diagnostics()
WHERE PercentActive > 10.0
ORDER BY PercentActive DESC;
```

Find all requests that are currently executing, and have been executing for more than 60 seconds:

```
SELECT  Number, Name,
      CAST( DATEDIFF( second, LastReqTime, CURRENT TIMESTAMP ) AS DOUBLE ) AS
ReqTime
FROM  dbo.sa_performance_diagnostics()
WHERE ReqStatus <> 'IDLE' AND ReqTime > 60.0
ORDER BY ReqTime DESC;
```

sa_performance_statistics system procedure

Returns a summary of memory diagnostic statistics for all connections when the database server has request timing logging enabled.

Syntax

sa_performance_statistics( )

Result set

Column name	Data type	Description
DBNumber	INT	The ID number of the database.
ConnNumber	INT	An INTEGER representing a connection ID. Returns NULL if the Type is Server.
PropNum	INT	The connection property number.
PropName	VARCHAR(255)	The connection property name.

Column name	Data type	Description
Value	INT	The connection property value.

Remarks

The sa_performance_statistics system procedure returns a result set consisting of a set of memory diagnostic statistics if the server has been told to collect the information. Recording of memory diagnostic statistics must be turned on the database server prior to calling sa_performance_statistics. To do this, specify the -zt option when starting the database server or execute the following:

```
CALL sa_server_option( 'RequestTiming','ON' );
```

Permissions

DBA authority required

Side effects

None

See also

- “-zt server option” [*SQL Anywhere Server - Database Administration*]
- “sa_performance_diagnostics system procedure” on page 915
- “sa_server_option system procedure” on page 935

Example

The following example unloads all performance statistics to a text file named *dump_stats.txt*:

```
UNLOAD
   SELECT CURRENT TIMESTAMP, *
   FROM sa_performance_statistics()
   TO 'dump_stats.txt'
   APPEND ON;
```

sa_post_login_procedure system procedure

Determines whether a user's password is about to expire.

Syntax

sa_post_login_procedure( )

Arguments

None

Result set

The sa_post_login_procedure system procedure returns the following:

Column name	Data type	Description
message_text	VARCHAR(255)	If message_action is 1, message_text returns the message to display. If message_action is 0, message_text is NULL.
message_action	INTEGER	Whether the password is about to expire (1=yes, 0=no).

Remarks

The sa_post_login_procedure system procedure is the default setting for the post_login_procedure database option. See "post_login_procedure option [database]" [*SQL Anywhere Server - Database Administration*].

sa_post_login_procedure uses the user's password_life_time and password_grace_time login policy option values, and the current date and time, to determine whether a user's password is about to expire. If it is, the message to display to the user is returned in the result set.

Permissions

DBA authority required

Side effects

None

See also

- "Managing login policies overview" [*SQL Anywhere Server - Database Administration*]

sa_procedure_profile system procedure

Reports information about the execution time for each line within procedures, functions, events, or triggers that have been executed in a database. This procedure provides the same information as the **Profile** tab in Sybase Central.

Syntax

sa_procedure_profile(
 [*filename*
 [**,** *save_to_file*]]
)

Arguments

- **filename** Use this optional LONG VARCHAR(128) parameter to specify the file to which the profiling information should be saved, or from which file it should be loaded. See the Remarks section below for more about saving and loading the profiling information.
- **save_to_file** Use this optional INT(1) parameter to specify whether to save the profiling information to a file, or load it from a previously stored file.

Result set

Column name	Data type	Description
object_type	CHAR(1)	The type of object. See the Remarks section below for a list of possible object types.
object_name	CHAR(128)	The name of the stored procedure, function, event, or trigger. If the object_type is C or D, then this is the name of the foreign key for which the system trigger was defined.
owner_name	CHAR(128)	The object's owner.
table_name	CHAR(128)	The table associated with a trigger (the value is NULL for other object types).
line_num	UNSIGNED INTEGER	The line number within the procedure.
executions	UNSIGNED INTEGER	The number of times the line has been executed.
millisecs	UNSIGNED INTEGER	The time to execute the line, in milliseconds.
percentage	DOUBLE	The percentage of the total execution time required for the specific line.
foreign_owner	CHAR(128)	The database user who owns the foreign table for a system trigger.
foreign_table	CHAR(128)	The name of the foreign table for a system trigger.

Remarks

You can use this procedure to:

- **Return detailed procedure profiling information** To do this, you can simply call the procedure without specifying any arguments.
- **Save detailed procedure profiling information to file** To do this, you must include the *filename* argument and specify 1 for the *save_to_file* argument.
- **Load detailed procedure profiling information from a previously saved file** To do this, you must include the *filename* argument and specify 0 for the *save_to_file* argument (or leave it off, since the default is 0). When using the procedure in this way, the loaded file must have been created by the same database as the one from which you are running the procedure; otherwise, the results may be unusable.

Since the result set includes information about the execution times for individual lines within procedures, triggers, functions, and events, and what percentage of the total procedure execution time those lines use, you can use this profiling information to fine-tune slower procedures that may decrease performance.

Before you can profile your database, you must enable profiling. See "Enable procedure profiling" [*SQL Anywhere Server - SQL Usage*].

The object_type field of the result set can be:

- **P** stored procedure
- **F** function
- **E** event
- **T** trigger
- **C** ON UPDATE system trigger
- **D** ON DELETE system trigger

If you want summary information instead of line by line details for each execution, use the sa_procedure_profile_summary procedure instead.

Permissions

DBA authority required

Side effects

None

See also

- "sa_server_option system procedure" on page 935
- "sa_procedure_profile_summary system procedure" on page 923

Example

The following statement returns the execution time for each line of every procedure, function, event, or trigger that has been executed in the database:

```
CALL sa_procedure_profile( );
```

The following statement returns the same detailed procedure profiling information as the example above, and saves it to a file called *detailedinfo.txt*:

```
CALL sa_procedure_profile( "detailedinfo.txt", 1 );
```

Either of the following statements can be used to load detailed procedure profiling information from a file called *detailedinfoOLD.txt*:

```
CALL sa_procedure_profile( "detailedinfoOLD.txt", 0 );
```

```
CALL sa_procedure_profile( "detailedinfoOLD.txt" );
```

sa_procedure_profile_summary system procedure

Reports summary information about the execution times for all procedures, functions, events, or triggers that have been executed in a database. This procedure provides the same information for these objects as the **Profile** tab in Sybase Central.

Syntax

sa_procedure_profile_summary(
  [*filename*
  [**,** *save_to_file*]]
)

Arguments

- **filename** Use this optional LONG VARCHAR(128) parameter to specify the file to which the profiling information is saved, or from which file it should be loaded. See the Remarks section below for more about saving and loading the profiling information.
- **save_to_file** Use this optional INT(1) parameter to specify whether to save the summary information to a file, or to load it from a previously saved file.

Result set

Column name	Data type	Description
object_type	CHAR(1)	The type of object. See the Remarks section below for a list of possible object types.
object_name	CHAR(128)	The name of the stored procedure, function, event, or trigger.
owner_name	CHAR(128)	The object's owner.
table_name	CHAR(128)	The table associated with a trigger (the value is NULL for other object types).
executions	UNSIGNED INTEGER	The number of times each procedure has been executed.
millisecs	UNSIGNED INTEGER	The time to execute the procedure, in milliseconds.
foreign_owner	CHAR(128)	The database user who owns the foreign table for a system trigger.
foreign_table	CHAR(128)	The name of the foreign table for a system trigger.

Remarks

You can use this procedure to:

- **Return current summary information** To do this, you can simply call the procedure without specifying any arguments.

- **Save current summary information to file** To do this, you must include the *filename* argument and specify 1 for the *save_to_file* argument.
- **Load stored summary information from a file** To do this, you must include the *filename* argument and specify 0 for the *save_to_file* argument (or leave it off, since the default is 0). When using the procedure in this way, the loaded file must have been created by the same database as the one from which you are running the procedure; otherwise, the results may be unusable.

Since the procedure returns information about the usage frequency and efficiency of stored procedures, functions, events, and triggers, you can use this information to fine-tune slower procedures to improve database performance.

Before you can profile your database, you must enable profiling. See "Enable procedure profiling" [*SQL Anywhere Server - SQL Usage*].

The object_type field of the result set can be:

- **P** stored procedure
- **F** function
- **E** event
- **T** trigger
- **S** system trigger
- **C** ON UPDATE system trigger
- **D** ON DELETE system trigger

If you want line by line details for each execution instead of summary information, use the sa_procedure_profile procedure instead.

Permissions

DBA authority required

Side effects

None

See also

- "sa_server_option system procedure" on page 935
- "sa_procedure_profile system procedure" on page 920

Example

The following statement returns the execution time for any procedure, function, event, or trigger that has been executed in the database:

```
CALL sa_procedure_profile_summary( );
```

The following statement returns the same summary information as the previous example, and saves it to a file called *summaryinfo.txt*:

```
CALL sa_procedure_profile_summary( "summaryinfo.txt", 1 );
```

Either of the following statements can be used to load stored summary information from a file called *summaryinfoOLD.txt*:

```
CALL sa_procedure_profile_summary( "summaryinfoOLD".txt, 0 );
```

```
CALL sa_procedure_profile_summary( "summaryinfoOLD.txt" );
```

sa_recompile_views system procedure

Locates view definitions stored in the catalog that do not have column definitions and causes the column definitions to be created.

Syntax

sa_recompile_views([*ignore_errors*] **)**

Arguments

- **ignore_errors** Use this optional INTEGER parameter to specify whether to return errors during the recompilation. If you specify 0, an error is returned for each view for which column definition failed. If you specify 1, or any value other than 0, no errors are returned. If no value is specified, 0 is used by default.

Remarks

This procedure is used to locate views in the catalog that do not have column definitions and execute an ALTER VIEW statement with the RECOMPILE clause to create the column definitions. The procedure does this for each view that does not have a column definition until there are none left that require compilation or until any remaining column definitions cannot be created. If the procedure is unable to recompile any views, an error is reported. Errors can be suppressed by specifying a non-zero parameter to this procedure.

Caution
The sa_recompile_views system procedure should only be called from within a *reload.sql* script. This procedure is used by the Unload utility (dbunload) and should not be used explicitly.

The sa_recompile_views system procedure does not attempt to recompile materialized views or any view marked DISABLED.

Permissions

DBA authority required

Side effects

For each regular view that does not have a VALID status, an `ALTER VIEW` *`owner.viewname`* `ENABLE` statement is executed, causing an automatic commit.

See also

- “Regular view statuses” [*SQL Anywhere Server - SQL Usage*]
- “force_view_creation option [database]” [*SQL Anywhere Server - Database Administration*]
- “ALTER VIEW statement” on page 403

sa_refresh_text_indexes system procedure

Refreshes all text indexes defined as MANUAL REFRESH or AUTO REFRESH.

Syntax

sa_refresh_text_indexes( )

Remarks

The sa_refresh_text_indexes system procedure refreshes all text indexes defined as MANUAL REFRESH or AUTO REFRESH. It does not refresh text indexes defined as IMMEDIATE REFRESH (the default) because changes to those indexes are made when data is changed in the underlying table.

Permissions

DBA authority required

Side effects

Automatic commit

See also

- “Full text searching” [*SQL Anywhere Server - SQL Usage*]
- “Text configuration objects” [*SQL Anywhere Server - SQL Usage*]
- “DROP TEXT INDEX statement” on page 590
- “REFRESH TEXT INDEX statement” on page 706
- “TRUNCATE statement” on page 768
- “SYSTEXTIDX system view” on page 1035
- “sa_text_index_stats system procedure” on page 957
- “sa_text_index_vocab system procedure” on page 959

Example

The following statement refreshes all MANUAL and AUTO REFRESH text indexes in the database:

```
CALL sa_refresh_text_indexes( );
```

sa_refresh_materialized_views system procedure

Initializes all materialized views that are in an uninitialized state.

Syntax

sa_refresh_materialized_views([*ignore_errors*] **)**

Arguments

- **ignore_errors** Use this optional INTEGER parameter to specify whether to return errors during the recompilation. If you specify 0, an error is returned for each view for which column definition failed. If you specify 1, or any value other than 0, no errors are returned. If no value is specified, 0 is used by default.

Remarks

A materialized view may be in an uninitialized state because it has just been created, has just been re-enabled, or the last attempt to initialize or refresh it failed due to an error. The sa_refresh_materialized_views system procedure scans the database for all such materialized views and attempts to initialize them. If the procedure encounters an error initializing a materialized view, it continues on attempting to process the remaining uninitialized views.

You can also use the REFRESH MATERIALIZED VIEW statement to initialize a materialized view.

Permissions

DBA authority

Side effects

None

See also

- "REFRESH MATERIALIZED VIEW statement" on page 703
- "Refresh manual views" [*SQL Anywhere Server - SQL Usage*]

sa_remove_tracing_data system procedure

Permanently deletes from the diagnostic tracing tables all records pertaining to the specified logging (tracing) session ID.

Syntax

sa_remove_tracing_data(*log_session_id* **)**

Arguments

- **log_session_id** Use this INTEGER parameter to specify the ID of the logging session for which to remove the data.

Remarks

If there are no records for the specified *log_session_id*, the procedure has no effect. The procedure has no return values.

Permissions

DBA authority required

Side effects

Causes a commit upon completion, even if no records were found for the specified *log_session_id*.

See also

- "Diagnostic tracing tables" on page 810

sa_report_deadlocks system procedure

Retrieves information about deadlocks from an internal buffer created by the database server.

Syntax

sa_report_deadlocks( )

Result set

Column name	Data type	Description
snapshotId	BIGINT	The deadlock instance (all rows pertaining to a particular deadlock have the same ID).
snapshotAt	TIMESTAMP	The time when the deadlock occurred.
waiter	INT	The connection handle of the waiting connection.
who	VARCHAR(128)	The user ID associated with the connection that is waiting.
what	LONG VARCHAR	The command being executed by the waiting connection. This information is only available if you have turned on capturing of the most recently-prepared SQL statement by specifying the -zl option on the database server command line or have turned this feature on using the sa_server_option system procedure.
object_id	UNSIGNED BIGINT	The object ID of the table containing the row.
record_id	BIGINT	The row ID of the associated row.
owner	INT	The connection handle of the connection owning the lock being waited on.
is_victim	BIT	Identifies the rolled back transaction.
rollback_operation_count	UNSIGNED INT	The number of uncommitted operations that may be lost if the transaction rolls back.

Remarks

When the log_deadlocks option is set to On, the database server logs information about deadlocks in an internal buffer. You can view the information in the log using the sa_report_deadlocks system procedure.

Permissions

DBA authority required

Side effects

None

See also

- "Understanding system events" [*SQL Anywhere Server - Database Administration*]
- "log_deadlocks option [database]" [*SQL Anywhere Server - Database Administration*]
- "Determining who is blocked" [*SQL Anywhere Server - SQL Usage*]
- "-zl server option" [*SQL Anywhere Server - Database Administration*]
- "sa_server_option system procedure" on page 935

sa_reset_identity system procedure

Allows the next identity value to be set for a table. Use this to change the autoincrement value for the next row that will be inserted.

Syntax

sa_reset_identity(
tbl_name,
owner_name,
new_identity
)

Arguments

- **tbl_name** Use this CHAR(128) parameter to specify the table for which you want to reset the identity value. If the owner of the table is not specified, *tbl_name* must uniquely identify a table in the database.

- **owner_name** Use this CHAR(128) parameter to specify the owner of the table for which you want to reset the identity value.

- **new_identity** Use this BIGINT parameter to specify the value from which you want the auto-incrementing to start.

Remarks

The next identity value generated for a row inserted into the table is *new_identity* + 1.

No checking occurs to see whether *new_identity* + 1 conflicts with existing rows in the table. For example, if you specify *new_identity* as 100, the next row inserted gets an identity value of 101. However, if 101 already exists in the table, the row insertion fails.

If *owner* is not specified or is NULL, *tbl_name* must uniquely identify a table in the database.

Permissions

DBA authority required

Side effects

Causes a checkpoint to occur after the value has been updated

Example

The following statement resets the next identity value to 101:

```
CALL sa_reset_identity( 'Employees', 'DBA', 100 );
```

sa_rowgenerator system procedure

Returns a result set with rows between a specified start and end value.

Syntax

sa_rowgenerator(
[*rstart*
[, *rend*
[, *rstep*]]]
)

Arguments

- **rstart** Use this optional INTEGER parameter to specify the starting value. The default value is 0.
- **rend** Use this optional INTEGER parameter to specify the ending value. The default value is 100.
- **rstep** Use this optional INTEGER parameter to specify the increment by which the sequence values are increased. The default value is 1.

Result set

Column name	Data type	Description
row_num	INTEGER	Sequence number.

Remarks

The sa_rowgenerator procedure can be used in the FROM clause of a query to generate a sequence of numbers. This procedure is an alternative to using the RowGenerator system table. You can use sa_rowgenerator for such tasks as:

- generating test data for a known number of rows in a result set.
- generating a result set with rows for values in every range. For example, you can generate a row for every day of the month, or you can generate ranges of zip codes.
- generating a query that has a specified number of rows in the result set. This may be useful for testing the performance of queries.

You can emulate the behavior of the RowGenerator table with the following statement:

```
SELECT row_num FROM sa_rowgenerator( 1, 255 );
```

Permissions

None

Side effects

None

See also

- "RowGenerator table (dbo)" on page 827

Example

The following query returns a result set containing one row for each day of the current month.

```
SELECT DATEADD( day, row_num-1,
        YMD( DATEPART( year, CURRENT DATE ),
            DATEPART( month, CURRENT DATE ), 1 ) )
    AS day_of_month
    FROM sa_rowgenerator( 1, 31, 1 )
    WHERE DATEPART( month, day_of_month ) =
       DATEPART( month, CURRENT DATE )
    ORDER BY row_num;
```

The following query shows how many employees live in zip code ranges (0-9999), (10000-19999), ..., (90000-99999). Some of these ranges have no employees, which causes the warning `Null value eliminated in aggregate function (-109)`. The sa_rowgenerator procedure can be used to generate these ranges, even though no employees have a zip code in the range.

```
SELECT row_num AS r1, row_num+9999
 AS r2, COUNT( PostalCode ) AS zips_in_range
FROM sa_rowgenerator( 0, 99999, 10000 ) D LEFT JOIN Employees
   ON PostalCode BETWEEN r1 AND r2
GROUP BY r1, r2
ORDER BY 1;
```

The following example generates 10 rows of data and inserts them into the NewEmployees table:

```
INSERT INTO NewEmployees ( ID, Salary, Name )
SELECT row_num,
   CAST( RAND() * 1000 AS INTEGER ),
   'Mary'
FROM sa_rowgenerator( 1, 10 );
```

The following example uses the sa_rowgenerator system procedure to create a view containing all integers. The value 2147483647 in this example represents the maximum signed integer supported in SQL Anywhere.

```
CREATE VIEW Integers AS
SELECT row_num AS n
FROM sa_rowgenerator( 0, 2147483647, 1 );
```

This example uses the sa_rowgenerator system procedure to create a view containing dates from 0001-01-01 to 9999-12-31. The value 3652058 in this example represents the number of days between 0001-01-01 and 9999-12-31, the earliest and latest dates supported in SQL Anywhere.

```
CREATE VIEW Dates AS
SELECT DATEADD( day, row_num, '0001-01-01' ) AS d
FROM sa_rowgenerator( 0, 3652058, 1 );
```

sa_save_trace_data system procedure

Saves tracing data to base tables.

Syntax

sa_save_trace_data( )

Remarks

While a tracing session is running, diagnostic data is stored in temporary versions of the diagnostic tracing tables. When you stop a tracing session, you specify whether you want to permanently store the tracing data in the base tables for diagnostic tracing. If you do not choose to save the data, you can still save the data after the session is stopped by using the sa_save_trace_data system procedure.

The sa_save_trace_data system procedure returns an error if tracing is still in progress; you must stop tracing to use this system procedure.

The sa_save_trace_data system procedure can be used even if the user specified WITHOUT SAVING when stopping tracing. Also, the procedure must be called from the tracing database.

Permissions

DBA authority required

Side effects

Automatic commit.

See also

- "Create a diagnostic tracing session" [*SQL Anywhere Server - SQL Usage*]
- "Diagnostic tracing tables" on page 810

sa_send_udp system procedure

Sends a UDP packet to the specified address.

Syntax

sa_send_udp(
destAddress,
destPort,
msg
)

Arguments

- **destAddress** Use this CHAR(254) to specify either the host name or IP number.

- **destPort** Use this UNSIGNED SMALLINT parameter to specify the port number to use.
- **msg** Use this LONG BINARY parameter to specify the message to send to the specified address. If this value is a string, it must be enclosed in single quotes.

Remarks

This procedure sends a single UDP packet to the specified address. The procedure returns 0 if the message is sent successfully, and returns an error code if an error occurs. The error code is one of the following:

- -1 if the message is too large to send over a UDP socket (as determined by the operating system) or if there is a problem with the destination address
- the Winsock/Posix error code that is returned by the operating system

If the *msg* parameter contains binary data or is more complex than a string, you may want to use a variable. For example,

```
CREATE VARIABLE v LONG BINARY;
SET v='This is a UDP message';
SELECT dbo.sa_send_udp( '10.25.99.124', 1234, v );
DROP VARIABLE v;
```

This procedure can be used with MobiLink server-initiated synchronization to wake up the Listener utility (*dblsn.exe*). If you use the sa_send_udp system procedure as a way to notify the Listener, you should append a 1 to your UDP packet. This number is a server-initiated synchronization protocol number. In future versions of MobiLink, new protocol versions may cause the Listener to behave differently.

Permissions

DBA authority required

Side effects

None

Example

The following example sends the message "This is a test" to IP address 10.25.99.196 on port 2345:

```
CALL sa_send_udp( '10.25.99.196', 2345', 'This is a test' );
```

sa_server_messages system procedure

Allows you to return messages from the database server messages window as a result set.

Syntax

sa_server_messages([*first_msg*] [**,** *num_msgs*] **)**

Arguments

- **first_msg** Use this optional UNSIGNED BIGINT parameter to specify the ID of the first or last message to be returned, depending on the sign of the *num_msgs* parameter. The default is NULL, which means that the search starts at the beginning of the list if *num_msgs* is NULL or non-negative; the search starts past the end of the list if *num_msgs* is negative.

- **num_msgs** Use this optional BIGINT parameter to specify the number of messages to be returned. The sign indicates whether the request is for messages starting at *first_msg* or ending at *first_msg*. The default is NULL, which means that all messages starting at *first_msg* to the end of the list are returned.

Result set

Column name	Data type	Description
msg_id	UNSIGNED BIGINT	Unique message ID. Message IDs start at 0.
msg_text	LONG VARCHAR	Message text.
msg_time	TIMESTAMP	Time when the message was issued.
msg_severity	VARCHAR(255)	Message severity. This column contains one of the following values: ● **INFO** Informational message. ● **WARN** Warning. ● **ERR** Error.
msg_category	VARCHAR(255)	Message category. This column contains one of the following values: ● **STARTUP** Messages related to database server or database startup or shutdown. ● **CHKPT** Messages related to checkpoints. ● **MSG** Messages generated using the MESSAGE or PRINT statements. ● **DBA_MSG** Messages generated using the MESSAGE statement that would have required DBA permissions, such as messages sent to the event log. ● **CONN** Messages about database server connectivity. ● **OTHER** All other types of messages.
msg_database	VARCHAR(255)	Database name associated with the message if it applies to one specific database. Otherwise, NULL.

Remarks

When new messages are sent to the console, old messages with the same category or severity are deleted if the number of messages exceeds the value of the MessageCategoryLimit property. As a result, there may be gaps in the result set, and two consecutive rows may not have consecutive message IDs.

Permissions

None

Side effects

None

See also

- MessageCategoryLimit property: "Database server properties" [*SQL Anywhere Server - Database Administration*]

Example

The following command requests 100 messages starting at the message whose ID is 3:

```
CALL sa_server_messages( 3, 100 );
```

The following command requests 500 messages up to, and including, message 4032:

```
CALL sa_server_messages( 4032, -500 );
```

The following commands request all messages starting with message 3:

```
CALL sa_server_messages( 3, NULL );
```

```
CALL sa_server_messages( 3 );
```

The following command requests the first 100 messages in the list:

```
CALL sa_server_messages( NULL, 100 );
```

The following command requests the last 100 messages in the list:

```
CALL sa_server_messages( NULL, -100 );
```

The following commands request all the messages in the list:

```
CALL sa_server_messages( NULL, NULL );
```

```
CALL sa_server_messages( );
```

sa_server_option system procedure

Overrides a server option while the server is running.

Syntax

sa_server_option(
opt,

```
val
)
```

Arguments

- **opt** Use this CHAR(128) parameter to specify a server option name.
- **val** Use this CHAR(128) parameter to specify the new value for the server option.

Remarks

Database administrators can use this procedure to override some database server options temporarily, without restarting the database server.

The option values that are changed using this procedure are reset to their default values when the server shuts down. If you want to change an option value every time the server is started, you can specify the corresponding database server option when the database server is started if one exists (these are listed in the rightmost column in the table below).

The following option settings can be changed:

Option name	Values	Default	Server option
CacheSizingStatistics	YES, NO	NO	"-cs server option" [*SQL Anywhere Server - Database Administration*]
CollectStatistics	YES, NO	YES	"-k server option" [*SQL Anywhere Server - Database Administration*]
ConnsDisabled	YES, NO	NO	
ConnsDisabledForDB	YES, NO	NO	
ConsoleLogFile	*filename*		"-o server option" [*SQL Anywhere Server - Database Administration*]
ConsoleLogMaxSize	*file-size*, in bytes		"-on server option" [*SQL Anywhere Server - Database Administration*]
DatabaseCleaner	ON, OFF	ON	
DebuggingInformation	YES, NO	NO	"-z server option" [*SQL Anywhere Server - Database Administration*]

Option name	**Values**	**Default**	**Server option**
IdleTimeout	INTEGER, in minutes	240	"-ti server option" [*SQL Anywhere Server - Database Administration*]
LivenessTimeout	INTEGER, in seconds	120	"-tl server option" [*SQL Anywhere Server - Database Administration*]
MessageCategoryLimit	INTEGER	400	
OptionWatchAction	MESSAGE, ERROR	MESSAGE	"Monitoring option settings" [*SQL Anywhere Server - Database Administration*]
OptionWatchList	comma-separated list of database options		"Monitoring option settings" [*SQL Anywhere Server - Database Administration*]
ProcedureProfiling	YES, NO, RESET, CLEAR	NO	
ProfileFilterConn	*connection-id*		
ProfileFilterUser	*user-id*		
QuittingTime	valid date and time		"-tq server option" [*SQL Anywhere Server - Database Administration*]
RememberLastPlan	YES, NO	NO	"-zp server option" [*SQL Anywhere Server - Database Administration*]
RememberLastStatement	YES, NO	NO	"-zl server option" [*SQL Anywhere Server - Database Administration*]
RequestFilterConn	*connection-id*, -1		
RequestFilterDB	*database-id*, -1		

Option name	Values	Default	Server option
RequestLogFile	*filename*		"-zo server option" [*SQL Anywhere Server - Database Administration*]
RequestLogging	SQL, HOSTVARS, PLAN, PROCEDURES, TRIGGERS, OTHER, BLOCKS, REPLACE, ALL, YES, NONE, NO	NONE	"-zr server option" [*SQL Anywhere Server - Database Administration*]
RequestLogMaxSize	*file-size*, in bytes		"-zs server option" [*SQL Anywhere Server - Database Administration*]
RequestLogNumFiles	INTEGER		"-zn server option" [*SQL Anywhere Server - Database Administration*]
RequestTiming	YES, NO	NO	"-zt server option" [*SQL Anywhere Server - Database Administration*]
SecureFeatures	*feature-list*		"-sf server option" [*SQL Anywhere Server - Database Administration*]
WebClientLogFile	*filename*		"-zoc server option" [*SQL Anywhere Server - Database Administration*]
WebClientLogging	ON, OFF	OFF	

- **CacheSizingStatistics** When set to YES, display cache information in the database server messages window whenever the cache size changes. See "-cs server option" [*SQL Anywhere Server - Database Administration*].
- **CollectStatistics** When set to YES, the database server collects Performance Monitor statistics. See "-k server option" [*SQL Anywhere Server - Database Administration*].
- **ConnsDisabled** When set to YES, no other connections are allowed to any databases on the database server.
- **ConnsDisabledForDB** When set to YES, no other connections are allowed to the current database.
- **ConsoleLogFile** The name of the file used to record database server message log information. Specifying an empty string stops logging to the file. Any backslash characters in the path must be doubled

because this is a SQL string. See "-o server option" [*SQL Anywhere Server - Database Administration*].

- **ConsoleLogMaxSize** The maximum size, in bytes, of the file used to record database server message log information. When the database server message log file reaches the size specified by either this property or the -on server option, the file is renamed with the extension *.old* appended (replacing an existing file with the same name if one exists). The database server message log file is then restarted. See "-on server option" [*SQL Anywhere Server - Database Administration*].

- **DatabaseCleaner** Do not change the setting of this option except on the recommendation of iAnywhere Technical Support. See also "sa_clean_database system procedure" on page 845.

- **DebuggingInformation** Displays diagnostic messages and other messages for troubleshooting purposes. The messages appear in the database server messages window. See "-z server option" [*SQL Anywhere Server - Database Administration*].

- **IdleTimeout** Disconnects TCP/IP connections that have not submitted a request for the specified number of minutes. This prevents inactive connections from holding locks indefinitely. See "-ti server option" [*SQL Anywhere Server - Database Administration*].

- **LivenessTimeout** A liveness packet is sent periodically across a client/server TCP/IP network to confirm that a connection is intact. If the network server runs for a LivenessTimeout period without detecting a liveness packet, the communication is severed. See "-tl server option" [*SQL Anywhere Server - Database Administration*].

- **MessageCategoryLimit** Sets the minimum number of messages of each severity and category that can be retrieved using the sa_server_messages system procedure. See "sa_server_messages system procedure" on page 933.

- **OptionWatchAction** Specifies the action the database server should take when an attempt is made to set an option in the list. The supported values are MESSAGE and ERROR. When OptionWatchAction is set to MESSAGE, and an option specified by OptionWatchList is set, a message appears in the database server messages window indicating that the option being set is on the options watch list.

  When OptionWatchAction is set to ERROR, an error is returned indicating that the option cannot be set because it is on the options watch list.

  You can view the current setting for this property by executing the following query:

  ```
  SELECT DB_PROPERTY( 'OptionWatchAction' );
  ```

- **OptionWatchList** Specifies a comma-separated list of database options that you want to be notified about, or have the database server return an error for, when they are set. The string length is limited to 128 bytes. By default, it is an empty string. For example, the following command adds the automatic_timestamp, float_as_double, and tsql_hex_constant option to the list of options being watched:

  ```
  CALL dbo.sa_server_option( 'OptionWatchList','automatic_timestamp,
                                     float_as_double,tsql_hex_constant' )
  ```

  You can view the current setting for this property by executing the following query:

  ```
  SELECT DB_PROPERTY( 'OptionWatchList' );
  ```

- **ProcedureProfiling** Controls procedure profiling for stored procedures, functions, events, and triggers. Procedure profiling shows you how long it takes your stored procedures, functions, events, and triggers to execute. You can also set procedure profiling options on the **Database Properties** window in Sybase Central.

- ○ **YES** enables procedure profiling for the database you are currently connected to.
- ○ **NO** disables procedure profiling and leaves the profiling data available for viewing.
- ○ **RESET** returns the profiling counters to zero, without changing the YES or NO setting.
- ○ **CLEAR** returns the profiling counters to zero and disables procedure profiling.

  Once profiling is enabled, you can use the sa_procedure_profile_summary and sa_procedure_profile system procedures to retrieve profiling information from the database. See "Procedure profiling using system procedures" [*SQL Anywhere Server - SQL Usage*].

- **ProfileFilterConn** Instructs the database server to capture profiling information for a specific connection ID, without preventing other connections from using the database. When connection filtering is enabled, the value returned for `SELECT PROPERTY( 'ProfileFilterConn' )` is the connection ID of the connection being monitored. If no ID has been specified, or if connection filtering is disabled, the value returned is -1.
- **ProfileFilterUser** Instructs the database server to capture profiling information for a specific user ID.
- **QuittingTime** Instructs the database server to shut down at the specified time. See "-tq server option" [*SQL Anywhere Server - Database Administration*].
- **RememberLastPlan** Instructs the database server to capture the long text plan of the last query executed on the connection. This setting is also controlled by the -zp server option. See "-zp server option" [*SQL Anywhere Server - Database Administration*].

  When RememberLastPlan is turned on, you can obtain the textual representation of the plan of the last query executed on the connection by querying the value of the LastPlanText connection property:

```
SELECT CONNECTION_PROPERTY( 'LastPlanText' );
```

- **RememberLastStatement** Instructs the database server to capture the most recently prepared SQL statement for each database running on the server. For stored procedure calls, only the outermost procedure call appears, not the statements within the procedure.

  When RememberLastStatement is turned on, you can obtain the current value of the LastStatement for a connection by querying the value of the LastStatement connection property:

```
SELECT CONNECTION_PROPERTY( 'LastStatement' );
```

  When client statement caching is enabled, and a cached statement is reused, this property returns an empty string.

  For more information, see "Database server properties" [*SQL Anywhere Server - Database Administration*] and "-zl server option" [*SQL Anywhere Server - Database Administration*].

  When RememberLastStatement is turned on, the following statement returns the most recently-prepared statement for the specified connection:

```
SELECT CONNECTION_PROPERTY( 'LastStatement', connection-id );
```

  The sa_conn_activity system procedure returns this same information for all connections.

Caution
When -zl is specified, or when the RememberLastStatement server setting is turned on, any user can call the sa_conn_activity system procedure or obtain the value of the LastStatement connection property to find out the most recently-prepared SQL statement for any other user. This option should be used with caution and turned off when it is not required.

- **RequestFilterConn** Filter the request logging information so that only information for a particular connection is logged. This can help reduce the size of the request log file when monitoring a database server with many active connections or multiple databases. You can obtain the connection ID by executing the following:

  ```
  CALL sa_conn_info( );
  ```

  To specify a specific connection to be logged once you have obtained the connection ID, execute the following:

  ```
  CALL sa_server_option( 'RequestFilterConn', connection-id );
  ```

  Filtering remains in effect until it is explicitly reset, or until the database server is shut down. To reset filtering, use the following statement:

  ```
  CALL sa_server_option( 'RequestFilterConn', -1 );
  ```

- **RequestFilterDB** Filter the request logging information so that only information for a particular database is logged. This can help reduce the size of the request log file when monitoring a server with multiple databases. You can obtain the database ID by executing the following statement when you are connected to the desired database:

  ```
  SELECT CONNECTION_PROPERTY( 'DBNumber' );
  ```

  To specify that only information for a particular database is to be logged, execute the following:

  ```
  CALL sa_server_option( 'RequestFilterDB', database-id );
  ```

  Filtering remains in effect until it is explicitly reset, or until the database server is shut down. To reset filtering, use the following statement:

  ```
  CALL sa_server_option( 'RequestFilterDB', -1 );
  ```

- **RequestLogFile** The name of the file used to record request information. Specifying an empty string stops logging to the request log file. If request logging is enabled, but the request log file was not specified or has been set to an empty string, the server logs requests to the database server messages window. Any backslash characters in the path must be doubled as this is a SQL string. See "-zo server option" [*SQL Anywhere Server - Database Administration*].

- **RequestLogging** This call turns on logging of individual SQL statements sent to the database server for use in troubleshooting, in conjunction with the database server -zr and -zo options. Values can be combinations of the following, separated by either a plus sign (+), or a comma:

  - **PLAN** enables logging of execution plans (short form). Execution plans for procedures are also recorded if logging of procedures (PROCEDURES) is enabled.

  - **HOSTVARS** enables logging of host variable values. If you specify HOSTVARS, the information listed for SQL is also logged.

- ○ **PROCEDURES** enables logging of statements executed from within procedures.
- ○ **TRIGGERS** enables logging of statements executed from within triggers.
- ○ **OTHER** enables logging of additional request types not included by SQL, such as FETCH and PREFETCH. However, if you specify OTHER but do not specify SQL, it is the equivalent of specifying SQL+OTHER. Including OTHER can cause the log file to grow rapidly and could negatively impact server performance.
- ○ **BLOCKS** enables logging of details showing when a connection is blocked and unblocked on another connection.
- ○ **REPLACE** at the start of logging, the existing request log is replaced with a new (empty) one of the same name. Otherwise, the existing request log is opened and new entries are appended to the end of the file.
- ○ **ALL** logs all supported information. This is equivalent to specifying SQL+PLAN+HOSTVARS +PROCEDURES+TRIGGERS+OTHER+BLOCKS. This setting can cause the log file to grow rapidly and could negatively impact server performance.
- ○ **NO or NONE** turns off logging to the request log.

You can view the current setting for this property by executing the following query:

```
SELECT PROPERTY( 'RequestLogging' );
```

For more information, see "-zr server option" [*SQL Anywhere Server - Database Administration*], and "Database server properties" [*SQL Anywhere Server - Database Administration*].

- **RequestLogMaxSize** The maximum size of the file used to record request logging information, in bytes. If you specify 0, then there is no maximum size for the request logging file, and the file is never renamed. This is the default value.

  When the request log file reaches the size specified by either the sa_server_option system procedure or the -zs server option, the file is renamed with the extension *.old* appended (replacing an existing file with the same name if one exists). The request log file is then restarted. See "-zs server option" [*SQL Anywhere Server - Database Administration*].

- **RequestLogNumFiles** The number of request log file copies to retain.

  If request logging is enabled over a long period of time, the request log file can become large. The -zn option allows you to specify the number of request log file copies to retain. See "-zn server option" [*SQL Anywhere Server - Database Administration*].

- **RequestTiming** Instructs the database server to maintain timing information for each connection. This feature is turned off by default. When it is turned on, the database server maintains cumulative timers for each connection that indicate how much time the connection spent in the server in each of several states. You can use the sa_performance_diagnostics system procedure to obtain a summary of this timing information, or you can retrieve individual values by inspecting the following connection properties:

- ReqCountUnscheduled
- ReqTimeUnscheduled
- ReqCountActive
- ReqTimeActive
- ReqCountBlockIO
- ReqTimeBlockIO
- ReqCountBlockLock
- ReqTimeBlockLock
- ReqCountBlockContention
- ReqTimeBlockContention

See "Connection properties" [*SQL Anywhere Server - Database Administration*].

When the RequestTiming server property is on, there is a small overhead for each request to maintain the additional counters. See "-zt server option" [*SQL Anywhere Server - Database Administration*], and "sa_performance_diagnostics system procedure" on page 915.

- **SecureFeatures** Allows you to enable or disable secure features of a database server that is already running. *feature-list* is a comma-separated list of feature names or feature sets. By adding to the list of secure features, you are securing (preventing) a capability to do something. To remove items from the list of secure features, specify a minus sign (-) prior to the secure feature name. For a list of valid *feature-list* values, see "-sf server option" [*SQL Anywhere Server - Database Administration*].

  Any changes you make to enable or disable features take effect immediately for the connection. The settings do not affect the connection that executes the sa_server_option system procedure; you must disconnect and reconnect to see the change.

  Note
  To use the sa_server_option system procedure to enable or disable features, you must have specified a key with the -sk option when starting the database server, and set the value of the secure_feature_key database option to the key you specified for -sk (for example, `SET TEMPORARY OPTION secure_feature_key = 'j978kls12'`). Setting the secure_feature_key database option to the -sk value allows you to change the setting for secure features. See "-sk server option" [*SQL Anywhere Server - Database Administration*] and "secure_feature_key [database]" [*SQL Anywhere Server - Database Administration*].

  For example, to disable two features and enable a third, you would use this syntax:

  ```
  CALL sa_server_option('SecureFeatures', 'CONSOLE_LOG,WEBCLIENT_LOG,-
  REQUEST_LOG' );
  ```

  After executing this statement, CONSOLE_LOG, and WEBCLIENT_LOG are added to the list of secure features, and REQUEST_LOG is removed from the list.

- **WebClientLogFile** The name of the web service client log file. The web service client log file is truncated each time you use the -zoc server option or the WebClientLogFile property to set or reset the file name. Any backslash characters in the path must be doubled because this is a string. See "-zoc server option" [*SQL Anywhere Server - Database Administration*].

- **WebClientLogging** This option enables and disables logging of web service clients. The information that is logged includes HTTP requests and response data. Specify ON to start logging to the web service client log file, and specify OFF to stop logging to the file. See "-zoc server option" [*SQL Anywhere Server - Database Administration*].

Permissions

The following options, which are related to application profiling or request logging, require either DBA or PROFILE authority:

- ProcedureProfiling
- ProfileFilterConn
- ProfileFilterUser
- RequestFilterConn
- RequestFilterDB
- RequestLogFile
- RequestLogging
- RequestLogMaxSize
- RequestLogNumFiles

All other options require DBA authority.

Side effects

None

Example

The following statement disallows new connections to the database server:

```
CALL sa_server_option( 'ConnsDisabled', 'YES' );
```

The following statement disallows new connections to the current database:

```
CALL sa_server_option( 'ConnsDisabledForDB', 'YES' );
```

The following statement enables logging of all SQL statements, procedure calls, plans, blocking and unblocking events, and specifies that a new request log be started:

```
CALL dbo.sa_server_option( 'RequestLogging', 'SQL+PROCEDURES+BLOCKS+PLAN
+REPLACE' );
```

sa_set_http_header system procedure

Permits a web service to set an HTTP header in the result.

Syntax

sa_set_http_header(
fldname,

val
)

Arguments

- **fldname** Use this CHAR(128) parameter to specify a string containing the name of one of the HTTP header fields.
- **val** Use this LONG VARCHAR parameter to specify the value to which the named parameter should be set.

Remarks

Setting the special header field @HttpStatus sets the status code returned with the request. The status code is also known as the response code. For example, the following command sets the status code to 404 Not Found.

```
CALL dbo.sa_set_http_header( '@HttpStatus', '404' );
```

The body of the error message is inserted automatically. Only valid HTTP error codes can be used. Setting the status to an invalid code causes a SQL error.

With the exception of the Connection response header, response headers generated automatically by the database server can be removed. For example, the following command removes the Expires response header:

```
CALL dbo.sa_set_http_header( 'Expires', NULL );
```

Permissions

None

Side effects

None

See also

- “SQL Anywhere web services” [*SQL Anywhere Server - Programming*]
- “HTTP_HEADER function [HTTP]” on page 223
- “-xs server option” [*SQL Anywhere Server - Database Administration*]
- “sa_http_header_info system procedure” on page 883
- “sa_http_variable_info system procedure” on page 886
- “sa_set_http_option system procedure” on page 946
- “sa_split_list system procedure” on page 951
- “Web services functions” on page 130
- “Web services system procedures” on page 830

Example

The following example sets the Content-Type header field to text/html.

```
CALL dbo.sa_set_http_header( 'Content-Type', 'text/html' );
```

sa_set_http_option system procedure

Permits a web service to set an HTTP option in the result.

Syntax

sa_set_http_option(
optname,
val
)

Arguments

- **optname** Use this CHAR(128) parameter to specify a string containing the name of one of the HTTP options.
- **val** Use this LONG VARCHAR parameter to specify the value to which the named option should be set.

Remarks

Use this procedure within statements or procedures that handle web services to set options within an HTTP result set.

The supported options are:

- **CharsetConversion** Use this option to control whether the result set is to be automatically converted from the character set of the database to the character set of the client. The only permitted values are ON and OFF. The default value is ON. See "Using automatic character set conversion" [*SQL Anywhere Server - Programming*].
- **AcceptCharset** Use this option to specify the response character set used for XML, SOAP, and HTTP web services. The syntax for this option conforms to the syntax used for the HTTP Accept-Charset request-header field specification in the RFC2616 Hypertext Transfer Protocol.

  To specify a list of acceptable charsets, the HTTP client can include an Accept-Charset request header within the request sent to the database server. The Accept-Charset request header specifies which charset the database server should use for its response. When sending the response, a charset supported by both the client and database server is used.

  When the Accept-Charset request header is not specified, the charset most preferred by the client and supported by the database server is used. If the database server does not support any of the charsets specified by the client, or an Accept-Charset request header is not specified, the database charset is used.

  If the client does not send an Accept-Charset request header, the database server uses the first charset specified by the AcceptCharset option.

  If the client specifies an Accept-Charset request header and a charset list is also specified by an AcceptCharset HTTP option, the charset most preferred by the client and supported by the database server is used. If the database server does not support any of the charsets specified by the client, the first charset specified by the AcceptCharset HTTP option is used.

  A plus sign (+) in the AcceptCharset HTTP option instructs the database server to use the database charset. For example, `( 'AcceptCharset', '+,UTF-8,*')`. Whenever possible, the dbcharset is used, even if the client prefers a charset that is common to both the client and database server. Using the dbcharset for the response eliminates the need for translation and improves the response time of the database server.

An asterisk (*) in the AcceptCharset HTTP option instructs the database server to use a charset from the client Accept-Charset list. If a common charset does not exist, the first charset specified by the AcceptCharset option is used.

- **SessionId** Use this option to supply a name for an HTTP session. For example, `sa_set_http_option( 'SessionId', 'my_app_session_1' )` sets the ID for an HTTP session to my_app_session_1. SessionId must be a non-NULL string. For more information about HTTP sessions, see "Using HTTP sessions" [*SQL Anywhere Server - Programming*].
- **SessionTimeout** Use this option to specify the amount of time, in minutes, the HTTP session persists during inactivity. The SessionTimeout option is reset when an HTTP request using the session sends a response.

Permissions

None

Side effects

None

See also

- "Using HTTP sessions" [*SQL Anywhere Server - Programming*]
- "SQL Anywhere web services" [*SQL Anywhere Server - Programming*]
- "-xs server option" [*SQL Anywhere Server - Database Administration*]
- "sa_http_header_info system procedure" on page 883
- "sa_http_variable_info system procedure" on page 886
- "sa_set_http_header system procedure" on page 944
- "Web services functions" on page 130
- "Web services system procedures" on page 830

Examples

The following example specifies the use of the dbcharset when it can be accepted. If dbcharset cannot be accepted, UTF-8 is specified as an alternative. If UTF-8 cannot be used, the asterisk (*) specifies the use of the charset most preferred by the client and supported by the database server:

```
sa_set_http_option( ‘AcceptCharset’, ‘+,UTF-8,*’);
```

The following pseudo code illustrates how the AcceptCharset option is used:

```
if -- client sends an Accept-Charset header then
      if -- no AcceptCharset HTTP option set then
            if -- at least one of the client charsets are supported then
                 use most preferred client charset that is also supported by
the server
            else
                  use dbcharset
            end if;
      else -- an AcceptCharset HTTP option was set
               if -- at least one of the client charsets is also in the HTTP
```

```
option charset list
                     and is supported by the server then use most preferred
client charset that
                    is also in the HTTP option charset list and is supported
by the server(1)
              else
                     use the first charset listed in the AcceptCharset HTTP
option
           end if;
     end if
else -- client did not send an Accept-Charset header
     if -- no AcceptCharset HTTP option set then
           use dbcharset
        else -- an AcceptCharset HTTP option was set
                use the first charset listed in the AcceptCharset HTTP option
        end if
end if;
```

The following example sets the time out for an HTTP session to 5 minutes:

```
CALL sa_set_http_option('SessionTimeout', '5');
```

sa_set_soap_header system procedure

Permits the setting of SOAP headers for SOAP responses. This procedure is used within stored procedures called from SOAP web services.

Syntax

sa_set_soap_header(
fldname,
val
)

Arguments

- **fldname** Use this VARCHAR parameter to specify the header key, a unique string used to reference the given header entry (it need not be identical to the localname of the *val*).
- **val** Use this VARCHAR parameter to specify the raw XML of a top level header entry and its children within the scope of a SOAP Header element.

Remarks

All SOAP header entries set with this procedure are serialized within the SOAP Header element when the SOAP response message is sent. A *val* of NULL is not serialized. If no header entries exist for a SOAP response, then an enclosing Header element, within the SOAP envelope, is not created.

Permissions

None

Side effects

None

See also

- “Working with SOAP headers” [*SQL Anywhere Server - Programming*]
- “SQL Anywhere web services” [*SQL Anywhere Server - Programming*]
- “-xs server option” [*SQL Anywhere Server - Database Administration*]
- “Web services functions” on page 130
- “Web services system procedures” on page 830

Example

The following example sets the SOAP header welcome to Hello:

```
sa_set_soap_header( 'welcome', '<welcome>Hello</welcome>' )
```

sa_set_tracing_level system procedure

Initializes the level of tracing information to be stored in the diagnostic tracing tables.

Syntax

sa_set_tracing_level(
level
[, *specified_scope*
, *specified_name*]
[, *do_commit*]
)

Arguments

- **level** Use this INTEGER parameter to specify the level of diagnostic tracing to perform. Possible values include:
  - **0** Do not generate any tracing data. This level keeps the tracing session open, but does not send any tracing data to the diagnostic tracing tables.
  - **1** Sets a basic level of tracing.
  - **2** Sets a medium level of tracing.
  - **3** Sets a high level of tracing.
- **specified_scope** Use this optional LONG VARCHAR parameter to specify the tracing scope; for example, USER, DATABASE, CONNECTION_NAME, TRIGGER, and so on.
- **specified_name** Use this optional LONG VARCHAR parameter to specify the identifier for the object indicated in *specified_scope*.
- **do_commit** Use this optional TINYINT parameter to specify whether to commit, automatically, rows inserted by this procedure. Specify 1 (the default) to commit the rows automatically (recommended), and 0 to not commit them automatically.

Remarks

This procedure replaces the rows into the sa_diagnostic_tracing_level table, changing the tracing level and scope to the settings specified when calling the procedure.

Setting the level 0 does not stop the tracing session. Instead, the tracing session remains attached to the tracing database, but no tracing data is sent. The tracing session is still active when the level is 0.

This system procedure must be called from the database being profiled.

Permissions

DBA authority required

Side effects

None

See also

- "Choosing a diagnostic tracing level" [*SQL Anywhere Server - SQL Usage*]
- "Diagnostic tracing scopes" [*SQL Anywhere Server - SQL Usage*]
- "sa_diagnostic_tracing_level table" on page 824
- "Advanced application profiling using diagnostic tracing" [*SQL Anywhere Server - SQL Usage*]

Examples

The following example sets the tracing level to 1. This means that the entire database will be profiled for performance counter data, and some samples of executed statements:

```
CALL sa_set_tracing_level( 1 );
```

The following example sets the tracing level to 3, and specifies the user AG84756. This means that only activities associated with AG84756 will be traced:

```
CALL sa_set_tracing_level( 3, 'user', 'AG84756' );
```

sa_snapshots system procedure

Returns a list of snapshots that are currently active.

Syntax

sa_snapshots( )

Result set

Column name	Data type	Description
connection_num	INT	The connection ID for the connection on which the snapshot is running.
start_sequence_num	UNSIGNED BIGINT	A unique number that identifies the snapshot.
statement_level	BIT	True if the snapshot was created with statement-snapshot or readonly-statement-snapshot. Otherwise, false.

Remarks

Several statement snapshots can exist on one connection. In the case of nested or interleaved statements running under statement snapshot isolation levels, each one begins a different statement snapshot with its first read or update.

Usually there is only one transaction snapshot per connection (one entry per connection in sa_snapshots with statement_level=0). However, a snapshot associated with a cursor never changes after the cursor's first fetch and a cursor opened WITH HOLD stays open through a commit or rollback. If the cursor has an associated snapshot, then the snapshot also persists. Therefore, it is possible for multiple transaction snapshots to exist for the same connection_num: one for the current transaction snapshot and one or more for old transaction snapshots that persist because of WITH HOLD cursors.

Permissions

DBA authority required

Side effects

None

See also

- “sa_transactions system procedure” on page 961
- “Snapshot isolation” [*SQL Anywhere Server - SQL Usage*]

sa_split_list system procedure

Takes a string of values, separated by a delimiter, and returns a set of rows—one row for each value.

Syntax

sa_split_list(
str
[, *delim*]
[, *maxlen*]
)

Arguments

- **str** Use this LONG VARCHAR parameter to specify the string containing the values to be split, separated by *delim*.
- **delim** Use this optional CHAR(10) parameter to specify the delimiter used in *str* to separate values. The delimiter can be a string of any characters, up to 10 bytes. If *delim* is not specified, a comma is used by default.
- **maxlen** Use this optional INTEGER parameter to specify the maximum length of the returned values. For example, if *maxlen* is set to 3, the values in the result set are truncated to a length of 3 characters. If you specify 0 (the default), values can be any length.

Result set

Column name	Data type	Description
line_num	INTEGER	Sequential number for the row.
row_value	LONG VARCHAR	Value from the string, truncated to *maxlen* if required.

Remarks

The sa_split_list procedure accepts a string with a delimited list of values, and returns a result set with one value per row. This is the opposite of the action performed by the LIST function [Aggregate]. An empty string is returned for row_value if the string:

- begins with *delim*
- contains two successive instances of *delim* in the middle of the string
- ends with *delim*

White space within the input string is significant. If the delimiter is a space character, extra spaces in the input string result in extra rows in the result set. If the delimiter is not a space character, spaces in the input string are not trimmed from the values in the result set.

Permissions

None

Side effects

None

See also

- "LIST function [Aggregate]" on page 238

Examples

The following query returns a list of black colored products.

```
SELECT list( Name )
  FROM Products
  WHERE Color = 'Black';
```

list (Products.Name)
Tee Shirt,Baseball Cap,Visor,Shorts

In the following example, the sa_split_list procedure is used to return the original result set from the aggregated list.

```
SELECT *
  FROM sa_split_list( 'Tee Shirt,Baseball Cap,Visor,Shorts' );
```

line_num	row_value
1	Tee Shirt
2	Baseball Cap
3	Visor
4	Shorts

The following example returns a row for each word. To avoid returning rows where row_value is an empty string, the WHERE clause must be specified.

```
SELECT *
  FROM sa_split_list( 'one||three|four||six|', '|' )
  WHERE row_value <> '';
```

line_num	row_value
1	one
3	three
4	four
6	six

In the following example, a procedure called ProductsWithColor is created. When called, the ProductsWithColor procedure uses sa_split_list to parse the color values specified by the user, looks in the Color column of the Products table, and returns the name, description, size, and color for each product that matches one of the user-specified colors.

The result of the procedure call below is the name, description, size, and color of all products that are either white or black.

```
CREATE PROCEDURE ProductsWithColor( IN color_list LONG VARCHAR )
BEGIN
  SELECT Name, Description, Size, Color
  FROM Products
  WHERE Color IN ( SELECT row_value FROM sa_split_list( color_list ) )
END;
go

SELECT * from ProductsWithColor( 'white,black' );
```

sa_statement_text system procedure

Formats a SELECT statement so that individual items appear on separate lines. This is useful when viewing long statements from the request log, in which all newline characters are removed.

Syntax

sa_statement_text(*txt* **)**

Arguments

- **txt** Use this LONG VARCHAR parameter to specify a SELECT statement.

Remarks

The *txt* that is entered must be a string (in single quotes) or a string expression.

Permissions

None

Side effects

None

See also

- "sa_get_request_times system procedure" on page 879
- "sa_get_request_profile system procedure" on page 878

Example

The following call formats a SELECT statement so that individual items appear on separate lines.

```
CALL sa_statement_text( 'SELECT * FROM car WHERE name=''Audi''' );
```

	stmt_text
1	SELECT *
2	FROM car
3	WHERE name = 'Audi'

sa_table_fragmentation system procedure

Reports information about the fragmentation of database tables.

Syntax

sa_table_fragmentation(
 [*tbl_name*
 [**,** *owner_name*]]
)

Arguments

- **tbl_name** Use this optional CHAR(128) parameter to specify the name of the table to check for fragmentation.

- **owner_name** Use this optional CHAR(128) parameter to specify the owner of *tbl_name*.

Result set

Column name	Data type	Description
TableName	CHAR(128)	Name of the table.
rows	UNSIGNED INTEGER	Number of rows in the table.
row_segments	UNSIGNED BIGINT	Number of row segments in the table.
segs_per_row	DOUBLE	Number of segments per row.

Remarks

Database administrators can use this procedure to obtain information about the fragmentation in a database's tables. If no arguments are supplied, results are returned for all tables in the database.

When database tables become excessively fragmented, you can run REORGANIZE TABLE or rebuild the database to reduce table fragmentation and improve performance. See "Reducing table fragmentation" [*SQL Anywhere Server - SQL Usage*].

Permissions

DBA authority required

Side effects

None

See also

- "Reducing table fragmentation" [*SQL Anywhere Server - SQL Usage*]
- "Rebuilding databases" [*SQL Anywhere Server - SQL Usage*]
- "REORGANIZE TABLE statement" on page 712

sa_table_page_usage system procedure

Reports information about the page usage of database tables.

Syntax

sa_table_page_usage( )

Result set

Column name	Data type	Description
TableId	UNSIGNED INTEGER	The table ID.

Column name	Data type	Description
TablePages	INTEGER	The number of table pages used by the table.
PctUsedT	INTEGER	The percentage of used table page space.
IndexPages	INTEGER	The number of index pages used by the table.
PctUsedI	INTEGER	The percentage of used index page space.
PctOfFile	INTEGER	The percentage of the total database file the table occupies.
TableName	CHAR(128)	The table name.

Remarks

The results include the same information provided by the Information utility.

Permissions

DBA authority required

Side effects

None

See also

- "Information utility (dbinfo)" [*SQL Anywhere Server - Database Administration*]

sa_table_stats system procedure

Reports information about how many pages have been read from each table.

Syntax

sa_table_stats( )

Result set

Column name	Data type	Description
table_id	INT	The table ID.
creator	CHAR(128)	The user name of the table's creator.
table_name	CHAR(128)	The table name.
count	UNSIGNED BIGINT	The estimated number of rows in the table, taken from SYSTAB.

Column name	Data type	Description
table_page_count	UNSIGNED BIGINT	The number of main pages used by the table.
table_page_cached	UNSIGNED BIGINT	The number of tables pages currently stored in the cache.
table_page_reads	UNSIGNED BIGINT	The number of page reads performed for pages in the main table.
ext_page_count	UNSIGNED BIGINT	The estimated number of pages in the table
ext_page_cached	UNSIGNED BIGINT	Reserved for future use.
ext_page_reads	UNSIGNED BIGINT	Reserved for future use.

Remarks

Each row returned by the sa_table_stats procedure describes a table for which the optimizer is maintaining page statistics. The sa_table_stats procedure can be used to find which tables are using cache memory and how many disk reads are being performed for each table. For example, you can use the sa_table_stats procedure to find the table that is generating the most disk reads. The results of the procedure represent estimates and should be used only for diagnostic purposes.

The table_page_cached column indicates how many pages of the table are currently stored in the cache, and the table_page_reads column indicates how many table pages have been read from disk since the optimizer started maintaining counts for the table. These statistics are not stored persistently within the database; they represent the activity on tables after they are loaded into memory for the first time.

Permissions

DBA authority required

Side effects

None

See also

- “SYSTAB system view” on page 1028

sa_text_index_stats system procedure

Returns statistical information about the text indexes in the database.

Syntax

sa_text_index_stats( )

Remarks

Use the sa_text_index_stats system procedure to view statistical information for each text index in the database. The following table describes the information returned by sa_text_index_stats.

Column name	Type	Description
owner_id	UNSIGNED INT	ID of the owner of the table
table_id	UNSIGNED INT	ID of the table
index_id	UNSIGNED INT	ID of the text index
text_config_id	UNSIGNED BIGINT	ID of the text configuration object referenced by the index
owner_name	CHAR(128)	Name of the owner
table_name	CHAR(128)	Name of the table
index_name	CHAR(128)	Name of the text index
text_config_name	CHAR(128)	Name of the text configuration object
doc_count	UNSIGNED BIGINT	Total number of indexed column values in the text index
doc_length	UNSIGNED BIGINT	Total length of data in the text index
pending_length	UNSIGNED BIGINT	Total length of the pending changes
deleted_length	UNSIGNED BIGINT	Total length of the pending deletions
last_refresh	TIMESTAMP	Date and time of the last refresh

The pending_length, deleted_length, and last_refresh values are NULL for IMMEDIATE REFRESH text indexes.

For MANUAL REFRESH text indexes, you can use doc_length, pending_length, and deleted_length to decide whether to refresh the text index, and the type of refresh to perform (rebuild vs. incremental). See “REFRESH TEXT INDEX statement” on page 706.

Permissions

DBA authority required

Side effects

None

See also

- "Full text searching" [*SQL Anywhere Server - SQL Usage*]
- "Text indexes" [*SQL Anywhere Server - SQL Usage*]
- "DROP TEXT INDEX statement" on page 590
- "REFRESH TEXT INDEX statement" on page 706
- "TRUNCATE TEXT INDEX statement" on page 770
- "sa_refresh_text_indexes system procedure" on page 926
- "sa_text_index_vocab system procedure" on page 959
- "SYSTEXTIDX system view" on page 1035

Example

The following statement returns statistical information for each text index in the database:

```
CALL sa_text_index_stats( );
```

sa_text_index_vocab system procedure

Lists all terms that appear in a text index, and the total number of indexed values that each term appears in.

Syntax

sa_text_index_vocab(
'*text-index-name***',**
'*table-name***',**
'*table-owner***'**
)

Arguments

- **text-index-name** Use this CHAR(128) parameter to specify the name of the text index.
- **table-name** Use this CHAR(128) parameter to specify the name of the table on which the text index is built.
- **table-owner** Use this CHAR(128) parameter to specify the owner of the table.

Result set

Column name	Data type	Description
term	VARCHAR(60)	A term in the text index.
freq	BIGINT	The number of indexed values the term appears in.

Remarks

The sa_text_index_vocab system procedure returns all terms that appear in a text index, and the total number of indexed values that each term appears in (which is less than the total number of occurrences if the term appears multiple times in some indexed values).

The sa_text_index_vocab system procedure has the following limitations:

- It cannot be used with a CALL statement.
- It cannot be used in a statement within a procedure.
- Parameter values cannot be host variables or expressions. The arguments *text-index-name*, *table-name*, and *table-owner* must be constraints or variables.

Permissions

DBA authority, or SELECT permission on the indexed table is required.

Side effects

None

See also

- "Full text searching" [*SQL Anywhere Server - SQL Usage*]
- "Term and phrase searching" [*SQL Anywhere Server - SQL Usage*]
- "DROP TEXT INDEX statement" on page 590
- "REFRESH TEXT INDEX statement" on page 706
- "TRUNCATE TEXT INDEX statement" on page 770
- "sa_refresh_text_indexes system procedure" on page 926
- "SYSTEXTIDX system view" on page 1035

Example

The following example builds a text index called VocabTxtIdx on the Products.Description column in the sample database. The next statement executes the sa_text_index_vocab system procedure to return all the terms that appear in the text index.

```
CREATE TEXT INDEX VocabTxtIdx2 ON Products( Description );
SELECT *
   FROM sa_text_index_vocab( 'VocabTxtIdx2', 'Products', 'GROUPO' );
```

term	freq
Cap	2
Cloth	1
Cotton	2
Crew	1
Hooded	1
neck	2
...	...

sa_transactions system procedure

Returns a list of transactions that are currently active.

Syntax

sa_transactions( )

Result set

Column name	Data type	Description
connection_num	INT	The connection ID for the connection the transaction is running on.
transaction _id	INT	The ID that uniquely identifies the transaction as long as the database server keeps track of it. IDs are reused as old transaction information is discarded.
start_time	TIMESTAMP	The TIMESTAMP for when the transaction started.
start_sequence_num	UNSIGNED BIGINT	The start sequence number for the transaction.
end_sequence_num	UNSIGNED BIGINT	Then end sequence number for the transaction if it has been committed or rolled back, otherwise, NULL.
committed	bit	The state of the transaction: true if the transaction ended with a COMMIT, false if it ended with a ROLLBACK, and NULL if the transaction is still active.
version_entries	unsigned INT	The count of the number of row versions the transaction has saved.

Remarks

This procedure provides information about the transactions that are currently running against the database.

Permissions

DBA authority required

Side effects

None

See also

- “sa_snapshots system procedure” on page 950
- “Snapshot isolation” [*SQL Anywhere Server - SQL Usage*]

sa_unload_cost_model system procedure

Unloads the current cost model to the specified file.

Syntax

sa_unload_cost_model (*file_name* **)**

Arguments

- **file_name** Use this CHAR(256) parameter to specify the name of the file in which to unload the data. Because it is the database server that executes the system procedure, *file_name* specifies a file on the database server computer, and a relative *file_name* specifies a file relative to the database server's starting directory.

Remarks

The optimizer uses cost models to determine optimal access plans for queries. The database server maintains a cost model for each database. The cost model for a database can be recalibrated at any time using the CALIBRATE SERVER clause of the ALTER DATABASE statement. For example, you might decide to recalibrate the cost model if you move the database onto non-standard hardware.

The sa_unload_cost_model system procedure allows you save a cost model to an ASCII file (*file_name*). You can then log into another database and use the sa_load_cost_model system procedure to load the cost model from the first database into the second one. This avoids having to recalibrate the second database.

Note
The sa_unload_cost_model system procedure does not include CALIBRATE PARALLEL READ information in the file.

Using the sa_unload_cost_model system procedure eliminates repetitive, time-consuming recalibration activities when there is a large number of similar hardware installations.

Permissions

DBA authority required

You must have write permissions where the file is created.

Side effects

None

See also

- "ALTER DATABASE statement" on page 358
- "sa_load_cost_model system procedure" on page 892
- "Query optimization and execution" [*SQL Anywhere Server - SQL Usage*]

Example

The following example unloads the cost model to a file called costmodel8:

```
CALL sa_unload_cost_model( 'costmodel8' );
```

sa_external_library_unload system procedure

Unloads an external library.

Syntax

sa_external_library_unload(['*external-library*'] **)**

Arguments

- **external-library** Optionally use this LONG VARCHAR parameter to specify the name of a library to be unloaded. If no library is specified, all external libraries that are not in use are unloaded.

Remarks

If an external library is specified, but is in use or is not loaded, an error is returned. If no parameter is specified, an error is returned if no loaded external libraries are found.

Permissions

DBA authority required

Side effects

None

See also

- "Calling external libraries from procedures" [*SQL Anywhere Server - Programming*]

Example

The following example unloads an external library called myextlib.dll:

```
CALL sa_external_library_unload( 'myextlib.dll' );
```

The following example unloads all libraries that are not currently in use:

```
CALL sa_external_library_unload();
```

sa_validate system procedure

Validates all, or parts, of a database.

Syntax

sa_validate(
 [[*owner_name*.]*tbl_name* | *owner_name*]
)

Arguments

- **tbl_name** Use this optional VARCHAR(128) parameter to specify the name of a table or materialized view to validate.

- **owner_name** Use this optional VARCHAR(128) parameter to specify an owner. When specified by itself, all tables and materialized views owned by the owner are validated.

Permissions

DBA authority required

Side effects

None

Remarks

If you specify sa_validate() (no arguments), the database server validates all tables, materialized views, indexes, checksums, and the database file.

If neither *owner* nor *tbl_name* are specified, all tables and materialized views in the database are validated. Also, the database itself is validated, including checksum validation, and validation that the number of rows in the each table or materialized view matches the number of rows in each associated index.

The values for *tbl_name* and *owner_name* are strings and must be enclosed in quotes.

The procedure returns a single column named Messages. Errors returned during validation appear in the column. If validation succeeds without error, the column contains `No error detected`.

Caution
Validating a table or an entire database should be performed while no connections are making changes to the database; otherwise, errors may be reported indicating some form of database corruption even though no corruption actually exists.

Example

The following statement performs a validation of tables and materialized views owned by DBA:

```
CALL sa_validate( owner_name = 'DBA' );
```

sa_verify_password system procedure

Validates the password of the current user.

Syntax

sa_verify_password(*curr_pwsd* **)**

Arguments

- **curr_pwsd** Use this CHAR(128) parameter to specify the password of the current database user.

Remarks

This procedure is used by sp_password. If the password matches, the procedure simply returns. If it does not match, the error `Invalid user ID or password` is returned.

Permissions

None

Side effects

None

See also

- “Adaptive Server Enterprise system procedures” on page 834

sp_login_environment system procedure

Sets connection options when users log in.

Syntax

sp_login_environment( )

Remarks

sp_login_environment is the default procedure called by the login_procedure database option.

It is recommended that you do not edit this procedure. Instead, to change the login environment, set the login_procedure option to point to a different procedure.

Here is the text of the sp_login_environment procedure:

```
CREATE PROCEDURE dbo.sp_login_environment( )
BEGIN
  IF connection_property( 'CommProtocol' ) = 'TDS' THEN
    CALL dbo.sp_tsql_environment( )
  END IF
END;
```

Permissions

None

Side effects

None

See also

- “login_procedure option [database]” [*SQL Anywhere Server - Database Administration*]

sp_remote_columns system procedure

Produces a list of the columns in a remote table, and a description of their data types.

The server must be defined with the CREATE SERVER statement to use this system procedure.

Syntax

sp_remote_columns(
 @server_name,
 @table_name
 [**,** *@table_owner*
 [**,** *@table_qualifier*]]
)

Arguments

- **@server_name** Use this CHAR(128) parameter to specify a string containing the server name as specified by the CREATE SERVER statement.
- **@table_name** Use this CHAR(128) parameter to specify the name of the remote table.
- **@table_owner** Use this optional CHAR(128) parameter to specify the owner of *@table_name*.
- **@table_qualifier** Use this optional CHAR(128) parameter to specify the name of the database in which *@table_name* is located.

Result set

Column name	Data type	Description
database	CHAR(128)	The database name.
owner	CHAR(128)	The database owner name.
table-name	CHAR(128)	The table name.
column-name	CHAR(128)	The name of a column.
domain-id	SMALLINT	An INTEGER which indicates the data type of the column.
width	SMALLINT	The meaning of this field depends on the data type. For character types width represents the number of characters.
scale	SMALLINT	The meaning of this field depends on the data type. For NUMERIC data types scale is the number of digits after the decimal point.
nullable	SMALLINT	If null column values are allowed this field is 1. Otherwise nullable is 0.

Remarks

If you are entering a CREATE EXISTING statement and you are specifying a column list, it may be helpful to get a list of the columns that are available on a remote table. sp_remote_columns produces a list of the columns on a remote table and a description of their data types. If you specify a database, you must either specify an owner or provide the value NULL.

Standards and compatibility

- **Sybase** Supported by Open Client/Open Server.

Permissions

None

Side effects

None

See also

- "Accessing remote data" [*SQL Anywhere Server - SQL Usage*]
- "Server classes for remote data access" [*SQL Anywhere Server - SQL Usage*]
- "CREATE SERVER statement" on page 506

Example

The following example returns columns from the SYSOBJECTS table in the production database on an Adaptive Server Enterprise server named asetest. The owner is unspecified.

```
CALL sp_remote_columns( 'asetest', 'sysobjects', null, 'production' );
```

sp_remote_exported_keys system procedure

Provides information about tables with foreign keys on a specified primary table.

The server must be defined with the CREATE SERVER statement to use this system procedure.

Syntax

sp_remote_exported_keys(
 @server_name
 , *@sp_name*
 [, *@sp_owner*
 [, *@sp_qualifier*]]
)

Arguments

- **@server_name** Use this CHAR(128) parameter to specify identifies the server the primary table is located on. A value is required for this parameter.
- **@sp_name** Use this CHAR(128) parameter to specify the table containing the primary key. A value is required for this parameter.
- **@sp_owner** Use this optional CHAR(128) parameter to specify the primary table's owner.
- **@sp_qualifier** Use this optional CHAR(128) parameter to specify the database containing the primary table.

Result set

Column name	Data type	Description
pk_database	CHAR(128)	The database containing the primary key table.

Column name	Data type	Description
pk_owner	CHAR(128)	The owner of the primary key table.
pk_table	CHAR(128)	The primary key table.
pk_column	CHAR(128)	The name of the primary key column.
fk_database	CHAR(128)	The database containing the foreign key table.
fk_owner	CHAR(128)	The foreign key table's owner.
fk_table	CHAR(128)	The foreign key table.
fk_column	CHAR(128)	The name of the foreign key column.
key_seq	SMALLINT	The key sequence number.
fk_name	CHAR(128)	The foreign key name.
pk_name	CHAR(128)	The primary key name.

Remarks

This procedure provides information about the remote tables that have a foreign key on a particular primary table. The result set for the sp_remote_exported_keys system procedure includes the database, owner, table, column, and name for both the primary and the foreign key, and the foreign key sequence for the foreign key columns. The result set may vary because of the underlying ODBC and JDBC calls, but information about the table and column for a foreign key is always returned.

Permissions

None

Side effects

None

See also

- "CREATE SERVER statement" on page 506
- "Foreign keys" [*SQL Anywhere 11 - Introduction*]

Example

To get information about the remote tables with foreign keys on the SYSOBJECTS table, in the production database, on a server named asetest:

```
CALL sp_remote_exported_keys(
     @server_name='asetest',
     @sp_name='sysobjects',
     @sp_qualifier='production' );
```

sp_remote_imported_keys system procedure

Provides information about remote tables with primary keys that correspond to a specified foreign key.

The server must be defined with the CREATE SERVER statement to use this system procedure.

Syntax

sp_remote_imported_keys(
@server_name
, *@sp_name*
[, *@sp_owner*
[, *@sp_qualifier*]]
)

Arguments

- **@server_name** Use this optional CHAR(128) parameter to specify the server the foreign key table is located on. A value is required for this parameter.
- **@sp_name** Use this optional CHAR(128) parameter to specify the table containing the foreign key. A value is required for this parameter.
- **@sp_owner** Use this optional CHAR(128) parameter to specify the foreign key table's owner.
- **@sp_qualifier** Use this optional CHAR(128) parameter to specify the database containing the foreign key table.

Result set

Column name	Data type	Description
pk_database	CHAR(128)	The database containing the primary key table.
pk_owner	CHAR(128)	The owner of the primary key table.
pk_table	CHAR(128)	The primary key table.
pk_column	CHAR(128)	The name of the primary key column.
fk_database	CHAR(128)	The database containing the foreign key table.
fk_owner	CHAR(128)	The foreign key table's owner.
fk_table	CHAR(128)	The foreign key table.
fk_column	CHAR(128)	The name of the foreign key column.
key_seq	SMALLINT	The key sequence number.
fk_name	CHAR(128)	The foreign key name.
pk_name	CHAR(128)	The primary key name.

Remarks

Foreign keys reference a row in a separate table that contains the corresponding primary key. This procedure allows you to obtain a list of the remote tables with primary keys that correspond to a particular foreign table. The sp_remote_imported_keys result set includes the database, owner, table, column, and name for both the primary and the foreign key, and the foreign key sequence for the foreign key columns. The result set may vary because of the underlying ODBC and JDBC calls, but information about the table and column for a primary key is always returned.

Permissions

None

Side effects

None

See also

- "CREATE SERVER statement" on page 506
- "Foreign keys" [*SQL Anywhere 11 - Introduction*]

Example

To get information about the tables with primary keys that correspond to a foreign key on the SYSOBJECTS table in the asetest server:

```
CALL sp_remote_imported_keys(
     @server_name='asetest',
     @sp_name='sysobjects',
     @sp_qualifier='production' );
```

sp_remote_primary_keys system procedure

Provides primary key information about remote tables using remote data access.

Syntax

sp_remote_primary_keys(
 @server_name
 [**,** *@table_name*
 [**,** *@table_owner*
 [**,** *@table_qualifier*]]]
)

Arguments

- **@server_name** Use this CHAR(128) parameter to specify the server the remote table is located on.
- **@table_name** Use this optional CHAR(128) parameter to specify the remote table.
- **@table_owner** Use this optional CHAR(128) parameter to specify the owner of the remote table.
- **@table_qualifier** Use this optional CHAR(128) parameter to specify the name of the remote database.

Result set

Column name	Data type	Description
database	CHAR(128)	The name of the remote database.
owner	CHAR(128)	The owner of the remote table.
table-name	CHAR(128)	The remote table.
column-name	CHAR(128)	The column name.
key-seq	SMALLINT	The primary key sequence number.
pk-name	CHAR(128)	The primary key name.

Remarks

This system procedure provides primary key information about remote tables using remote data access.

Because of differences in the underlying ODBC/JDBC calls, the information returned differs slightly in terms of the catalog/database value depending upon the remote data access class that is specified for the server. However, the important information (for example, column name) is as expected.

Standards and compatibility

- **Sybase** Supported by Open Client/Open Server.

Permissions

None

Side effects

None

sp_remote_tables system procedure

Returns a list of the tables on a server.

The server must be defined with the CREATE SERVER statement to use this system procedure.

Syntax

sp_remote_tables(
@server_name
[, *@table_name*
[, *@table_owner*
[, *@table_qualifier*
[, *@with_table_type*]]]]
)

Arguments

- **@server_name** Use this CHAR(128) parameter to specify the server the remote table is located on.
- **@table_name** Use this CHAR(128) parameter to specify the remote table.
- **@table_owner** Use this CHAR(128) parameter to specify the owner of the remote table.
- **@table_qualifier** Use this CHAR(128) parameter to specify the database in which *table_name* is located.
- **@with_table_type** Use this optional BIT parameter to specify the type of remote table. This argument is a bit type and accepts two values, 0 (the default) and 1. You must enter the value 1 if you want the result set to include a column that lists table types.

Result set

Column name	Data type	Description
database	CHAR(128)	The name of the remote database.
owner	CHAR(128)	The name of the remote database owner.
table-name	CHAR(128)	The remote table.
table-type	CHAR(128)	Specifies the table type. The value of this field depends on the type of remote server. For example, TABLE, VIEW, SYS, and GBL TEMP are possible values.

Remarks

It may be helpful when you are configuring your database server to get a list of the remote tables available on a particular server. This procedure returns a list of the tables on a server.

The procedure accepts five parameters. If a table, owner, or database name is given, the list of tables will be limited to only those that match the arguments.

Standards and compatibility

- **Sybase** Supported by Open Client/Open Server.

Permissions

None

Side effects

None

See also

- "Accessing remote data" [*SQL Anywhere Server - SQL Usage*]
- "Server classes for remote data access" [*SQL Anywhere Server - SQL Usage*]
- "CREATE SERVER statement" on page 506

Examples

To get a list of all the Microsoft Excel worksheets available from an ODBC data source referenced by a server named excel:

```
CALL sp_remote_tables( 'excel' );
```

To get a list of all the tables owned by fred in the production database in an Adaptive Server Enterprise server named asetest:

```
CALL sp_remote_tables( 'asetest', null, 'fred', 'production' );
```

sp_servercaps system procedure

Displays information about a remote server's capabilities.

The server must be defined with the CREATE SERVER statement to use this system procedure.

Syntax

sp_servercaps(*@sname* **)**

Arguments

- **@sname** Use this CHAR(64) parameter to specify a server defined with the CREATE SERVER statement. The specified *@sname* must be the same server name used in the CREATE SERVER statement.

Remarks

This procedure displays information about a remote server's capabilities. SQL Anywhere uses this capability information to determine how much of a SQL statement can be forwarded to a remote server. The ISYSCAPABILITY system table, which lists the server capabilities, is not populated until after SQL Anywhere first connects to a remote server.

Standards and compatibility

- **Sybase** Supported by Open Client/Open Server.

Permissions

None

Side effects

None

See also

- “SYSCAPABILITY system view” on page 991
- “SYSCAPABILITYNAME system view” on page 992
- “Accessing remote data” [*SQL Anywhere Server - SQL Usage*]
- “Server classes for remote data access” [*SQL Anywhere Server - SQL Usage*]
- “CREATE SERVER statement” on page 506

Example

To display information about the remote server testasa:

```
CALL sp_servercaps( 'testasa' );
```

sp_tsql_environment system procedure

Sets connection options when users connect from jConnect or Open Client applications.

Syntax

sp_tsql_environment( )

Remarks

The sp_login_environment procedure is the default procedure specified by the login_procedure database option. For each new connection, the procedure specified by login_procedure is called. If the connection uses the TDS communications protocol (that is, if it is an Open Client or jConnect connection), then sp_login_environment in turn calls sp_tsql_environment.

This procedure sets database options so that they are compatible with default Sybase Adaptive Server Enterprise behavior.

If you want to change the default behavior, it is recommended that you create new procedures and alter your login_procedure option to point to these new procedures.

Permissions

None

Side effects

None

See also

- "sp_login_environment system procedure" on page 965
- "login_procedure option [database]" [*SQL Anywhere Server - Database Administration*]

Example

Here is the text of the sp_tsql_environment procedure:

```
CREATE PROCEDURE dbo.sp_tsql_environment()
BEGIN
    IF db_property( 'IQStore' ) = 'Off' THEN
        -- SQL Anywhere datastore
        SET TEMPORARY OPTION close_on_endtrans='OFF';
    END IF;
    SET TEMPORARY OPTION ansinull='OFF';
    SET TEMPORARY OPTION tsql_variables='ON';
    SET TEMPORARY OPTION ansi_blanks='ON';
    SET TEMPORARY OPTION chained='OFF';
    SET TEMPORARY OPTION quoted_identifier='OFF';
    SET TEMPORARY OPTION allow_nulls_by_default='OFF';
    SET TEMPORARY OPTION on_tsql_error='CONTINUE';
```

```
        SET TEMPORARY OPTION isolation_level='1';
        SET TEMPORARY OPTION date_format='YYYY-MM-DD';
        SET TEMPORARY OPTION timestamp_format='YYYY-MM-DD HH:NN:SS.SSS';
        SET TEMPORARY OPTION time_format='HH:NN:SS.SSS';
        SET TEMPORARY OPTION date_order='MDY';
        SET TEMPORARY OPTION escape_character='OFF';
    END
```

xp_cmdshell system procedure

Carries out an operating system command from a procedure.

Syntax

xp_cmdshell(
command
[, **'no_output'**] **)**

Arguments

- **command** Use this CHAR(8000) parameter to specify a system command.
- **'no_output'** Use this optional CHAR(254) parameter to specify whether to display output. The default behavior is to display output. If this parameter is the string 'no_output', no output is displayed.

Remarks

xp_cmdshell executes a system command and then returns control to the calling environment. The value returned by xp_cmdshell is the exit code from the executed shell process. The return value is 2 if an error occurs when the child process is started.

The second parameter affects only command line applications on Windows operating systems. For Unix, no output appears, regardless of the setting for the second parameter.

For Windows Mobile, any commands executed are visible in the database server message log, regardless of the setting for the second parameter. The console shell *\\windows\cmd.exe* is needed to run the procedure.

Permissions

DBA authority required

See also

- "CALL statement" on page 419

Example

The following statement lists the files in the current directory in the file *c:\temp.txt*:

```
xp_cmdshell( 'dir > c:\\temp.txt' )
```

The following statement carries out the same operation, but does so without displaying a **Command** window.

```
xp_cmdshell( 'dir > c:\\temp.txt', 'no_output' )
```

xp_msver system procedure

Retrieves version and name information about the database server.

Syntax

xp_msver(*string* **)**

- ***string*** The string must be one of the following, enclosed in string delimiters.

Argument	Description
ProductName	The name of the product (SQL Anywhere).
ProductVersion	The version number, followed by the build number. The format is as follows: `11.0.0.3512`
CompanyName	Returns the following string: `iAnywhere Solutions, Inc.`
FileDescription	Returns the name of the product, followed by the name of the operating system.
LegalCopyright	Returns a copyright string for the software.
LegalTrademarks	Returns trademark information for the software.

Remarks

xp_msver returns product, company, version, and other information.

Permissions

None

See also

- "System functions" on page 133

Example

The following statement requests the version and operating system description:

```
SELECT xp_msver( 'ProductVersion') Version,
   xp_msver( 'FileDescription' ) Description;
```

Sample output is as follows. The value for Version will likely be different on your system.

Version	Description
11.0.0.3512	SQL Anywhere Windows XP

xp_read_file system procedure

Reads a file and returns the contents of the file as a LONG BINARY variable.

Syntax

xp_read_file(*filename* **)**

Arguments

- **filename** Use this LONG VARCHAR parameter to specify the name of the file for which to return the contents.

Remarks

The function reads the contents of the named file, and returns the result as a LONG BINARY value.

The *filename* is relative to the starting directory of the database server.

The function can be useful for inserting entire documents or images stored in files into tables. If the file cannot be read, the function returns NULL.

If the data file is in a different character set, you can use the CSCONVERT function to convert it. See "CSCONVERT function [String]" on page 168.

You can also use the CSCONVERT function to address character set conversion requirements you have when using the xp_read_file system procedure. See "CSCONVERT function [String]" on page 168.

Permissions

DBA authority required

See also

- "CSCONVERT function [String]" on page 168
- "xp_write_file system procedure" on page 986
- "CALL statement" on page 419
- "Using openxml with xp_read_file" [*SQL Anywhere Server - SQL Usage*]

Example

The following statement inserts an image into a column named picture of the table t1 (assuming all other columns can accept NULL):

```
INSERT INTO t1 ( picture )
   SELECT xp_read_file( 'portrait.gif' );
```

xp_scanf system procedure

Extracts substrings from an input string and a format string.

Syntax

xp_scanf(
input_buffer,
format,
parm [**,** *parm2*, ...]
)

Arguments

- **input_buffer** Use this CHAR(254) parameter to specify the input string.
- **format** Use this CHAR(254) parameter to specify the format of the input string, using placeholders (%s) for each *parm* argument. There can be up to fifty placeholders in the *format* argument, and there must be the same number of placeholders as *parm* arguments.
- **parm** Use one or more of these CHAR(254) parameters to specify the substrings extracted from *input_buffer*. There can be up to 50 of these parameters.

Remarks

The xp_scanf system procedure extracts substrings from an input string using the specified *format*, and puts the results in the specified *parm* values.

Permissions

None

See also

- "CALL statement" on page 419

Example

The following statements extract the substrings Hello and World! from the input buffer Hello World!, and put them into variables string1 and string2, and then selects them:

```
CREATE VARIABLE string1 CHAR(254);
CREATE VARIABLE string2 CHAR(254);
CALL xp_scanf( 'Hello World!', '%s %s', string1, string2 );
SELECT string1, string2;
```

xp_sendmail system procedure

Sends an email message.

Syntax

xp_sendmail(
recipient = *mail-address*
[**, subject** = *subject*]
[**, cc_recipient** = *mail-address*]
[**, bcc_recipient** = *mail-address*]
[**, query** = *sql-query*]
[**, "message"** = *message-body*]
[**, attachname** = *attach-name*]

[, **attach_result** = *attach-result*]
[, **echo_error** = *echo-error*]
[, **include_file** = *filename*]
[, **no_column_header** = *no-column-header*]
[, **no_output** = *no-output*]
[, **width** = *width*]
[, **separator** = *separator-char*]
[, **dbuser** = *user-name*]
[, **dbname** = *db-name*]
[, **type** = *type*]
[, **include_query** = *include-query*]
[, **content_type** = *content-type*]
)

Arguments

Some arguments supply fixed values and are available for use to ensure Transact-SQL compatibility, as noted below.

- **recipient** This LONG VARCHAR parameter specifies the recipient mail address. When specifying multiple recipients, each mail address must be separated by a semicolon.
- **subject** This LONG VARCHAR parameter specifies the subject field of the message. The default is NULL.
- **cc_recipient** This LONG VARCHAR parameter specifies the cc recipient mail address. When specifying multiple cc recipients, each mail address must be separated by a semicolon. The default is NULL.
- **bcc_recipient** This LONG VARCHAR parameter specifies the bcc recipient mail address. When specifying multiple bcc recipients, each mail address must be separated by a semicolon. The default is NULL.
- **query** This LONG VARCHAR is for use with Transact-SQL. The default is NULL.
- **"message"** This LONG VARCHAR parameter specifies the message contents. The default is NULL. The "message" parameter name requires double quotes around it because MESSAGE is a reserved word. See “Reserved words” on page 4.
- **attachname** This LONG VARCHAR parameter is for use with Transact-SQL. The default is NULL.
- **attach_result** This INT parameter is for use with Transact-SQL. The default is 0.
- **echo_error** This INT parameter is for use with Transact-SQL. The default is 1.
- **include_file** This LONG VARCHAR parameter specifies an attachment file. The default is NULL.
- **no_column_header** This INT parameter is for use with Transact-SQL. The default is 0.
- **no_output** This INT parameter is for use with Transact-SQL. The default is 0.
- **width** This INT parameter is for use with Transact-SQL. The default is 80.
- **separator** This CHAR(1) parameter is for use with Transact-SQL. The default is CHAR(9).
- **dbuser** This LONG VARCHAR parameter is for use with Transact-SQL. The default is guest.
- **dbname** This LONG VARCHAR parameter is for use with Transact-SQL. The default is master.

- **type** This LONG VARCHAR parameter is for use with Transact-SQL. The default is NULL.
- **include_query** This INT parameter is for use with Transact-SQL. The default is 0.
- **content_type** This LONG VARCHAR parameter specifies the content type for the "message" parameter (for example, text/html, ASIS, and so on). The default is NULL. The value of content_type is not validated; setting an invalid content type results in an invalid or incomprehensible email being sent.

  If you want to set headers manually, you can set content_type parameter to ASIS. When you do this, the xp_sendmail procedure assumes that the data passed to the message parameter is a properly formed email with headers, and does not add any additional headers. When specifying ASIS, you must set all the headers manually in the message parameter, even headers that would normally be filled in by passing data to the other parameters.

Permissions

DBA authority required

Must have executed xp_startmail to start an email session using MAPI, or xp_startsmtp to start an email session using SMTP.

If you are sending mail using MAPI, the content_type parameter is not supported.

Remarks

xp_sendmail is a system procedure that sends an email message to the specified recipients once a session has been started with xp_startmail or xp_startsmtp. The procedure accepts messages of any length. The argument values for xp_sendmail are strings. The length of each argument is limited to the amount of available memory on your system.

The content_type argument is intended for users who understand the requirements of MIME email. xp_sendmail accepts ASIS as a content_type. When content_type is set to ASIS, xp_sendmail assumes that the message body ("message") is a properly formed email with headers, and does not add any additional headers. Specify ASIS to send multipart messages containing more than one content type. For more information about MIME, see RFCs 2045-2049 (http://www.ietf.org/).

Attachments specified by the include_file parameter are sent as application/octet-stream MIME type, with base64 encoding, and must be present on the database server.

In SQL Anywhere 10.0.0 and later, email sent with an SMTP email system is encoded if the subject line contains characters that are not 7-bit ASCII. Also, email sent to an SMS-capable device may not be decoded properly if the subject line contains characters that are not 7-bit ASCII.

Return codes

See “Return codes for MAPI and SMTP system procedures” on page 831.

See also

- “MAPI and SMTP procedures” on page 830
- “xp_startmail system procedure” on page 983
- “xp_startsmtp system procedure” on page 984
- “xp_stopmail system procedure” on page 985
- “xp_stopsmtp system procedure” on page 986
- “CALL statement” on page 419

Example

The following call sends a message to the user ID Sales Group containing the file *prices.doc* as a mail attachment:

```
CALL xp_sendmail( recipient='Sales Group',
      subject='New Pricing',
      include_file = 'C:\\DOCS\\PRICES.DOC' );
```

The following sample program shows various uses of the xp_sendmail system procedure, as described in the example itself:

```
BEGIN
DECLARE to_list LONG VARCHAR;
DECLARE email_subject CHAR(256);
DECLARE content LONG VARCHAR;
DECLARE uid CHAR(20);

SET to_list='test_account@mytestdomain.com';
SET email_subject='This is a test';
SET uid='test_sender@mytestdomain.com';

// Call xp_startsmtp to start an SMTP email session
CALL xp_startsmtp( uid, 'mymailserver.mytestdomain.com' );

// Basic email example
SET content='This text is the body of my email.\n';
CALL xp_sendmail( recipient=to_list,
     subject=email_subject,
     "message"=content );

// Send email containing HTML using the content_type parameter,
// as well as including an attachment with the include_file
// parameter
SET content='Plain text.<BR><BR><B>Bold text.</B><BR><BR><a
href="www.iAnywhere.com">iAnywhere
 Home Page</a></B><BR><BR>';
CALL xp_sendmail( recipient=to_list,
     subject=email_subject,
     "message"=content,
     content_type = 'text/html',
     include_file = 'test.zip' );

// Send email "ASIS".  Here the content-type has been specified
// by the user as part of email body.  Note the attachment can
// also be done separately
SET content='Content-Type: text/html;\nContent-Disposition: inline; \n\nThis
text
 is not bold<BR><BR><B>This text is bold</B><BR><BR><a
href="www.iAnywhere.com">iAnywhere Home
 Page</a></B><BR><BR>';
CALL xp_sendmail( recipient=to_list,
          subject=email_subject,
          "message"=content,
          content_type = 'ASIS',
          include_file = 'test.zip' );

     // Send email "ASIS" along with an include file.  Note that
     // "message" contains the information for another attachment
     SET content = 'Content-Type: multipart/mixed; boundary="xxxxx";\n';
     SET content = content || 'This part of the email should not be shown.  If this
     is shown
      then the email client is not MIME compatible\n\n';
     SET content = content || '--xxxxx\n';
     SET content = content || 'Content-Type: text/html;\n';
     SET content = content || 'Content-Disposition: inline;\n\n';
     SET content = content || 'This text is not bold<BR><BR><B>This text is bold</
     B><BR>
      <BR><a href="www.iAnywhere.com">iAnywhere Home Page</a></B><BR><BR>\n\n';
     SET content = content || '--xxxxx\n';
     SET content = content || 'Content-Type: application/zip; name="test.zip"\n';
     SET content = content || 'Content-Transfer-Encoding: base64\n';
     SET content = content || 'Content-Disposition: attachment;
     filename="test.zip"\n\n';

     // Encode the attachment yourself instead of adding this one in
     // the include_file parameter
     SET content = content || base64_encode( xp_read_file( 'othertest.zip' ) ) ||
     '\n\n';
     SET content = content || '--xxxxx--\n';
     CALL xp_sendmail( recipient=to_list,
           subject=email_subject,
           "message"=content,
           content_type = 'ASIS',
           include_file = 'othertest.zip' );

     // End the SMTP session
         CALL xp_stopsmtp();
     END
```

xp_sprintf system procedure

Builds a result string from a set of input strings.

Syntax

xp_sprintf(
 output_buffer,
 format,
 parm [**,** *parm2*, ...]
)

Arguments

- **output_buffer** Use this CHAR(254) parameter to specify the output buffer containing the result string.
- **format** Use this CHAR(254) parameter to specify how to format the result string, using placeholders (%s) for each *parm* argument. There can be up to fifty placeholders in the *format* argument, and there should be the same number of placeholders as *parm* arguments.

- **parm** These are the input strings that are used in the result string. You can specify up to 50 of these CHAR(254) arguments.

Remarks

The xp_sprintf system procedure builds up a string using the *format* argument and the *parm* argument(s), and puts the results in *output_buffer*.

Permissions

None

See also

- "CALL statement" on page 419

Example

The following statements put the string Hello World! into the result variable.

```
CREATE VARIABLE result CHAR(254);
Call xp_sprintf( result, '%s %s', 'Hello', 'World!' );
```

xp_startmail system procedure

Starts an email session under MAPI.

Syntax

xp_startmail(
 [**mail_user** = *mail-login-name*]
 [**, mail_password** = *mail-password*] **)**

Arguments

- **mail_user** Use this LONG VARCHAR parameter to specify the MAPI login name.
- **mail_password** Use this LONG VARCHAR parameter to specify the MAPI password.

Permissions

DBA authority required

Not supported on Unix.

Remarks

xp_startmail is a system procedure that starts an email session.

If you are using Microsoft Exchange, the *mail-login-name* argument is an Exchange profile name, and you should not include a password in the procedure call.

Return codes

See "Return codes for MAPI and SMTP system procedures" on page 831.

See also

- "MAPI and SMTP procedures" on page 830
- "xp_stopmail system procedure" on page 985
- "xp_sendmail system procedure" on page 978
- "xp_startsmtp system procedure" on page 984
- "xp_stopsmtp system procedure" on page 986
- "CALL statement" on page 419

xp_startsmtp system procedure

Starts an email session under SMTP.

Syntax

xp_startsmtp(
smtp_sender = *email-address*
, smtp_server = *smtp-server*
[**, smtp_port =** *port-number*]
[**, timeout =** *timeout*]
[**, smtp_sender_name =** *username*]
[**, smtp_auth_username =** *auth-username*
[**, smtp_auth_password =** *auth-password*
)

Arguments

- **smtp_sender** This LONG VARCHAR parameter specifies the email address of the sender.
- **smtp_server** This LONG VARCHAR parameter specifies which SMTP server to use, and is the server name or IP address.
- **smtp_port** This optional INTEGER parameter specifies the port number to connect to on the SMTP server. The default is 25.
- **timeout** This optional INTEGER parameter specifies how long to wait, in seconds, for a response from the database server before aborting the current call to xp_sendmail. The default is 60 seconds.
- **smtp_sender_name** This optional LONG VARCHAR parameter specifies an alias for the sender's email address. For example, 'JSmith' instead of '*email-address*'.
- **smtp_auth_username** This optional LONG VARCHAR parameter specifies the user name to provide to SMTP servers requiring authentication.
- **smtp_auth_password** This optional LONG VARCHAR parameter specifies the user name to provide to SMTP servers requiring authentication.

Permissions

DBA authority required

Remarks

xp_startsmtp is a system procedure that starts a mail session for a specified email address by connecting to an SMTP server. This connection can time out. Therefore, it is recommended that you call xp_start_smtp just before executing xp_sendmail.

Virus scanners can affect xp_startsmtp, causing it to return error code 100. For McAfee VirusScan version 8.0.0 and later, settings for preventing mass mailing of email worms also prevent xp_sendmail from executing properly. If your virus scanning software allows you to specify processes that can bypass the mass mailing protections, specify *dbeng11.exe* and *dbsrv11.exe*. For example, in the case of McAfee VirusScan, you can add these two processes to the list of Excluded Processes in the Properties area for preventing mass mailing.

Return codes

See "Return codes for MAPI and SMTP system procedures" on page 831.

See also

- "MAPI and SMTP procedures" on page 830
- "xp_startmail system procedure" on page 983
- "xp_stopmail system procedure" on page 985
- "xp_sendmail system procedure" on page 978
- "xp_stopsmtp system procedure" on page 986
- "CALL statement" on page 419

xp_stopmail system procedure

Closes a MAPI email session.

Syntax

xp_stopmail( )

Permissions

DBA authority required

Not supported on Unix.

Remarks

xp_stopmail is a system procedure that ends an email session.

Return codes

See "Return codes for MAPI and SMTP system procedures" on page 831.

See also

- “MAPI and SMTP procedures” on page 830
- “xp_startmail system procedure” on page 983
- “xp_sendmail system procedure” on page 978
- “xp_startsmtp system procedure” on page 984
- “xp_stopsmtp system procedure” on page 986
- “CALL statement” on page 419

xp_stopsmtp system procedure

Closes an SMTP email session.

Syntax

xp_stopsmtp( )

Permissions

DBA authority required

Remarks

xp_stopsmtp is a system procedure that ends an email session.

Return codes

See “Return codes for MAPI and SMTP system procedures” on page 831.

See also

- “MAPI and SMTP procedures” on page 830
- “xp_startmail system procedure” on page 983
- “xp_stopmail system procedure” on page 985
- “xp_sendmail system procedure” on page 978
- “xp_startsmtp system procedure” on page 984
- “CALL statement” on page 419

xp_write_file system procedure

Writes data to a file from a SQL statement.

Syntax

xp_write_file(
 filename,
 file_contents
)

Arguments

- **filename** Use this LONG VARCHAR parameter to specify the file name.

- **file_contents** Use this LONG BINARY parameter to specify the contents to write to the file.

Remarks

The function writes *file_contents* to the file *filename*. It returns 0 if successful, and non-zero if it fails.

The *filename* value can be prefixed by either an absolute or a relative path. If *filename* is prefixed by a relative path, then the file name is relative to the current working directory of the database server. If the file already exists, its contents are overwritten.

This function can be useful for unloading long binary data into files.

You can also use the CSCONVERT function to address character set conversion requirements you have when using the xp_write_file system procedure. See "CSCONVERT function [String]" on page 168.

Permissions

DBA authority required

See also

- "CSCONVERT function [String]" on page 168
- "xp_read_file system procedure" on page 977
- "CALL statement" on page 419

Examples

This example uses xp_write_file to create a file *accountnum.txt* containing the data 123456:

```
CALL xp_write_file( 'accountnum.txt', '123456' );
```

This example queries the Contacts table of the sample database, and then creates a text file for each contact living in New Jersey. Each text file is named using a concatenation of the contact's first name (GivenName), last name (Surname), and then the string .txt (for example, *Reeves_Scott.txt*), and contains the contact's street address (Street), city (City), and state (State), on separate lines.

```
SELECT xp_write_file(
Surname || '_' ||  GivenName || '.txt',
Street || '\n' || City || '\n' || State )
FROM Contacts WHERE State = 'NJ';
```

This example uses xp_write_file to create an image file (JPG) for every product in the Products table. Each value of the ID column becomes a file name for a file with the contents of the corresponding value of the Photo column:

```
SELECT xp_write_file( ID || '.jpg' , Photo ) FROM Products;
```

In the example above, ID is a row with a UNIQUE constraint. This is important to ensure that a file isn't overwritten with the contents of subsequent row. Also, you must specify the file extension applicable to the data stored in the column. In this case, the Products.Photo stores image data (JPGs).

Views

Contents

System views

The catalog contains system tables that link together by keys and indexes. In SQL Anywhere, the system tables are hidden. However, there is a system view for each table. In some cases, a system view may also include columns from more than one system table, to satisfy a commonly needed join.

To ensure compatibility with future versions of the SQL Anywhere catalog, make sure your applications make use of system views and not the underlying system tables, which may change.

To view detailed system information views and definitions (Sybase Central)

1. Connect to the database as a user with DBA authority.
2. Right-click the database and choose **Configure Filter Owner**.
3. Click **SYS** and then click **OK**.
4. In the left pane, double-click **Views**.
5. In the left pane click a view, and in the right pane click the **SQL** tab.

   Click the **Data** tab to view details about the selected view.

SYSARTICLE system view

Each row of the SYSARTICLE system view describes an article in a publication. The underlying system table for this view is ISYSARTICLE.

Column name	Column type	Description
publication_id	UNSIGNED INT	The publication of which the article is a part.
table_id	UNSIGNED INT	Each article consists of columns and rows from a single table. This column contains the table ID for this table.
where_expr	LONG VARCHAR	For articles that contain a subset of rows defined by a WHERE clause, this column contains the search condition.
subscribe_by_expr	LONG VARCHAR	For articles that contain a subset of rows defined by a SUBSCRIBE BY expression, this column contains the expression.
query	CHAR(1)	Indicates information about the article type to the database server.
alias	VARCHAR(256)	The alias for the article.

Constraints on underlying system table

PRIMARY KEY (publication_id, table_id)

FOREIGN KEY (publication_id) references SYS.ISYSPUBLICATION (publication_id)

FOREIGN KEY (table_id) references SYS.ISYSTAB (table_id)

SYSARTICLECOL system view

Each row of the SYSARTICLECOL system view identifies a column in an article. The underlying system table for this view is ISYSARTICLECOL.

Column name	Column type	Description
publication_id	UNSIGNED INT	A unique identifier for the publication of which the column is a part.
table_id	UNSIGNED INT	The table to which the column belongs.
column_id	UNSIGNED INT	The column identifier, from the SYSTABCOL system view.

Constraints on underlying system table

PRIMARY KEY (publication_id, table_id, column_id)

FOREIGN KEY (publication_id, table_id) references SYS.ISYSARTICLE (publication_id, table_id)

FOREIGN KEY (table_id, column_id) references SYS.ISYSTABCOL (table_id, column_id)

SYSCAPABILITY system view

Each row of the SYSCAPABILITY system view identifies a capability of a remote server. The underlying system table for this view is ISYSCAPABILITY.

Column name	Column type	Description
capid	INTEGER	The ID of the capability, as listed in the SYSCAPABILITYNAME system view.
srvid	UNSIGNED INT	The server to which the capability applies, as listed in the SYSSERVER system view.
capvalue	CHAR(128)	The value of the capability.

Constraints on underlying system table

PRIMARY KEY (capid, srvid)

FOREIGN KEY (srvid) references SYS.ISYSSERVER (srvid)

See also

- “SYSCAPABILITYNAME system view” on page 992

SYSCAPABILITYNAME system view

Each row in the SYSCAPABILITYNAME system view names a capability that is defined in the SYSCAPABILITY system view.

Column name	Column type	Description
capid	INTEGER	A number uniquely identifying the capability.
capname	CHAR(128)	The name of the capability.

Remarks

The SYSCAPABILITYNAME system view is defined using a combination of sa_rowgenerator and the following server properties:

- RemoteCapability
- MaxRemoteCapability

See also

- "Database server properties" [*SQL Anywhere Server - Database Administration*]
- "SYSCAPABILITY system view" on page 991

SYSCHECK system view

Each row in the SYSCHECK system view provides the definition for a named check constraint in a table. The underlying system table for this view is ISYSCHECK.

Column name	Column type	Description
check_id	UNSIGNED INT	A number that uniquely identifies the constraint in the database.
check_defn	LONG VARCHAR	The CHECK expression.

Constraints on underlying system table

PRIMARY KEY (check_id)

FOREIGN KEY (check_id) references SYS.ISYSCONSTRAINT (constraint_id)

SYSCOLPERM system view

The GRANT statement can give UPDATE, SELECT, or REFERENCES permission to individual columns in a table. Each column with UPDATE, SELECT, or REFERENCES permission is recorded in one row of the SYSCOLPERM system view. The underlying system table for this view is ISYSCOLPERM.

Column name	Column type	Description
table_id	UNSIGNED INT	The table number for the table containing the column.
grantee	UNSIGNED INT	The user number of the user ID that is given permission on the column. If the grantee is the user number for the special PUBLIC user ID, the permission is given to all user IDs.
grantor	UNSIGNED INT	The user number of the user ID that grants the permission.
column_id	UNSIGNED INT	This column number, together with the table_id, identifies the column for which permission has been granted.
privilege_type	SMALLINT	The number in this column indicates the kind of column permission (16=REFERENCES, 1=SELECT, or 8=UPDATE).
is_grantable	CHAR(1)	Indicates if the permission on the column was granted WITH GRANT OPTION.

Constraints on underlying system table

PRIMARY KEY (table_id, grantee, grantor, column_id, privilege_type)

FOREIGN KEY (table_id, column_id) references SYS.ISYSTABCOL (table_id, column_id)

FOREIGN KEY (grantor) references SYS.ISYSUSER (user_id)

FOREIGN KEY (grantee) references SYS.ISYSUSER (user_id)

SYSCOLSTAT system view

The SYSCOLSTAT system view contains the column statistics, including histograms, that are used by the optimizer. The contents of this view are best retrieved using the sa_get_histogram stored procedure or the Histogram utility. The underlying system table for this view is ISYSCOLSTAT.

Column name	Column type	Description
table_id	UNSIGNED INT	A number that uniquely identifies the table or materialized view to which the column belongs.
column_id	UNSIGNED INT	A number that, together with table_id, uniquely identifies the column.
format_id	SMALLINT	For system use only.
update_time	TIMESTAMP	The time of the last update of the column statistics.

Column name	Column type	Description
density	FLOAT	An estimate of the average selectivity of a single value for the column, not counting the large single value selectivities stored in the row.
max_steps	SMALLINT	For system use only.
actual_steps	SMALLINT	For system use only.
step_values	LONG BINARY	For system use only.
frequencies	LONG BINARY	For system use only.

NOTE
For databases created using SQL Anywhere version 11.0.0 and later, the underlying system table for this view is always encrypted to protect the data from unauthorized access.

Constraints on underlying system table

PRIMARY KEY (table_id, column_id)

FOREIGN KEY (table_id, column_id) references SYS.ISYSTABCOL (table_id, column_id)

SYSCONSTRAINT system view

Each row in the SYSCONSTRAINT system view describes a named constraint in the database. The underlying system table for this view is ISYSCONSTRAINT.

Column name	Column type	Description
constraint_id	UNSIGNED INT	The unique ID for the constraint.
constraint_type	CHAR(1)	The type of constraint: ● C - column check constraint. ● T - table constraint. ● P - primary key. ● F - foreign key. ● U - unique constraint.
ref_object_id	UNSIGNED BIGINT	The object ID of the column, table, or index to which the constraint applies.

Column name	Column type	Description
table_object_id	UNSIGNED BIGINT	The table ID of the table to which the constraint applies.
constraint_name	CHAR(128)	The name of the constraint.

Constraints on underlying system table

PRIMARY KEY (constraint_id)

FOREIGN KEY (ref_object_id) references SYS.ISYSOBJECT (object_id)

FOREIGN KEY (table_object_id) references SYS.ISYSOBJECT (object_id)

SYSDBFILE system view

Each row in the SYSDBFILE system view describes a dbspace file. The underlying system table for this view is ISYSDBFILE.

Column name	Column type	Description
dbfile_id	SMALLINT	For internal use only.
dbspace_id	SMALLINT	Each dbspace file in a database is assigned a unique number. The system dbspace contains all system objects and has a dbspace_id of 0.
dbfile_name	CHAR(128)	The file name for the dbspace. For dbspaces other than system and TEMPORARY, the file name can be changed using the following statement: `ALTER DBSPACE` *`dbspace`* `RENAME '`*`new-file-name`*`';`
file_name	LONG VARCHAR	A unique name for the dbspace. It is used in the CREATE TABLE command.
lob_map	LONG VARBIT	For internal use only.

Constraints on underlying system table

PRIMARY KEY (dbfile_id)

FOREIGN KEY (dbspace_id) references SYS.ISYSDBSPACE (dbspace_id)

SYSDBSPACE system view

Each row in the SYSDBSPACE system view describes a dbspace file. The underlying system table for this view is ISYSDBSPACE.

Column name	Column type	Description
dbspace_id	SMALLINT	Unique number identifying the dbspace. The system dbspace contains all system objects and has a dbspace_id of 0.
object_id	UNSIGNED BIGINT	The file name for the dbspace. For the system dbspace, the value is the name of the database file when the database was created and is for informational purposes only; it cannot be changed. For other dbspaces, the file name can be changed using the following statement: `ALTER DBSPACE dbspace RENAME 'new-filename';`
dbspace_name	CHAR(128)	A unique name for the dbspace. It is used in the CREATE TABLE command.
store_type	TINYINT	For internal use only.

Constraints on underlying system table

PRIMARY KEY (dbspace_id)

FOREIGN KEY (object_id) references SYS.ISYSOBJECT MATCH UNIQUE FULL

SYSDBSPACEPERM system view

Each row in the SYSDBSPACEPERM system view describes a permission on a dbspace file. The underlying system table for this view is ISYSDBSPACEPERM.

Column name	Column type	Description
dbspace_id	SMALLINT	Unique number identifying the dbspace. The system dbspace contains all system objects and has a dbspace_id of 0.
grantee	UNSIGNED INT	The user ID of the user getting the permission.
privilege_type	SMALLINT	The permission that is granted to the grantee. For example, CREATE gives the grantee permission to create objects on the dbspace.

Constraints on underlying system table

FOREIGN KEY (dbspace_ID) references SYS.ISYSDBSPACE

FOREIGN KEY (grantee) references SYS.ISYSUSER (user_id)

See also

- “GRANT statement” on page 627
- “Database permissions and authorities overview” [*SQL Anywhere Server - Database Administration*]

SYSDEPENDENCY system view

Each row in the SYSDEPENDENCY system view describes a dependency between two database objects. The underlying system table for this view is ISYSDEPENDENCY.

A dependency exists between two database objects when one object references another object in its definition. For example, if the query specification for a view references a table, the view is said to be dependent on the table. The database server tracks dependencies of views on tables, views, materialized views, and columns.

Column name	Column type	Description
ref_object_id	UNSIGNED BIGINT	The object ID of the referenced object.
dep_object_id	UNSIGNED BIGINT	The ID of the referencing object.

Constraints on underlying system table

PRIMARY KEY (ref_object_id, dep_object_id)

FOREIGN KEY (ref_object_id) references SYS.ISYSOBJECT (object_id)

FOREIGN KEY (dep_object_id) references SYS.ISYSOBJECT (object_id)

See also

- “sa_dependent_views system procedure” on page 863
- “View dependencies” [*SQL Anywhere Server - SQL Usage*]

SYSDOMAIN system view

The SYSDOMAIN system view records information about built-in data types (also called domains). The contents of this view does not change during normal operation. The underlying system table for this view is ISYSDOMAIN.

Column name	Column type	Description
domain_id	SMALLINT	The unique number assigned to each data type. These numbers cannot be changed.
domain_name	CHAR(128)	The name of the data type normally found in the CREATE TABLE command, such as CHAR or INTEGER.

Column name	Column type	Description
type_id	SMALLINT	The ODBC data type. This value corresponds to the value for data_type in the Transact-SQL-compatibility dbo.SYSTYPES table.
"precision"	SMALLINT	The number of significant digits that can be stored using this data type. The column value is NULL for non-numeric data types.

Constraints on underlying system table

PRIMARY KEY (domain_id)

SYSEVENT system view

Each row in the SYSEVENT system view describes an event created with CREATE EVENT. The underlying system table for this view is ISYSEVENT.

Column name	Column type	Description
event_id	UNSIGNED INT	The unique number assigned to each event.
object_id	UNSIGNED BIGINT	The internal ID for the event, uniquely identifying it in the database.
creator	UNSIGNED INT	The user number of the owner of the event. The name of the user can be found by looking in the SYSUSER system view.
event_name	VARCHAR(128)	The name of the event.
enabled	CHAR(1)	Indicates whether the event is allowed to fire.
location	CHAR(1)	The location where the event is to fire: ● C = consolidated ● R = remote ● A = all
event_type_id	UNSIGNED INT	For system events, the event type as listed in the SYSEVENT-TYPE system view.
action	LONG VARCHAR	The event handler definition. An obfuscated value indicates a hidden event.
external_action	LONG VARCHAR	For system use only.
condition	LONG VARCHAR	The condition used to control firing of the event handler.

Column name	Column type	Description
remarks	LONG VARCHAR	Remarks for the event; this column comes from ISYSRE-MARK.
source	LONG VARCHAR	The original source for the event; this column comes from ISYSSOURCE.

Constraints on underlying system table

PRIMARY KEY (event_id)

FOREIGN KEY (creator) references SYS.ISYSUSER (user_id)

FOREIGN KEY (object_id) references SYS.ISYSOBJECT (object_id) MATCH UNIQUE FULL

See also

- "SYSEVENTTYPE system view" on page 999

SYSEVENTTYPE system view

The SYSEVENTTYPE system view defines the system event types that can be referenced by CREATE EVENT.

Column name	Column type	Description
event_type_id	UNSIGNED INT	The unique number assigned to each event type.
name	VARCHAR(128)	The name of the system event type.
description	LONG VARCHAR	A description of the system event type.

Remarks

The SYSEVENTTYPE system view is defined using a combination of sa_rowgenerator and the following server properties:

- EventTypeName
- EventTypeDesc
- MaxEventType

See also

- "Database server properties" [*SQL Anywhere Server - Database Administration*]
- "SYSEVENT system view" on page 998

SYSEXTERNENV system view

SQL Anywhere includes support for six external runtime environments. These include embedded SQL and ODBC applications written in C/C++, and applications written in Java, Perl, PHP, or languages such as C# and Visual Basic that are based on the Microsoft .NET Framework Common Language Runtime (CLR).

Each row in the SYSEXTERNENV system view describes the information needed to identify and launch each of the external environments. The underlying system table for this view is ISYSEXTERNENV.

Column name	Column type	Description
object_id	unsigned bigint	A unique identifier for the external environment.
name	varchar(128)	The name of the external environment or language.
scope	char(1)	Identifies if the external environment is launched as one-per-connection (C), or one-per-database (D).
support_result_sets	char(1)	Identifies the external environments that can return result sets to the user.
location	long varchar	Identifies the location where the main executable for the environment can be found.
options	long varchar	Identifies the options required on the command line to launch the external environment.
user_id	unsigned int	Identifies a user in the database that has DBA authority.

Constraints on underlying system table

PRIMARY KEY (object_id)

FOREIGN KEY (object_id) references SYS.ISYSOBJECT (object_id) MATCH UNIQUE FULL

FOREIGN KEY (user_id) references SYS.ISYSUSER (user_id)

SYSEXTERNENVOBJECT system view

SQL Anywhere includes support for six external runtime environments. These include embedded SQL and ODBC applications written in C/C++, and applications written in Java, Perl, PHP, or languages such as C# and Visual Basic that are based on the Microsoft .NET Framework Common Language Runtime (CLR).

Each row in the SYSEXTERNENVOBJECT system view describes an installed external object. The underlying system table for this view is ISYSEXTERNENVOBJECT.

Column name	Column type	Description
object_id	unsigned bigint	A unique identifier for the external object.
extenv_id	unsigned bigint	The unique identifier for the external environment (SYSEXTERNENV.object_id).
owner	unsigned int	Identifies the creator/owner of the external object.
name	long varchar	Identifies the name of the external object as specified in the INSTALL statement.
contents	long binary	The contents of the external object.
update_time	timestamp	Identifies the last time the object was modified (or installed).

Constraints on underlying system table

PRIMARY KEY (object_id)

FOREIGN KEY (object_id) references SYS.ISYSOBJECT (object_id) MATCH UNIQUE FULL

FOREIGN KEY (extenv_id) references SYS.ISYSEXTERNENV (object_id)

FOREIGN KEY (owner) references SYS.ISYSUSER (user_id)

SYSEXTERNLOGIN system view

Each row in the SYSEXTERNLOGIN system view describes an external login for remote data access. The underlying system table for this view is ISYSEXTERNLOGIN.

Note
Previous versions of the catalog contained a SYSEXTERNLOGINS system table. That table has been renamed to be ISYSEXTERNLOGIN (without an 'S'), and is the underlying table for this view.

Column name	Column type	Description
user_id	UNSIGNED INT	The user ID on the local database.
srvid	UNSIGNED INT	The remote server, as listed in the SYSSERVER system view.
remote_login	VARCHAR(128)	The login name for the user, for the remote server.
remote_password	VARBINARY(128)	The password for the user, for the remote server.

NOTE
For databases created using SQL Anywhere version 11.0.0 and later, the underlying system table for this view is always encrypted to protect the data from unauthorized access.

Constraints on underlying system table

PRIMARY KEY (user_id, srvid)

FOREIGN KEY (user_id) references SYS.ISYSUSER (user_id)

FOREIGN KEY (srvid) references SYS.ISYSSERVER (srvid)

SYSFKEY system view

Each row in the SYSFKEY system view describes a foreign key constraint in the system. The underlying system table for this view is ISYSFKEY.

Column name	Column type	Description
foreign_table_id	UNSIGNED INT	The table number of the foreign table.
foreign_index_id	UNSIGNED INT	The index number for the foreign key.
primary_table_id	UNSIGNED INT	The table number of the primary table.
primary_index_id	UNSIGNED INT	The index number of the primary key.
match_type	TINYINT	The matching type for the constraint. Matching types include: ● 0 - Use the default matching ● 1 - SIMPLE ● 2 - FULL ● 129 - SIMPLE UNIQUE ● 130 - FULL UNIQUE For more information about match types, see the MATCH clause of the "CREATE TABLE statement" on page 523.
check_on_commit	CHAR(1)	Indicates whether INSERT and UPDATE statements should wait until the COMMIT to check if foreign keys are still valid.

Column name	Column type	Description
nulls	CHAR(1)	Indicates whether the columns in the foreign key are allowed to contain the NULL value. Note that this setting is independent of the nulls setting in the columns contained in the foreign key.

Constraints on underlying system table

PRIMARY KEY (foreign_table_id, foreign_index_id)

FOREIGN KEY (foreign_table_id, foreign_index_id) references SYS.ISYSIDX (table_id, index_id)

FOREIGN KEY (primary_table_id, primary_index_id) references SYS.ISYSIDX (table_id, index_id)

SYSGROUP system view

There is one row in the SYSGROUP system view for each member of each group. This view describes the many-to-many relationship between groups and members. A group may have many members, and a user may be a member of many groups. The underlying system table for this view is ISYSGROUP.

Column name	Column type	Description
group_id	UNSIGNED INT	The user number of the group.
group_member	UNSIGNED INT	The user number of a member.

Constraints on underlying system table

PRIMARY KEY (group_id, group_member)

FOREIGN KEY group_id (group_id) references SYS.ISYSUSER (user_id)

FOREIGN KEY group_member (group_member) references SYS.ISYSUSER (user_id)

SYSHISTORY system view

Each row in the SYSHISTORY system view records a system operation on the database, such as a database start, a database calibration, and so on. The underlying system table for this view is ISYSHISTORY.

Column name	Column type	Description
operation	CHAR(128)	The type of operation performed on the database file. The operation must be one of the following values: ● INIT - Information about when the database was created. ● UPGRADE - Information about when the database was upgraded. ● START - Information about when the database was started using a specific version of the database server on a particular operating system. ● LAST_START - Information about the most recent time the database server was started. A LAST_START operation is converted to a START operation when the database is started with a different version of the database server and/or on a different operating system than those values currently stored in the LAST_START row. ● DTT - Information about the *second to last* Disk Transfer Time (DTT) calibration operation performed on the dbspace. That is, information on the second to last execution of either an ALTER DATABASE CALIBRATE or ALTER DATABASE RESTORE DEFAULT CALIBRATION statement. ● LAST_DTT - Information about the *most recent* DTT calibration operation performed on the dbspace. That is, information on the most recent execution of either an ALTER DATABASE CALIBRATE or ALTER DATABASE RESTORE DEFAULT CALIBRATION statement. ● LAST_BACKUP - Information about the last backup, including date and time of the backup, the backup type, the files that were backed up, and the version of database server that performed the backup.
object_id	UNSIGNED INT	For any operation other than DTT and LAST_DTT, the value in this column will be 0. For DTT and LAST_DTT operations, this is the dbspace_id of the dbspace as defined in the SYSDBSPACE system view. See "SYSDBSPACE system view" on page 996.
sub_operation	CHAR(128)	For any operation other than DTT and LAST_DTT, the value in this column will be a set of empty single quotes (''). For DTT and LAST_DTT operations, this column contains the type of sub-operation performed on the dbspace. Values include: ● DTT_SET - The dbspace calibration has been set. ● DTT_UNSET - The dbspace calibration has been restored to the default setting.

Column name	Column type	Description
version	CHAR(128)	The version and build number of the database server used to carry out the operation.
platform	CHAR(128)	The operating system on which the operation was carried out.
first_time	TIME-STAMP	The date and time the database was first started on a particular operating system with a particular version of the software.
last_time	TIME-STAMP	The most recent date and time the database was started on a particular operating system with a particular version of the software.
details	LONG VAR-CHAR	This column stores information such as command line options used to start the database server or the capability bits enabled for the database. This information is for use by technical support.

Constraints on underlying system table

PRIMARY KEY (operation, object_id, version, platform)

SYSIDX system view

Each row in the SYSIDX system view defines a logical index in the database. The underlying system table for this view is ISYSIDX.

Column name	Column type	Description
table_id	UNSIGNED INT	Uniquely identifies the table to which this index applies.
index_id	UNSIGNED INT	A unique number identifying the index within its table.
object_id	UNSIGNED BIGINT	The internal ID for the index, uniquely identifying it in the database.
phys_index_id	UNSIGNED INT	Identifies the underlying physical index used to implement the logical index. This value is NULL for indexes on temporary tables or remote tables. Otherwise, the value corresponds to the object_id of a physical index in the SYSPHYSIDX system view. See "SYSPHYSIDX system view" on page 1014.
dbspace_id	SMALLINT	The ID of the file in which the index is contained. This value corresponds to an entry in the SYSDBSPACE system view. See "SYSDBSPACE system view" on page 996.

Column name	Column type	Description
file_id	SMALLINT	DEPRECATED. This column is present in SYSVIEW, but not in the underlying system table ISYSIDX. The contents of this column is the same as dbspace_id and is provided for compatibility. Use dbspace_id instead.
index_category	TINYINT	The type of index. Values include: ● 1 - Primary key ● 2 - Foreign key ● 3 - Secondary index (includes unique constraints) ● 4 - Text indexes
"unique"	TINYINT	Indicates whether the index is a unique index (1), a non-unique index (4), or a unique constraint (2). A unique index prevents two rows in the indexed table from having the same values in the index columns.
index_name	CHAR(128)	The name of the index.
not_enforced	CHAR(1)	For system use only.
file_id	SMALLINT	For system use only.

Constraints on underlying system table

PRIMARY KEY (table_id, index_id)

FOREIGN KEY (table_id) references SYS.ISYSTAB (table_id)

FOREIGN KEY (object_id) references SYS.ISYSOBJECT (object_id) MATCH UNIQUE FULL

FOREIGN KEY (table_id, phys_index_id) references SYS.ISYSPHYSIDX (table_id, phys_index_id)

See also

- "SYSIDXCOL system view" on page 1006
- "SYSPHYSIDX system view" on page 1014
- "SYSDBSPACE system view" on page 996

SYSIDXCOL system view

Each row in the SYSIDXCOL system view describes one column of an index described in the SYSIDX system view. The underlying system table for this view is ISYSIDXCOL.

Column name	Column type	Description
table_id	UNSIGNED INT	Identifies the table to which the index applies.

Column name	Column type	Description
index_id	UNSIGNED INT	Identifies the index to which the column applies. Together, table_id and index_id identify one index described in the SYSIDX system view.
sequence	SMALLINT	Each column in an index is assigned a unique number starting at 0. The order of these numbers determines the relative significance of the columns in the index. The most important column has sequence number 0.
column_id	UNSIGNED INT	Identifies which column of the table is indexed. Together, table_id and column_id identify one column described in the SYSCOLUMN system view.
"order"	CHAR(1)	Indicates whether the column in the index is kept in ascending(A) or descending(D) order. This value is NULL for text indexes.
primary_column_id	UNSIGNED INT	The ID of the primary key column that corresponds to this foreign key column. The value is NULL for non foreign key columns.

Constraints on underlying system table

PRIMARY KEY (table_id, index_id, column_id)

FOREIGN KEY (table_id, index_id) references SYS.ISYSIDX (table_id, index_id)

FOREIGN KEY (table_id, column_id) references SYS.ISYSTABCOL (table_id, column_id)

See also

- "SYSIDX system view" on page 1005

SYSJAR system view

Each row in the SYSJAR system view defines a JAR file stored in the database. The underlying system table for this view is ISYSJAR.

Column name	Column type	Description
jar_id	INTEGER	A unique number identifying the JAR file.
object_id	UNSIGNED BIGINT	The internal ID for the JAR file, uniquely identifying it in the database.
creator	UNSIGNED INT	The user number of the creator of the JAR file.

Column name	Column type	Description
jar_name	LONG VARCHAR	The name of the JAR file.
jar_file	LONG VARCHAR	The external file name of the JAR file within the database.
update_time	TIMESTAMP	The time the JAR file was last updated.

Constraints on underlying system table

PRIMARY KEY (jar_id)

FOREIGN KEY (object_id) references SYS.ISYSOBJECT (object_id) MATCH UNIQUE FULL

See also

- "SYSJARCOMPONENT system view" on page 1008

SYSJARCOMPONENT system view

Each row in the SYSJAR system view defines a JAR file component. The underlying system table for this view is ISYSJARCOMPONENT.

Column name	Column type	Description
component_id	INTEGER	The primary key containing the id of the component.
jar_id	INTEGER	A field containing the ID number of the JAR.
component_name	LONG VARCHAR	The name of the component.
component_type	CHAR(1)	The type of the component.
contents	LONG BINARY	The byte code of the JAR file.

Constraints on underlying system table

PRIMARY KEY (component_id)

FOREIGN KEY (jar_id) references SYS.ISYSJAR (jar_id)

See also

- "SYSJAR system view" on page 1007

SYSJAVACLASS system view

Each row in the SYSJAVACLASS system view describes one Java class stored in the database. The underlying system table for this view is ISYSJAVACLASS.

Column name	Column type	Description
class_id	INTEGER	The unique number for the Java class. Also the primary key for the table.
object_id	UNSIGNED BIGINT	The internal ID for the Java class, uniquely identifying it in the database.
creator	UNSIGNED INT	The user number of the creator of the class.
jar_id	INTEGER	The id of the JAR file from which the class came.
class_name	LONG VARCHAR	The name of the Java class.
public	CHAR(1)	Indicates whether the class is public (Y) or private (N).
component_id	INTEGER	The id of the component in the SYSJARCOMPONENT system view.
update_time	TIMESTAMP	The last update time of the class.

Constraints on underlying system table

PRIMARY KEY (class_id)

FOREIGN KEY (object_id) references SYS.ISYSOBJECT (object_id) MATCH UNIQUE FULL

FOREIGN KEY (creator) references SYS.ISYSUSER (user_id)

FOREIGN KEY (component_id) references SYS.ISYSJARCOMPONENT (component_id)

SYSLOGINMAP system view

The SYSLOGINMAP system view contains one row for each user that can connect to the database using either an integrated login, or Kerberos login. As a security measure, only users with DBA authority can view the contents of this view. The underlying system table for this view is ISYSLOGINMAP.

Column name	Column type	Description
login_mode	TINYINT	The type of login: 1 for integrated logins, 2 for Kerberos logins.
login_id	VARCHAR(1024)	Either the integrated login user profile name, or the Kerberos principal that maps to database_uid.

Column name	Column type	Description
object_id	UNSIGNED BIGINT	A unique identifier, one for each mapping between user ID and database user ID.
database_uid	UNSIGNED INT	The database user ID to which the login ID is mapped.

Constraints on underlying system table

PRIMARY KEY (login_mode, login_id)

FOREIGN KEY (object_id) references SYS.ISYSOBJECT (object_id) MATCH UNIQUE FULL

FOREIGN KEY (database_uid) references SYS.ISYSUSER (user_id)

SYSLOGINPOLICY system view

The underlying system table for this view is ISYSLOGINPOLICY.

Column name	Column type	Description
login_policy_id	UNSIGNED BIGINT	A unique identifier for the login policy.
login_policy_name	CHAR(128)	The name of the login policy.

Constraints on underlying system table

PRIMARY KEY (login_policy_id)

FOREIGN KEY (login_policy_id) references SYS.ISYSOBJECT (object_id)

UNIQUE (login_policy_name)

See also

- “SYSLOGINPOLICYOPTION system view” on page 1010
- “SYSUSER system view” on page 1039

SYSLOGINPOLICYOPTION system view

The underlying system table for this view is ISYSLOGINPOLICYOPTION.

Column name	Column type	Description
login_policy_id	UNSIGNED BIGINT	A unique identifier for the login policy.
login_option_name	CHAR(128)	The name of the login policy.

Column name	Column type	Description
login_option_value	LONG VARCHAR	The value of the login policy at the time it was created.

Constraints on underlying system table

PRIMARY KEY (login_policy_id, login_option_name)

FOREIGN KEY (login_policy_id) references SYS.ISYSLOGINPOLICY (login_policy_id)

See also

- "SYSLOGINPOLICY system view" on page 1010
- "SYSUSER system view" on page 1039

SYSMVOPTION system view

Each row in the SYSMVOPTION system view describes the setting of one option value for a materialized view at the time of its creation. However, the description does not contain the option name for the option. The underlying system table for this view is ISYSMVOPTION.

Column name	Column type	Description
view_object_id	UNSIGNED BIGINT	The object ID of the materialized view.
option_id	UNSIGNED INT	A unique number identifying the option in the database. To see the option name, see the SYSMVOPTIONNAME system view.
option_value	LONG VARCHAR	The value of the option when the materialized view was created.

Constraints on underlying system table

PRIMARY KEY (view_object_id, option_id)

FOREIGN KEY (view_object_id) references SYS.ISYSOBJECT (object_id)

FOREIGN KEY (option_id) references SYS.ISYSMVOPTIONNAME (option_id)

See also

- "SYSMVOPTIONNAME system view" on page 1011

SYSMVOPTIONNAME system view

Each row in the SYSMVOPTIONNAME system view contains the name of an option defined in the SYSMVOPTION system view. The underlying system table for this view is ISYSMVOPTIONNAME.

Column name	Column type	Description
option_id	UNSIGNED INT	A number uniquely identifying the option in the database.
option_name	CHAR(128)	The name of the option.

Constraints on underlying system table

PRIMARY KEY (option_id)

See also

- "SYSMVOPTION system view" on page 1011

SYSOBJECT system view

Each row in the SYSOBJECT system view describes a database object. The underlying system table for this view is ISYSOBJECT.

Column name	Column type	Description
object_id	UN-SIGNED BIGINT	The internal ID for the object, uniquely identifying it in the database.
status	TI-NYINT	The status of the object. Values include: ● 1 (valid) - The object is available for use by the database server. This status is synonymous with ENABLED. That is, if you ENABLE an object, the status changes to VALID. ● 2 (invalid) - An attempt to recompile the object after an internal operation has failed, for example, after a schema-altering modification to an object on which it depends. The database server continues to try to recompile the object whenever it is referenced in a statement. ● 4 (disabled) - The object has been explicitly disabled by the user, for example using an ALTER TABLE...DISABLE VIEW DEPENDENCIES statement.

Column name	Column type	Description
object_type	TI-NYINT	Type of object. Values include: ● 1 - Table ● 2 - View ● 3 - Materialized view ● 4 - Column ● 5 - Index ● 6 - Procedure ● 7 - Trigger ● 8 - Event ● 9 - User ● 10 - Publication ● 11 - Remote type ● 12 - Login mapping ● 13 - JAR ● 14 - Java class ● 16 - Service ● 17 - Text configuration ● 18 - Dbspace
crea-tion_time	TIME-STAMP	The date and time when the object was created.

Constraints on underlying system table

PRIMARY KEY (object_id)

SYSOPTION system view

The SYSOPTION system view contains the options one row for each option setting stored in the database. Each user can have their own setting for a given option. In addition, settings for the PUBLIC user ID define the default settings to be used for users that do not have their own setting. The underlying system table for this view is ISYSOPTION.

Column name	Column type	Description
user_id	UNSIGNED INT	The user number to whom the option setting applies.
"option"	CHAR(128)	The name of the option.
"setting"	LONG VARCHAR	The current setting for the option.

Constraints on underlying system table

PRIMARY KEY (user_id, "option")

FOREIGN KEY (user_id) references SYS.ISYSUSER (user_id)

SYSOPTSTAT system view

The SYSOPTSTAT system view stores the cost model calibration information as computed by the ALTER DATABASE CALIBRATE statement. The contents of this view are for internal use only and are best accessed via the sa_get_dtt system procedure. The underlying system table for this view is ISYSOPTSTAT.

Column name	Column type	Description
stat_id	UNSIGNED INT	For system use only.
group_id	UNSIGNED INT	For system use only.
format_id	SMALLINT	For system use only.
data	LONG BINARY	For system use only.

Constraints on underlying system table

PRIMARY KEY (stat_id, group_id, format_id)

SYSPHYSIDX system view

Each row in the SYSPHYSIDX system view defines a physical index in the database. The underlying system table for this view is ISYSPHYSIDX.

Column name	Column type	Description
table_id	UNSIGNED INT	The object ID of the table to which the index corresponds.
phys_index_id	UNSIGNED INT	The unique number of the physical index within its table.
root	INTEGER	Identifies the location of the root page of the physical index in the database file.
key_value_count	UNSIGNED INT	The number of distinct key values in the index.
leaf_page_count	UNSIGNED INT	The number of leaf index pages.
depth	UNSIGNED SMALLINT	The depth (number of levels) of the physical index.

Column name	Column type	Description
max_key_distance	UNSIGNED INT	For system use only.
seq_transitions	UNSIGNED INT	For system use only.
rand_transitions	UNSIGNED INT	For system use only.
rand_distance	UNSIGNED INT	For system use only.
allocation_bitmap	LONG VARBIT	For system use only.
long_value_bitmap	LONG VARBIT	For system use only.

Constraints on underlying system table

PRIMARY KEY (table_id, phys_index_id)

See also

- "SYSIDXCOL system view" on page 1006
- "SYSIDX system view" on page 1005

SYSPROCEDURE system view

Each row in the SYSPROCEDURE system view describes one procedure in the database. The underlying system table for this view is ISYSPROCEDURE.

Column name	Column type	Description
proc_id	UNSIGNED INT	Each procedure is assigned a unique number (the procedure number).
creator	UNSIGNED INT	The owner of the procedure.
object_id	UNSIGNED BIGINT	The internal ID for the procedure, uniquely identifying it in the database.
proc_name	CHAR(128)	The name of the procedure. One creator cannot have two procedures with the same name.
proc_defn	LONG VARCHAR	The definition of the procedure.
remarks	LONG VARCHAR	Remarks about the procedure. This value is stored in the ISYSREMARK system table.
replicate	CHAR(1)	Indicates whether the procedure is a primary data source in a Replication Server installation.

Column name	Column type	Description
srvid	UNSIGNED INT	If the procedure is a proxy for a procedure on a remote database server, indicates the remote server.
source	LONG VARCHAR	The preserved source for the procedure. This value is stored in the ISYSSOURCE system table.
avg_num_rows	FLOAT	Information collected for use in query optimization when the procedure appears in the FROM clause.
avg_cost	FLOAT	Information collected for use in query optimization when the procedure appears in the FROM clause.
stats	LONG BINARY	Information collected for use in query optimization when the procedure appears in the FROM clause.

Constraints on underlying system table

PRIMARY KEY (proc_id)

FOREIGN KEY (srvid) references SYS.ISYSSERVER (srvid)

FOREIGN KEY (object_id) references SYS.ISYSOBJECT (object_id) MATCH UNIQUE FULL

FOREIGN KEY (creator) references SYS.ISYSUSER (user_id)

SYSPROCPARM system view

Each row in the SYSPROCPARM system view describes one parameter to a procedure in the database. The underlying system table for this view is ISYSPROCPARM.

Column name	Column type	Description
proc_id	UNSIGNED INT	Uniquely identifies the procedure to which the parameter belongs.
parm_id	SMALLINT	Each procedure starts numbering parameters at 1. The order of parameter numbers corresponds to the order in which they were defined. For functions, the first parameter has the name of the function and represents the return value for the function.

Column name	Column type	Description
parm_type	SMALLINT	The type of parameter will be one of the following: ● 0 - Normal parameter (variable) ● 1 - Result variable - used with a procedure that returns result sets ● 2 - SQLSTATE error value ● 3 - SQLCODE error value ● 4 - Return value from function
parm_mode_in	CHAR(1)	Indicates whether the parameter supplies a value to the procedure (IN or INOUT parameters).
parm_mode_out	CHAR(1)	Indicates whether the parameter returns a value from the procedure (OUT or INOUT parameters) or columns in the RESULT clause.
domain_id	SMALLINT	Identifies the data type for the parameter, by the data type number listed in the SYSDOMAIN system view.
width	UNSIGNED INT	Contains the length of a string parameter, the precision of a numeric parameter, or the number of bytes of storage for any other data type.
scale	SMALLINT	For numeric data types, the number of digits after the decimal point. For all other data types, the value of this column is 1.
user_type	SMALLINT	The user type of the parameter, if applicable.
parm_name	CHAR(128)	The name of the procedure parameter.
"default"	LONG VARCHAR	Default value of the parameter. Provided for informational purposes only.
remarks	LONG VARCHAR	Always returns NULL. Provided to allow the use of previous versions of ODBC drivers with newer personal database servers.

Constraints on underlying system table

PRIMARY KEY (proc_id, parm_id)

FOREIGN KEY (proc_id) references SYS.ISYSPROCEDURE (proc_id)

FOREIGN KEY (domain_id) references SYS.ISYSDOMAIN (domain_id)

FOREIGN KEY (user_type) references SYS.ISYSUSERTYPE (type_id)

SYSPROCPERM system view

Each row of the SYSPROCPERM system view describes a user granted permission to execute a procedure. Only users who have been granted permission can execute a procedure. The underlying system table for this view is ISYSPROCPERM.

Column name	Column type	Description
proc_id	UNSIGNED INT	The procedure number uniquely identifies the procedure for which permission has been granted.
grantee	UNSIGNED INT	The user number of the user receiving the permission.

Constraints on underlying system table

PRIMARY KEY (proc_id, grantee)

FOREIGN KEY (grantee) references SYS.ISYSUSER (user_id)

FOREIGN KEY (proc_id) references SYS.ISYSPROCEDURE (proc_id)

SYSPROXYTAB system view

Each row of the SYSPROXYTAB system view describes the remote parameters of one proxy table. The underlying system table for this view is ISYSPROXYTAB.

Column name	Column type	Description
table_object_id	UNSIGNED BIGINT	The object ID of the proxy table.
existing_obj	CHAR(1)	Indicates whether the proxy table previously existed on the remote server .
srvid	UNSIGNED INT	The unique ID for the remote server associated with the proxy table.
remote_location	LONG VARCHAR	The location of the proxy table on the remote server.

Constraints on underlying system table

PRIMARY KEY (table_object_id)

FOREIGN KEY (table_object_id) references ISYSOBJECT (object_id) MATCH UNIQUE FULL

FOREIGN KEY (srvid) references SYS.ISYSSERVER (srvid)

SYSPUBLICATION system view

Each row in the SYSPUBLICATION system view describes a SQL Remote or MobiLink publication. The underlying system table for this view is ISYSPUBLICATION.

Column name	Column type	Description
publication_id	UNSIGNED INT	A number uniquely identifying the publication.
object_id	UNSIGNED BIGINT	The internal ID for the publication, uniquely identifying it in the database.
creator	UNSIGNED INT	The owner of the publication.
publication_name	CHAR(128)	The name of the publication.
remarks	LONG VARCHAR	Remarks about the publication. This value is stored in the ISYSREMARK system table.
type	CHAR(1)	This column is deprecated.
sync_type	UNSIGNED INT	The type of synchronization for the publication. Values include: ● logscan - This is a regular publication that uses the transaction log to upload all relevant data that has changed since the last upload. ● scripted upload - For this publication, the transaction log is ignored and the upload is defined by the user using stored procedures. Information about the stored procedures is stored in the ISYSSYNCSCRIPT system table. ● download only - This is a download-only publication; no data is uploaded.

Constraints on underlying system table

PRIMARY KEY (publication_id)

FOREIGN KEY (object_id) references SYS.ISYSOBJECT (object_id) MATCH UNIQUE FULL

FOREIGN KEY (creator) references SYS.ISYSUSER (user_id)

See also

- "Scripted upload" [*MobiLink - Client Administration*]
- "SYSSYNCSCRIPT system view" on page 1027

SYSREMARK system view

Each row in the SYSREMARK system view describes a remark (or comment) for an object. The underlying system table for this view is ISYSREMARK.

Column	Data type	Description
object_id	UNSIGNED BIGINT	The internal ID for the object that has an associated remark.
remarks	LONG VARCHAR	The remark or comment associated with the object.

Constraints on underlying system table

PRIMARY KEY (object_id)

FOREIGN KEY (object_id) references SYS.ISYSOBJECT (object_id) MATCH UNIQUE FULL

SYSREMOTEOPTION system view

Each row in the SYSREMOTEOPTION system view describes the value of a SQL Remote message link parameter. The underlying system table for this view is ISYSREMOTEOPTION.

Some columns in this view contain potentially sensitive data. For that reason, access to this view is restricted to users with DBA authority. The SYSREMOTEOPTION2 view provides public access to the data in this view except for the potentially sensitive columns.

Column	Data type	Description
option_id	UNSIGNED INT	An identification number for the message link parameter.
user_id	UNSIGNED INT	The user ID for which the parameter is set.
"setting"	VARCHAR(255)	The value of the message link parameter.

Constraints on underlying system table

PRIMARY KEY (option_id, user_id)

FOREIGN KEY (option_id) references SYS.ISYSREMOTEOPTIONTYPE (option_id)

FOREIGN KEY (user_id) references SYS.ISYSUSER (user_id)

SYSREMOTEOPTIONTYPE system view

Each row in the SYSREMOTEOPTIONTYPE system view describes one of the SQL Remote message link parameters. The underlying system table for this view is ISYSREMOTEOPTIONTYPE.

Column	Data type	Description
option_id	UNSIGNED INT	An identification number for the message link parameter.
type_id	SMALLINT	An identification number for the message type that uses the parameter.
"option"	VARCHAR(128)	The name of the message link parameter.

Constraints on underlying system table

PRIMARY KEY (option_id)

FOREIGN KEY (type_id) references SYS.ISYSREMOTETYPE (type_id)

SYSREMOTETYPE system view

The SYSREMOTETYPE system view contains information about SQL Remote. The underlying system table for this view is ISYSREMOTETYPE.

Column name	Column type	Description
type_id	SMALLINT	Identifies which of the message systems supported by SQL Remote is to be used to send messages to the user.
object_id	UNSIGNED BIGINT	The internal ID for the remote type, uniquely identifying it in the database.
type_name	CHAR(128)	The name of the message system supported by SQL Remote.
publisher_address	LONG VARCHAR	The address of the remote database publisher.
remarks	LONG VARCHAR	Remarks about the remote type. This value is stored in the ISYSREMARK system table.

Constraints on underlying system table

PRIMARY KEY (type_id)

FOREIGN KEY (object_id) references SYS.ISYSOBJECT (object_id) MATCH UNIQUE FULL

SYSREMOTEUSER system view

Each row in the SYSREMOTEUSER system view describes a user ID with REMOTE permissions (a subscriber), together with the status of SQL Remote messages that were sent to and from that user. The underlying system table for this view is ISYSREMOTEUSER.

Column name	Column type	Description
user_id	UNSIGNED INT	The user number of the user with REMOTE permissions.
consolidate	CHAR(1)	Indicates whether the user was granted CONSOLIDATE permissions (Y) or REMOTE permissions (N).
type_id	SMALLINT	Identifies which of the message systems supported by SQL Remote is used to send messages to the user.
address	LONG VARCHAR	The address to which SQL Remote messages are to be sent. The address must be appropriate for the address_type.
frequency	CHAR(1)	How frequently SQL Remote messages are sent.
send_time	TIME	The next time messages are to be sent to this user.
log_send	UNSIGNED BIGINT	Messages are sent only to subscribers for whom log_send is greater than log_sent.
time_sent	TIMESTAMP	The time the most recent message was sent to this subscriber.
log_sent	UNSIGNED BIGINT	The log offset for the most recently sent operation.
confirm_sent	UNSIGNED BIGINT	The log offset for the most recently confirmed operation from this subscriber.
send_count	INTEGER	How many SQL Remote messages have been sent.
resend_count	INTEGER	Counter to ensure that messages are applied only once at the subscriber database.
time_received	TIMESTAMP	The time when the most recent message was received from this subscriber.
log_received	UNSIGNED BIGINT	The log offset in the subscriber's database for the operation that was most recently received at the current database.
confirm_received	UNSIGNED BIGINT	The log offset in the subscriber's database for the most recent operation for which a confirmation message has been sent.
receive_count	INTEGER	How many messages have been received.
rereceive_count	INTEGER	Counter to ensure that messages are applied only once at the current database.

Constraints on underlying system table

PRIMARY KEY (user_id)

FOREIGN KEY (user_id) references SYS.ISYSUSER (user_id)

FOREIGN KEY (type_id) references SYS.ISYSREMOTETYPE (type_id)

SYSSCHEDULE system view

Each row in the SYSSCHEDULE system view describes a time at which an event is to fire, as specified by the SCHEDULE clause of CREATE EVENT. The underlying system table for this view is ISYSSCHEDULE.

Column name	Column type	Description
event_id	UNSIGNED INT	The unique number assigned to each event.
sched_name	VAR-CHAR(128)	The name associated with the schedule for the event.
recurring	TINYINT	Indicates if the schedule is repeating.
start_time	TIME	The schedule start time.
stop_time	TIME	The schedule stop time if BETWEEN was used.
start_date	DATE	The first date on which the event is scheduled to execute.
days_of_week	TINYINT	A bit mask indicating the days of the week on which the event is scheduled: ● x01 = Sunday ● x02 = Monday ● x04 = Tuesday ● x08 = Wednesday ● x10 = Thursday ● x20 = Friday ● x40 = Saturday
days_of_month	UNSIGNED INT	A bit mask indicating the days of the month on which the event is scheduled. Some examples include: ● x01 = first day ● x02 = second day ● x40000000 = 31st day ● x80000000 = last day of month

Column name	Column type	Description
interval_units	CHAR(10)	The interval unit specified by EVERY: ● HH = hours ● NN = minutes ● SS = seconds
interval_amt	INTEGER	The period specified by EVERY.

Constraints on underlying system table

PRIMARY KEY (event_id, sched_name)

FOREIGN KEY (event_id) references SYS.ISYSEVENT (event_id)

SYSSERVER system view

Each row in the SYSSERVER system view describes a remote server. The underlying system table for this view is ISYSSERVER.

Note
Previous versions of the catalog contained a SYSSERVERS system table. That table has been renamed to be ISYSSERVER (without an 'S'), and is the underlying table for this view.

Column name	Column type	Description
srvid	UNSIGNED INT	An identifier for the remote server.
srvname	VARCHAR(128)	The name of the remote server.
srvclass	LONG VARCHAR	The server class, as specified in the CREATE SERVER statement.
srvinfo	LONG VARCHAR	Server information.
srvreadonly	CHAR(1)	Whether the server is read only.

Constraints on underlying system table

PRIMARY KEY (srvid)

SYSSOURCE system view

Each row in the SYSSOURCE system view contains the source code, if applicable, for an object listed in the SYSOBJECT system view. The underlying system table for this view is ISYSSOURCE.

Column name	Column type	Description
object_id	UN-SIGNED BIGINT	The internal ID for the object whose source code is being defined.
source	LONG VAR-CHAR	This column contains the original source code for the object if the preserve_source_format database option is On when the object was created. For more information, see "preserve_source_format option [database]" [*SQL Anywhere Server - Database Administration*].

Constraints on underlying system table

PRIMARY KEY (object_id)

FOREIGN KEY (object_id) references SYS.ISYSOBJECT (object_id) MATCH UNIQUE FULL

SYSSQLSERVERTYPE system view

The SYSSQLSERVERTYPE system view contains information relating to compatibility with Adaptive Server Enterprise. The underlying system table for this view is ISYSSQLSERVERTYPE.

Column name	Column type	Description
ss_user_type	SMALLINT	The Adaptive Server Enterprise user type.
ss_domain_id	SMALLINT	The Adaptive Server Enterprise domain id.
ss_type_name	VARCHAR (30)	The Adaptive Server Enterprise type name.
primary_sa_domain_id	SMALLINT	The corresponding SQL Anywhere primary domain id.
primary_sa_user_type	SMALLINT	The corresponding SQL Anywhere primary user type.

Constraints on underlying system table

PRIMARY KEY (ss_user_type)

SYSSUBSCRIPTION system view

Each row in the SYSSUBSCRIPTION system view describes a subscription from one user ID (which must have REMOTE permissions) to one publication. The underlying system table for this view is ISYSSUBSCRIPTION.

Column name	Column type	Description
publication_id	UNSIGNED INT	The identifier for the publication to which the user ID is subscribed.
user_id	UNSIGNED INT	The ID of the user who is subscribed to the publication.
subscribe_by	CHAR(128)	The value of the SUBSCRIBE BY expression, if any, for the subscription.
created	UNSIGNED BIGINT	The offset in the transaction log at which the subscription was created.
started	UNSIGNED BIGINT	The offset in the transaction log at which the subscription was started.

Constraints on underlying system table

PRIMARY KEY (publication_id, user_id, subscribe_by)

FOREIGN KEY (publication_id) references SYS.ISYSPUBLICATION (publication_id)

FOREIGN KEY (user_id) references SYS.ISYSUSER (user_id)

SYSSYNC system view

The SYSSYNC system view contains information relating to MobiLink synchronization. Some columns in this view contain potentially sensitive data. For that reason, access to this view is restricted to users with DBA authority. The SYSSYNC2 view provides public access to the data in this view except for the potentially sensitive columns. The underlying system table for this view is ISYSSYNC.

Column name	Column type	Description
sync_id	UNSIGNED INT	A number that uniquely identifies the row.
type	CHAR(1)	The type of synchronization object: D means definition, T means template, and S means site.
publication_id	UNSIGNED INT	A publication_id found in the SYSPUBLICATION system view.
progress	UNSIGNED BIGINT	The log offset of the last successful upload.
site_name	CHAR(128)	A MobiLink user name.
"option"	LONG VARCHAR	Synchronization options.
server_connect	LONG VARCHAR	The address or URL of the MobiLink server.

Column name	Column type	Description
server_conn_type	LONG VARCHAR	The communication protocol, such as TCP/IP, to use when synchronizing.
last_download_time	TIMESTAMP	Indicates the last time a download stream was received from the MobiLink server.
last_upload_time	TIMESTAMP	Indicates the last time (measured at the MobiLink server) that information was successfully uploaded. The default is jan-1-1900.
created	UNSIGNED BIGINT	The log offset at which the subscription was created.
log_sent	UNSIGNED BIGINT	The log progress up to which information has been uploaded. It is not necessary that an acknowledgement of the upload be received for the entry in this column to be updated.
generation_number	INTEGER	For file-base downloads, the last generation number received for this subscription. The default is 0.
extended_state	VARCHAR(1024)	For internal use only.

Constraints on underlying system table

PRIMARY KEY (sync_id)

FOREIGN KEY (publication_id) references SYS.ISYSPUBLICATION (publication_id)

SYSSYNCSCRIPT system view

Each row in the SYSSYNCSCRIPT system view identifies a stored procedure for MobiLink scripted upload. This view is almost identical to the SYSSYNCSCRIPTS view, except that the values in this view are in their raw format. To see them in their human-readable format, see "SYSSYNCSCRIPTS consolidated view" on page 1056.

For information about which publications use scripted upload, see "SYSPUBLICATION system view" on page 1019.

For information about stored procedure definitions, see "SYSPROCEDURE system view" on page 1015.

The underlying system table for this view is ISYSSYNCSCRIPT.

Column name	Column type	Description
pub_object_id	UNSIGNED BIGINT	The object ID of the publication to which the script belongs.
table_object_id	UNSIGNED BIGINT	The object ID of the table to which the script applies.

Column name	Column type	Description
type	UNSIGNED INT	The type of upload procedure.
proc_object_id	UNSIGNED BIGINT	The object ID of the stored procedure to use for the publication.

Constraints on underlying system table

PRIMARY KEY (pub_object_id, table_object_id, type)

FOREIGN KEY (pub_object_id) references SYS.ISYSOBJECT (object_id)

FOREIGN KEY (table_object_id) references SYS.ISYSOBJECT (object_id)

FOREIGN KEY (proc_object_id) references SYS.ISYSOBJECT (object_id)

See also

- "Scripted upload" [*MobiLink - Client Administration*]
- "SYSPUBLICATION system view" on page 1019
- "SYSPROCEDURE system view" on page 1015
- "SYSSYNCSCRIPTS consolidated view" on page 1056

SYSTAB system view

Each row of the SYSTAB system view describes one table or view in the database. Additional information for views can be found in the SYSVIEW system view. The underlying system table for this view is ISYSTAB.

Column name	Column type	Description
table_id	UNSIGNED INT	Each table is assigned a unique number (the table number).
dbspace_id	SMALLINT	A value indicating which dbspace contains the table.
file_id	SMALLINT	DEPRECATED. This column is present in SYSVIEW, but not in the underlying system table ISYSTAB. The contents of this column is the same as dbspace_id and is provided for compatibility. Use dbspace_id instead.
count	UNSIGNED BIGINT	The number of rows in the table or materialized view. This value is updated during each successful checkpoint. This number is used by SQL Anywhere when optimizing database access. The count is always 0 for a non-materialized view or remote table.
creator	UNSIGNED INT	The user number of the owner of the table or view.
table_page_count	INTEGER	The total number of main pages used by the underlying table.

Column name	Column type	Description
ext_page_count	INTEGER	The total number of extension pages used by the underlying table.
commit_action	INTEGER	For global temporary tables, 0 indicates that the ON COMMIT PRESERVE ROWS clause was specified when the table was created, 1 indicates that the ON COMMIT DELETE ROWS clause was specified when the table was created (the default behavior for temporary tables), and 3 indicates that the NOT TRANSACTIONAL clause was specified when the table was created. For non-temporary tables, commit_action is always 0.
share_type	INTEGER	For global temporary tables, 4 indicates that the SHARE BY ALL clause was specified when the table was created, and 5 indicates that the SHARE BY ALL clause was *not* specified when the table was created. For non-temporary tables, share_type is always 5 because the SHARE BY ALL clause cannot be specified when creating non-temporary tables.
object_id	UNSIGNED BI-GINT	The object ID of the table.
last_modified_at	TIMESTAMP	The time at which the data in the table was last modified. This column is only updated at checkpoint time.
last_modified_tsn	UNSIGNED BI-GINT	A sequence number assigned to the transaction that modified the table.
table_name	CHAR(128)	The name of the table or view. One creator cannot have two tables or views with the same name.
table_type	TINYINT	The type of table or view. Values include: ● 1 - Base table ● 2 - Materialized view ● 3 - Global temporary table ● 4 - Local temporary table ● 5 - Text index base table ● 6 - Text index global temp table ● 21 - View
replicate	CHAR(1)	A value indicating whether the underlying table is a primary data source in a Replication Server installation.

Column name	Column type	Description
server_type	TINYINT	The location of the data for the underlying table. Values include: ● 1 - Local server (SQL Anywhere) ● 2 - Remote server
tab_page_list	LONG VARBIT	For internal use only. The set of pages that contain information for the table, expressed as a bitmap.
ext_page_list	LONG VARBIT	For internal use only. The set of pages that contain row extensions and large object (LOB) pages for the table, expressed as a bitmap.
pct_free	UNSIGNED INT	The PCT_FREE specification for the table, if one has been specified; otherwise, NULL.
clustered_index_id	UNSIGNED INT	The ID of the clustered index for the table. If none of the indexes are clustered, then this field is NULL.
encrypted	CHAR(1)	Whether the table or materialized view is encrypted.
file_id	SMALLINT	
table_type_str	CHAR(9)	Readable value for table_type. Values include: ● BASE - Base table ● MAT VIEW - Materialized view ● GBL TEMP - Global temporary table ● VIEW - View

Constraints on underlying system table

FOREIGN KEY (dbspace_id) references SYS.ISYSDBSPACE (dbspace_id)

FOREIGN KEY (object_id) references SYS.ISYSOBJECT (object_id)

PRIMARY KEY (table_id)

FOREIGN KEY (creator) references SYS.ISYSUSER (user_id)

See also

- "SYSVIEW system view" on page 1041

SYSTABCOL system view

The SYSTABCOL system view contains one row for each column of each table and view in the database. The underlying system table for this view is ISYSTABCOL.

Column name	Column type	Description
table_id	UNSIGNED INT	The object ID of the table or view to which the column belongs.
column_id	UNSIGNED INT	The ID of the column. For each table, column numbering starts at 1.
domain_id	SMALLINT	The data type for the column, indicated by a data type number listed in the SYSDOMAIN system view.
nulls	CHAR(1)	Indicates whether NULL values are allowed in the column.
width	UNSIGNED INT	The length of a string column, the precision of numeric columns, or the number of bytes of storage for any other data type.
scale	SMALLINT	The number of digits after the decimal point for NUMERIC or DECIMAL data type columns. For string columns, a value of 1 indicates character-length semantics and 0 indicates byte-length semantics.
object_id	UNSIGNED BIGINT	The object ID of the table column.
max_identity	BIGINT	The largest value of the column, if it is an AUTOINCREMENT, IDENTITY, or GLOBAL AUTOINCREMENT column.
column_name	CHAR(128)	The name of the column.
"default"	LONG VARCHAR	The default value for the column. This value, if specified, is only used when an INSERT statement does not specify a value for the column.
user_type	SMALLINT	The data type, if the column is defined using a user-defined data type.
column_type	CHAR(1)	The type of column (C=computed column, and R=other columns).
compressed	TINYINT	Whether this column is stored in a compressed format.
collect_stats	TINYINT	Whether the system automatically collects and updates statistics on this column.
inline_max	SMALLINT	The maximum number of bytes of a BLOB to store in a row. A NULL value indicates that either the default value has been applied, or that the column is not a character or binary type. A non-NULL inline_max value corresponds to the INLINE value specified for the column using the CREATE TABLE or ALTER TABLE statement. For more information about the INLINE clause, see "CREATE TABLE statement" on page 523.

Column name	Column type	Description
inline_long	SMALLINT	The number of duplicate bytes of a BLOB to store in a row if the BLOB size exceeds the inline_max value. A NULL value indicates that either the default value has been applied, or that the column is not a character or binary type. A non-NULL inline_long value corresponds to the PREFIX value specified for the column using the CREATE TABLE or ALTER TABLE statement. For more information about the PREFIX clause, see "CREATE TABLE statement" on page 523.
lob_index	TINYINT	Whether to build indexes on BLOB values in the column that exceed an internal threshold size (approximately eight database pages). A NULL value indicates either that the default is applied, or that the column is not BLOB type. A value of 1 indicates that indexes will be built. A value of 0 indicates that no indexes will be built. A non-NULL lob_index value corresponds to whether INDEX or NO INDEX was specified for the column using the CREATE TABLE or ALTER TABLE statement. For more information about the [NO] INDEX clause, see "CREATE TABLE statement" on page 523.

Constraints on underlying system table

PRIMARY KEY (table_id, column_id)

FOREIGN KEY (table_id) references SYS.ISYSTAB (table_id)

FOREIGN KEY (domain_id) references SYS.ISYSDOMAIN (domain_id)

FOREIGN KEY (object_id) references SYS.ISYSOBJECT (object_id) MATCH UNIQUE FULL

FOREIGN KEY (user_type) references SYS.ISYSUSERTYPE (type_id)

SYSTABLEPERM system view

Permissions given by the GRANT statement are stored in the SYSTABLEPERM system view. Each row in this view corresponds to one table, one user ID granting the permission (grantor) and one user ID granted the permission (grantee). The underlying system table for this view is ISYSTABLEPERM.

Column name	Column type	Description
stable_id	UNSIGNED INT	The table number of the table or view to which the permissions apply.
grantee	UNSIGNED INT	The user number of the user ID receiving the permission.
grantor	UNSIGNED INT	The user number of the user ID granting the permission.

Column name	Column type	Description
selectauth	CHAR(1)	Indicates whether SELECT permission has been granted. Possible values are Y, N, or G. See the Remarks area below for further information on what these values mean.
insertauth	CHAR(1)	Indicates whether INSERT permission has been granted. Possible values are Y, N, or G. See the Remarks area below for further information on what these values mean.
deleteauth	CHAR(1)	Indicates whether DELETE permission has been granted. Possible values are Y, N, or G. See the Remarks area below for further information on what these values mean.
updateauth	CHAR(1)	Indicates whether UPDATE permission has been granted for all columns in the table. Possible values are Y, N, or G. See the Remarks area below for further information on what these values mean.
updatecols	CHAR(1)	Indicates whether UPDATE permission has only been granted for some of the columns in the underlying table. If updatecols has the value Y, there will be one or more rows in the SYSCOLPERM system view granting update permission for the columns.
alterauth	CHAR(1)	Indicates whether ALTER permission has been granted. Possible values are Y, N, or G. See the Remarks area below for further information on what these values mean.
referenceauth	CHAR(1)	Indicates whether REFERENCE permission has been granted. Possible values are Y, N, or G. See the Remarks area below for further information on what these values mean.

Remarks

There are several types of permission that can be granted. Each permission can have one of the following three values.

- **N** No, the grantee has not been granted this permission by the grantor.
- **Y** Yes, the grantee has been given this permission by the grantor.
- **G** The grantee has been given this permission and can grant the same permission to another user. See "GRANT statement" on page 627.

Permissions
The grantee might have been given permission for the same table by another grantor. If so, this information would be found in a different row of the SYSTABLEPERM system view.

Constraints on underlying system table

PRIMARY KEY (stable_id, grantee, grantor)

FOREIGN KEY (stable_id) references SYS.ISYSTAB (table_id)

FOREIGN KEY (grantor) references SYS.ISYSUSER (user_id)

FOREIGN KEY (grantee) references SYS.ISYSUSER (user_id)

SYSTEXTCONFIG system view

Each row in the SYSTEXTCONFIG system view describes one text configuration object, for use with the full text search feature. The underlying system table for this view is ISYSTEXTCONFIG.

For more information about what each configuration setting means, see "Text configuration object settings" [*SQL Anywhere Server - SQL Usage*].

For more information about the full text search feature, see "Full text searching" [*SQL Anywhere Server - SQL Usage*].

Column name	Column type	Description
object_id	UNSIGNED BIGINT	The object ID for the text configuration object.
creator	UNSIGNED INT	The creator of the text configuration object.
term_breaker	TINYINT	The algorithm used to separate a string into terms or words. Values are 0 for GENERIC and 1 for NGRAM. With GENERIC, any string of one or more alphanumeric characters separated by non-alphanumerics are treated as a term. NGRAM is for approximate matching or for documents that do not use a whitespace to separate terms.
stemmer	TINYINT	For internal use only.
min_term_length	TINYINT	The minimum length, in characters, allowed for a term. Terms that are shorter than min_term_length are ignored. The MINIMUM TERM LENGTH setting is only meaningful for the GENERIC term breaker. For NGRAM text indexes, the setting is ignored.
max_term_length	TINYINT	For GENERIC text indexes, the maximum length, in characters, allowed for a term. Terms that are longer than max_term_length are ignored. For NGRAM text indexes, this is the length of the n-grams into which terms are broken.

Column name	Column type	Description
collation	CHAR(128)	For internal use only.
text_config_name	CHAR(128)	The name of the text configuration object.
prefilter	CHAR(128)	For internal use only.
postfilter	CHAR(128)	For internal use only.
char_stoplist	LONG VAR-CHAR	Terms to ignore when performing a full text search on CHAR columns. These terms are also omitted from text indexes. This column is used when the text configuration object is created from default_char.
nchar_stoplist	LONG NVARCHAR	Terms to ignore when performing a full text search on NCHAR columns. These terms are also omitted from text indexes. This column is used when the text configuration object is created from default_nchar.

Constraints on underlying system table

PRIMARY KEY (object_id)

FOREIGN KEY (object_id) references SYS.ISYSOBJECT (object_id) MATCH UNIQUE FULL

FOREIGN KEY (creator) references SYS.ISYSUSER (user_id)

SYSTEXTIDX system view

Each row in the SYSTEXTIDX system view describes one text index. The underlying system table for this view is ISYSTEXTIDX.

For more information about the full text search feature, see "Full text searching" [*SQL Anywhere Server - SQL Usage*].

Column name	Column type	Description
index_id	UNSIGNED BIGINT	The object ID of the text index in SYSIDX.
sequence	UNSIGNED INT	For internal use only.
status	UNSIGNED INT	For internal use only.
text_config	UNSIGNED BIGINT	The object ID of the text configuration object in SYSTEXT-CONFIG.
next_handle	UNSIGNED INT	For internal use only.

Column name	Column type	Description
last_handle	UNSIGNED INT	For internal use only.
deleted_length	UNSIGNED BIGINT	The total size of deleted indexed values in the text index.
pending_length	UNSIGNED BIGINT	The total size of indexed values that will be added to the text index at the next refresh.
refresh_type	TINYINT	The type of refresh. One of: ● 1 - MANUAL ● 2 - AUTO ● 3 - IMMEDIATE
refresh_interval	UNSIGNED INT	The AUTO REFRESH interval, in minutes.
last_refresh	TIMESTAMP	The time of the last refresh.

Constraints on underlying system table

PRIMARY KEY (index_id, sequence)

FOREIGN KEY (index_id) references SYS.ISYSOBJECT (object_id)

FOREIGN KEY (text_config) references SYS.ISYSTEXTCONFIG (object_id)

SYSTEXTIDXTAB system view

Each row in the SYSTEXTIDXTAB system view describes a generated table that is part of a text index. The underlying system table for this view is ISYSTEXTIDXTAB.

For more information about the full text search feature, see "Full text searching" [*SQL Anywhere Server - SQL Usage*].

Column name	Column type	Description
index_id	UNSIGNED BIGINT	For internal use only.
sequence	UNSIGNED INT	For internal use only.
table_type	UNSIGNED INT	For internal use only.
table_id	UNSIGNED INT	For internal use only.

Constraints on underlying system table

PRIMARY KEY (index_id, sequence, table_type)

FOREIGN KEY (index_id, sequence) references SYS.ISYSTEXTIDX (index_id, sequence)

FOREIGN KEY (table_id) references SYS.ISYSTAB (table_id)

SYSTRIGGER system view

Each row in the SYSTRIGGER system view describes one trigger in the database. This view also contains triggers that are automatically created for foreign key definitions which have a referential triggered action (such as ON DELETE CASCADE). The underlying system table for this view is ISYSTRIGGER.

Column name	Column type	Description
trigger_id	UNSIGNED INT	Each trigger is assigned a unique number (the trigger number).
table_id	UNSIGNED INT	The table ID of the table to which this trigger belongs.
object_id	UNSIGNED BIGINT	The event or events that cause the trigger to fire. This single-character value corresponds to the trigger event that was specified when the trigger was created. ● A - INSERT, DELETE ● B - INSERT, UPDATE ● C - UPDATE COLUMNS ● D - DELETE ● E - DELETE, UPDATE ● I - INSERT ● U - UPDATE ● M - INSERT, DELETE, UPDATE
event	CHAR(1)	The time at which the trigger will fire. This single-character value corresponds to the trigger time that was specified when the trigger was created.
trigger_time	CHAR(1)	● A - AFTER (row-level trigger) ● B - BEFORE (row-level trigger) ● I - INSTEAD OF (row-level trigger) ● K - INSTEAD OF (statement-level trigger) ● R - RESOLVE ● S - AFTER (statement-level trigger)
trigger_order	SMALLINT	The order in which the trigger will fire. This determines the order that triggers are fired when there are triggers of the same type (insert, update, or delete) that fire at the same time (before or after).
foreign_table_id	UNSIGNED INT	The table number of the table containing a foreign key definition which has a referential triggered action (such as ON DELETE CASCADE).

Column name	Column type	Description
foreign_key_id	UNSIGNED INT	The foreign key number of the foreign key for the table referenced by foreign_table_id.
referential_action	CHAR(1)	The action defined by a foreign key. This single-character value corresponds to the action that was specified when the foreign key was created. ● C - CASCADE ● D - SET DEFAULT ● N - SET NULL ● R - RESTRICT
trigger_name	CHAR(128)	The name of the trigger. One table cannot have two triggers with the same name.
trigger_defn	LONG VARCHAR	The command that was used to create the trigger.
remarks	LONG VARCHAR	Remarks about the trigger. This value is stored in the ISYSREMARK system table.
source	LONG VARCHAR	The SQL source for the trigger. This value is stored in the ISYSSOURCE system table.

Constraints on underlying system table

PRIMARY KEY (trigger_id)

FOREIGN KEY (object_id) references SYS.ISYSOBJECT (object_id) MATCH UNIQUE FULL

FOREIGN KEY (table_id) references SYS.ISYSTAB (table_id)

FOREIGN KEY fkey_index (foreign_table_id, foreign_key_id) references SYS.ISYSIDX (table_id, index_id)

SYSTYPEMAP system view

The SYSTYPEMAP system view contains the compatibility mapping values for entries in the SYSSQLSERVERTYPE system view. The underlying system table for this view is ISYSTYPEMAP.

Column name	Column type	Description
ss_user_type	SMALLINT	Contains the Adaptive Server Enterprise user type.
sa_domain_id	SMALLINT	Contains the corresponding SQL Anywhere domain_id.

Column name	Column type	Description
sa_user_type	SMALLINT	Contains the corresponding SQL Anywhere user type.
nullable	CHAR(1)	Whether the type allows NULL values.

Constraints on underlying system table

FOREIGN KEY (sa_domain_id) references SYS.ISYSDOMAIN (domain_id)

SYSUSER system view

Each row in the SYSUSER system view describes a user in the system. The underlying system table for this view is ISYSUSER.

Column name	Column type	Description
user_id	UNSIGNED INT	A unique identifier for the user assigned to the login policy.
object_id	UNSIGNED BIGINT	A unique identifier for the user in the database.
user_name	CHAR(128)	The login name for the user.
password	BINARY(128)	The password for the user.
login_policy_id	UNSIGNED BIGINT	A unique identifier for the login policy.
expired_password_on_login	TINYINT	Indicates if the user's password expires at the next login.
password_creation_time	TIMESTAMP	The time the user's password was created.
failed_login_attempts	UNSIGNED INT	The number of times a user can fail to log in before the account is locked.
last_login_time	TIMESTAMP	The time the user last logged in.

NOTE
For databases created using SQL Anywhere version 11.0.0 and later, the underlying system table for this view is always encrypted to protect the data from unauthorized access.

Constraints on underlying system table

PRIMARY KEY (user_id)

FOREIGN KEY (object_id) references SYS.ISYSOBJECT (object_id) MATCH UNIQUE FULL

FOREIGN KEY (login_policy_id) references SYS.ISYSLOGINPOLICY (login_policy_id)

See also

- "SYSLOGINPOLICY system view" on page 1010
- "SYSLOGINPOLICYOPTION system view" on page 1010

SYSUSERAUTHORITY system view

Each row of SYSUSERAUTHORITY system view describes an authority granted to one user ID. The underlying system table for this view is ISYSUSERAUTHORITY.

Column name	Column type	Description
user_id	UNSIGNED INT	The ID of the user to whom the authority belongs.
auth	VARCHAR(20)	The authority granted to the user.

Constraints on underlying system table

PRIMARY KEY (user_id, auth)

FOREIGN KEY (user_id) references SYS.ISYSUSER (user_id)

SYSUSERMESSAGE system view

Each row in the SYSUSERMESSAGE system view holds a user-defined message for an error condition. The underlying system table for this view is ISYSUSERMESSAGE.

Note
Previous versions of the catalog contained a SYSUSERMESSAGES system table. That table has been renamed to be ISYSUSERMESSAGE (without an 'S'), and is the underlying table for this view.

Column name	Column type	Description
error	INTEGER	A unique identifying number for the error condition.
uid	UNSIGNED INT	The user number that defined the message.
description	VARCHAR(255)	The message corresponding to the error condition.
langid	SMALLINT	Reserved.

Constraints on underlying system table

FOREIGN KEY (uid) references SYS.ISYSUSER (user_id)

SYSUSERTYPE system view

Each row in the SYSUSERTYPE system view holds a description of a user-defined data type. The underlying system table for this view is ISYSUSERTYPE.

Column name	Column type	Description
type_id	SMALLINT	A unique identifying number for the user-defined data type.
creator	UNSIGNED INT	The user number of the owner of the data type.
domain_id	SMALLINT	The data type on which this user defined data type is based, indicated by a data type number listed in the SYSDOMAIN system view.
nulls	CHAR(1)	Whether the user-defined data type allows nulls. Possible values are Y, N, or U. A value of U indicates that nullability is unspecified.
width	UNSIGNED INT	The length of a string column, the precision of a numeric column, or the number of bytes of storage for any other data type.
scale	SMALLINT	The number of digits after the decimal point for numeric data type columns, and zero for all other data types.
type_name	CHAR(128)	The name for the data type.
"default"	LONG VARCHAR	The default value for the data type.
"check"	LONG VARCHAR	The CHECK condition for the data type.

Constraints on underlying system table

PRIMARY KEY (type_id)

FOREIGN KEY (creator) references SYS.ISYSUSER (user_id)

FOREIGN KEY (domain_id) references SYS.ISYSDOMAIN (domain_id)

SYSVIEW system view

Each row in the SYSVIEW system view describes a view in the database. Additional information about views can also be found in the SYSTAB system view. The underlying system table for this view is ISYSVIEW.

You can also use the sa_materialized_view_info system procedure for a more readable format of the information for materialized views. See “sa_materialized_view_info system procedure” on page 899.

Column name	Column type	Description
view_object_id	UNSIGNED BIGINT	The object ID of the view.
view_def	LONG VARCHAR	The definition (query specification) of the view.
mv_build_type	TINYINT	Currently unused.
mv_refresh_type	TINYINT	The refresh type defined for the view. Possible values are IMMEDIATE and MANUAL. See "Manual and immediate materialized views" [*SQL Anywhere Server - SQL Usage*] .
mv_use_in_optimization	TINYINT	Whether the materialized view can be used during query optimization (0=cannot be used in optimization, 1=can be used in optimization). See "Enable and disable optimizer use of a materialized view" [*SQL Anywhere Server - SQL Usage*].
mv_last_refreshed_at	TIMESTAMP	Indicates the date and time that the materialized view was last refreshed.
mv_last_refreshed_tsn	UNSIGNED BIGINT	The sequence number assigned to the transaction that refreshed the materialized view.
mv_known_stale_at	TIMESTAMP	The time at which the materialized view became stale. This value corresponds to the time at which one of the underlying base tables was detected as having changed. A value of 0 indicates that the view is either fresh, or that it has become stale but the database server has not marked it as such because the view has not been used since it became stale. Use the sa_materialized_view_info system procedure to determine the status of a materialized view. See "sa_materialized_view_info system procedure" on page 899.

Remarks

When a materialized view is refreshed with SNAPSHOT isolation, mv_last_refreshed_at and mv_last_refreshed_tsn refer to the earliest transaction that modified any row used during the computation of the materialized view contents.

Constraints on underlying system table

PRIMARY KEY (view_object_id)

FOREIGN KEY (view_object_id) references SYS.ISYSOBJECT (object_id) MATCH UNIQUE FULL

See also

- "SYSTAB system view" on page 1028
- "CREATE MATERIALIZED VIEW statement" on page 477
- "REFRESH MATERIALIZED VIEW statement" on page 703
- "CREATE VIEW statement" on page 547

SYSWEBSERVICE system view

Each row in the SYSWEBSERVICE system view holds a description of a web service. The underlying system table for this view is ISYSWEBSERVICE.

Column name	Column type	Description
service_id	UNSIGNED INT	A unique identifying number for the web service.
object_id	UNSIGNED BIGINT	The ID of the webservice.
service_name	CHAR(128)	The name assigned to the web service.
service_type	VARCHAR(40)	The type of the service; for example, RAW, HTTP, XML, SOAP, or DISH.
auth_required	CHAR(1)	Whether all requests must contain a valid user name and password.
secure_required	CHAR(1)	Whether insecure connections, such as HTTP, are to be accepted, or only secure connections, such as HTTPS.
url_path	CHAR(1)	Controls the interpretation of URLs.
user_id	UNSIGNED INT	If authentication is enabled, identifies the user, or group of users, that have permission to use the service. If authentication is disabled, specifies the account to use when processing requests.
parameter	LONG VARCHAR	A prefix that identifies the SOAP services to be included in a DISH service.
statement	LONG VARCHAR	A SQL statement that is always executed in response to a request. If NULL, arbitrary statements contained in each request are executed instead. Ignored for services of type DISH.
remarks	LONG VARCHAR	Remarks about the webservice. This value is stored in the ISYSREMARK system table.
enabled	CHAR(1)	

Constraints on underlying system table

PRIMARY KEY (service_id)

FOREIGN KEY (object_id) references SYS.ISYSOBJECT (object_id) MATCH UNIQUE FULL

Consolidated views

Consolidated views provide data in a form more frequently required by users. For example, consolidated views often provide commonly-needed joins. Consolidated views differ from system views in that they are not just a straight forward view of raw data in a underlying system table(s). For example, many of the columns in the system views are unintelligible ID values, whereas in the consolidated views, they are readable names.

SYSARTICLECOLS consolidated view

Each row in the SYSARTICLECOLS view identifies a column in an article.

The tables and columns that make up this view are provided in the SQL statement below. To learn more about a particular table or column, use the links provided beneath the view definition.

```
ALTER VIEW "SYS"."SYSARTICLECOLS"
  as select p.publication_name,t.table_name,c.column_name
    from SYS.ISYSARTICLECOL as ac
      join SYS.ISYSPUBLICATION as p on p.publication_id = ac.publication_id
      join SYS.ISYSTAB as t on t.table_id = ac.table_id
      join SYS.ISYSTABCOL as c on c.table_id = ac.table_id
      and c.column_id = ac.column_id;
```

See also

- "SYSARTICLECOL system view" on page 991
- "SYSPUBLICATION system view" on page 1019
- "SYSTAB system view" on page 1028
- "SYSTABCOL system view" on page 1030

SYSARTICLES consolidated view

Each row in the SYSARTICLES view describes an article in a publication.

The tables and columns that make up this view are provided in the SQL statement below. To learn more about a particular table or column, use the links provided beneath the view definition.

```
ALTER VIEW "SYS"."SYSARTICLES"
  as select u1.user_name as publication_owner,p.publication_name,
    u2.user_name as table_owner,t.table_name,
    a.where_expr,a.subscribe_by_expr,a.alias
    from SYS.ISYSARTICLE as a
      join SYS.ISYSPUBLICATION as p on(a.publication_id = p.publication_id)
      join SYS.ISYSTAB as t on(a.table_id = t.table_id)
      join SYS.ISYSUSER as u1 on(p.creator = u1.user_id)
      join SYS.ISYSUSER as u2 on(t.creator = u2.user_id);
```

See also

- “SYSARTICLE system view” on page 990
- “SYSPUBLICATION system view” on page 1019
- “SYSTAB system view” on page 1028
- “SYSUSER system view” on page 1039

SYSCAPABILITIES consolidated view

Each row in the SYSCAPABILITIES view describes a capability. This view gets its data from the ISYSCAPABILITY and ISYSCAPABILITYNAME system tables.

The tables and columns that make up this view are provided in the SQL statement below. To learn more about a particular table or column, use the links provided beneath the view definition.

```
ALTER VIEW "SYS"."SYSCAPABILITIES"
  as select ISYSCAPABILITY.capid,
            ISYSCAPABILITY.srvid,
            property('RemoteCapability', ISYSCAPABILITY.capid) as capname,
            ISYSCAPABILITY.capvalue
    from SYS.ISYSCAPABILITY;
```

See also

- “SYSCAPABILITY system view” on page 991
- “SYSCAPABILITYNAME system view” on page 992

SYSCATALOG consolidated view

Each row in the SYSCATALOG view describes a system table.

The tables and columns that make up this view are provided in the SQL statement below. To learn more about a particular table or column, use the links provided beneath the view definition.

```
ALTER VIEW "SYS"."SYSCATALOG"( creator,
  tname,dbspacename,tabletype,ncols,primary_key,"check",
  remarks )
  as select u.user_name,tab.table_name,dbs.dbspace_name,
    if tab.table_type_str = 'BASE' then 'TABLE' else tab.table_type_str
endif,
    (select count() from SYS.ISYSTABCOL
      where ISYSTABCOL.table_id = tab.table_id),
    if ix.index_id is null then 'N' else 'Y' endif,
    null,
    rmk.remarks
    from SYS.SYSTAB as tab
      join SYS.ISYSDBSPACE as dbs on(tab.dbspace_id = dbs.dbspace_id)
      join SYS.ISYSUSER as u on u.user_id = tab.creator
      left outer join SYS.ISYSIDX as ix on(tab.table_id = ix.table_id and
ix.index_id = 0)
      left outer join SYS.ISYSREMARK as rmk on(tab.object_id = rmk.object_id);
```

See also

- “SYSTAB system view” on page 1028
- “SYSTABCOL system view” on page 1030
- “SYSDBSPACE system view” on page 996
- “SYSUSER system view” on page 1039
- “SYSIDX system view” on page 1005
- “SYSREMARK system view” on page 1020

SYSCOLAUTH consolidated view

Each row in the SYSCOLAUTH view describes the set of privileges (UPDATE, SELECT, or REFERENCES) granted on a column. The SYSCOLAUTH view provides a user-friendly presentation of data in the “SYSCOLPERM system view” on page 992.

The tables and columns that make up this view are provided in the SQL statement below. To learn more about a particular table or column, use the links provided beneath the view definition.

```
ALTER VIEW "SYS"."SYSCOLAUTH"( grantor,grantee,creator,tname,colname,
  privilege_type,is_grantable )
  as select u1.user_name,u2.user_name,u3.user_name,tab.table_name,
    col.column_name,cp.privilege_type,cp.is_grantable
    from SYS.ISYSCOLPERM as cp
      join SYS.ISYSUSER as u1 on u1.user_id = cp.grantor
      join SYS.ISYSUSER as u2 on u2.user_id = cp.grantee
      join SYS.ISYSTAB as tab on tab.table_id = cp.table_id
      join SYS.ISYSUSER as u3 on u3.user_id = tab.creator
      join SYS.ISYSTABCOL as col on col.table_id = cp.table_id
      and col.column_id = cp.column_id;
```

See also

- “SYSCOLPERM system view” on page 992
- “SYSTABCOL system view” on page 1030
- “SYSUSER system view” on page 1039
- “SYSTAB system view” on page 1028

SYSCOLSTATS consolidated view

The SYSCOLSTATS view contains the column statistics that are stored as histograms and used by the optimizer.

The tables and columns that make up this view are provided in the SQL statement below. To learn more about a particular table or column, use the links provided beneath the view definition.

```
ALTER VIEW "SYS"."SYSCOLSTATS"
  as select u.user_name,t.table_name,c.column_name,
    s.format_id,s.update_time,s.density,s.max_steps,
    s.actual_steps,s.step_values,s.frequencies
    from SYS.ISYSCOLSTAT as s
      join SYS.ISYSTABCOL as c on(s.table_id = c.table_id
      and s.column_id = c.column_id)
```

```
        join SYS.ISYSTAB as t on(t.table_id = c.table_id)
        join SYS.ISYSUSER as u on(u.user_id = t.creator)
```

See also

- "SYSCOLSTAT system view" on page 993
- "SYSTABCOL system view" on page 1030
- "SYSTAB system view" on page 1028
- "SYSUSER system view" on page 1039

SYSCOLUMNS consolidated view

Each row in the SYSCOLUMNS view describes one column of each table and view in the catalog.

The tables and columns that make up this view are provided in the SQL statement below. To learn more about a particular table or column, use the links provided beneath the view definition.

```
ALTER VIEW "SYS"."SYSCOLUMNS"( creator,cname,tname,coltype,nulls,length,
  syslength,in_primary_key,colno,default_value,
  column_kind,remarks )
  as select u.user_name,col.column_name,tab.table_name,dom.domain_name,
   col.nulls,col.width,col.scale,if ixcol.sequence is null then 'N' else 'Y'
endif,col.column_id,
    col."default",col.column_type,rmk.remarks
    from SYS.ISYSTABCOL as col
     left outer join SYS.ISYSIDXCOL as ixcol on(col.table_id = ixcol.table_id
and col.column_id = ixcol.column_id and ixcol.index_id = 0)
      join SYS.ISYSTAB as tab on(tab.table_id = col.table_id)
      join SYS.ISYSDOMAIN as dom on(dom.domain_id = col.domain_id)
      join SYS.ISYSUSER as u on u.user_id = tab.creator
      left outer join SYS.ISYSREMARK as rmk on(col.object_id = rmk.object_id)
```

See also

- "SYSTABCOL system view" on page 1030
- "SYSIDXCOL system view" on page 1006
- "SYSTAB system view" on page 1028
- "SYSDOMAIN system view" on page 997
- "SYSUSER system view" on page 1039
- "SYSREMARK system view" on page 1020

SYSFOREIGNKEYS consolidated view

Each row in the SYSFOREIGNKEYS view describes one foreign key for each table in the catalog.

The tables and columns that make up this view are provided in the SQL statement below. To learn more about a particular table or column, use the links provided beneath the view definition.

```
ALTER VIEW "SYS"."SYSFOREIGNKEYS"( foreign_creator,
  foreign_tname,
  primary_creator,primary_tname,role,columns )
  as select fk_up.user_name,fk_tab.table_name,pk_up.user_name,
    pk_tab.table_name,ix.index_name,
    (select list(string(fk_col.column_name,' IS ',
            pk_col.column_name))
            from SYS.ISYSIDXCOL as fkc
              join SYS.ISYSTABCOL as fk_col on(
              fkc.table_id = fk_col.table_id
              and fkc.column_id = fk_col.column_id)
              ,SYS.ISYSTABCOL as pk_col
            where fkc.table_id = fk.foreign_table_id
            and fkc.index_id = fk.foreign_index_id
            and pk_col.table_id = fk.primary_table_id
            and pk_col.column_id = fkc.primary_column_id)
          from SYS.ISYSFKEY as fk
            join SYS.ISYSTAB as fk_tab on fk_tab.table_id = fk.foreign_table_id
            join SYS.ISYSUSER as fk_up on fk_up.user_id = fk_tab.creator
            join SYS.ISYSTAB as pk_tab on pk_tab.table_id = fk.primary_table_id
            join SYS.ISYSUSER as pk_up on pk_up.user_id = pk_tab.creator
            join SYS.ISYSIDX as ix on ix.table_id = fk.foreign_table_id
               and ix.index_id = fk.foreign_index_id
```

See also

- “SYSIDXCOL system view” on page 1006
- “SYSTABCOL system view” on page 1030
- “SYSFKEY system view” on page 1002
- “SYSTAB system view” on page 1028
- “SYSUSER system view” on page 1039
- “SYSIDX system view” on page 1005
- “SYSDOMAIN system view” on page 997
- “SYSREMARK system view” on page 1020

SYSGROUPS consolidated view

There is one row in the SYSGROUPS view for each member of each group. This view describes the many-to-many relationship between groups and members. A group may have many members, and a user may be a member of many groups.

The tables and columns that make up this view are provided in the SQL statement below. To learn more about a particular table or column, use the links provided beneath the view definition.

```
ALTER VIEW "SYS"."SYSGROUPS"( group_name,
  member_name )
  as select g.user_name,u.user_name
    from SYS.ISYSGROUP, SYS.ISYSUSER as g, SYS.ISYSUSER as u
    where ISYSGROUP.group_id = g.user_id
      and ISYSGROUP.group_member = u.user_id;
```

See also

- “SYSGROUP system view” on page 1003
- “SYSUSER system view” on page 1039

SYSINDEXES consolidated view

Each row in the SYSINDEXES view describes one index in the database. As an alternative to this view, you could also use the SYSIDX and SYSIDXCOL system views.

The tables and columns that make up this view are provided in the SQL statement below. To learn more about a particular table or column, use the links provided beneath the view definition.

```
ALTER VIEW "SYS"."SYSINDEXES"( icreator,
  iname,fname,creator,tname,indextype,
  colnames,interval,level_num )
  as select u.user_name,idx.index_name,dbs.dbspace_name,u.user_name,
    tab.table_name,
    case idx.index_category
    when 1 then 'Primary Key'
    when 2 then 'Foreign Key'
    when 3 then(
      if idx."unique" = 4 then 'Non-unique'
      else if idx."unique" = 2 then 'UNIQUE constraint'
        else 'Unique'
        endif
      endif) when 4 then 'Text Index' end,
    (select list(string(c.column_name,
      if ixc."order" = 'A' then ' ASC' else ' DESC' endif) order by
      ixc.table_id asc,ixc.index_id asc,ixc.sequence asc)
      from SYS.ISYSIDXCOL as ixc
        join SYS.ISYSTABCOL as c on(
        c.table_id = ixc.table_id
        and c.column_id = ixc.column_id)
      where ixc.index_id = idx.index_id
      and ixc.table_id = idx.table_id),
    0,0
    from SYS.ISYSTAB as tab
      join SYS.ISYSDBSPACE as dbs on(tab.dbspace_id = dbs.dbspace_id)
      join SYS.ISYSIDX as idx on(idx.table_id = tab.table_id)
      join SYS.ISYSUSER as u on u.user_id = tab.creator;
```

See also

- "SYSIDXCOL system view" on page 1006
- "SYSTABCOL system view" on page 1030
- "SYSTAB system view" on page 1028
- "SYSDBSPACE system view" on page 996
- "SYSIDX system view" on page 1005
- "SYSUSER system view" on page 1039

SYSOPTIONS consolidated view

Each row in the SYSOPTIONS view describes one option created using the SET command. Each user can have their own setting for each option. In addition, settings for the PUBLIC user define the default settings to be used for users that do not have their own setting.

The tables and columns that make up this view are provided in the SQL statement below. To learn more about a particular table or column, use the links provided beneath the view definition.

```
ALTER VIEW "SYS"."SYSOPTIONS"( user_name,"option",setting )
  as select u.user_name,opt."option",opt.setting
    from SYS.ISYSOPTION as opt
      join SYS.ISYSUSER as u on opt.user_id = u.user_id
```

See also

- "SYSOPTION system view" on page 1013
- "SYSUSER system view" on page 1039

SYSPROCAUTH consolidated view

Each row in the SYSPROCAUTH view describes a set of privileges granted on a procedure. As an alternative, you can also use the SYSPROCPERM system view.

The tables and columns that make up this view are provided in the SQL statement below. To learn more about a particular table or column, use the links provided beneath the view definition.

```
ALTER VIEW "SYS"."SYSPROCAUTH"( grantee,
  creator,procname )
  as select u1.user_name,u2.user_name,p.proc_name
    from SYS.ISYSPROCEDURE as p
      join SYS.ISYSPROCPERM as pp on(p.proc_id = pp.proc_id)
      join SYS.ISYSUSER as u1 on u1.user_id = pp.grantee
      join SYS.ISYSUSER as u2 on u2.user_id = p.creator
```

See also

- "SYSPROCEDURE system view" on page 1015
- "SYSPROCPERM system view" on page 1018
- "SYSUSER system view" on page 1039

SYSPROCPARMS consolidated view

Each row in the SYSPROCPARMS view describes a parameter to a procedure in the database.

The tables and columns that make up this view are provided in the SQL statement below. To learn more about a particular table or column, use the links provided beneath the view definition.

```
ALTER VIEW "SYS"."SYSPROCPARMS"( creator,
  procname,parmname,parm_id,parmtype,parmmode,parmdomain,
  length,scale,"default",user_type )
  as select up.user_name,p.proc_name,pp.parm_name,pp.parm_id,pp.parm_type,
    if pp.parm_mode_in = 'Y' and pp.parm_mode_out = 'N' then 'IN'
    else if pp.parm_mode_in = 'N' and pp.parm_mode_out = 'Y' then 'OUT'
      else 'INOUT'
      endif
    endif,dom.domain_name,pp.width,pp.scale,pp."default",ut.type_name
    from SYS.ISYSPROCPARM as pp
      join SYS.ISYSPROCEDURE as p on p.proc_id = pp.proc_id
      join SYS.ISYSUSER as up on up.user_id = p.creator
      join SYS.ISYSDOMAIN as dom on dom.domain_id = pp.domain_id
      left outer join SYS.ISYSUSERTYPE as ut on ut.type_id = pp.user_type;
```

See also

- “SYSPROCPARM system view” on page 1016
- “SYSPROCEDURE system view” on page 1015
- “SYSUSER system view” on page 1039
- “SYSDOMAIN system view” on page 997
- “SYSUSERTYPE system view” on page 1041

SYSPROCS consolidated view

The SYSPROCS view shows the procedure or function name, the name of its creator and any comments recorded for the procedure or function.

The tables and columns that make up this view are provided in the ALTER VIEW statement below.

```
ALTER VIEW "SYS"."SYSPROCS"( creator,
  procname,remarks )
  as select u.user_name,p.proc_name,r.remarks
    from SYS.ISYSPROCEDURE as p
      join SYS.ISYSUSER as u on u.user_id = p.creator
      left outer join SYS.ISYSREMARK as r on(p.object_id = r.object_id);
```

See also

- “SYSPROCEDURE system view” on page 1015
- “SYSUSER system view” on page 1039
- “SYSREMARK system view” on page 1020

SYSPUBLICATIONS consolidated view

Each row in the SYSPUBLICATIONS view describes a SQL Remote or MobiLink publication.

The tables and columns that make up this view are provided in the SQL statement below. To learn more about a particular table or column, use the links provided beneath the view definition.

```
ALTER VIEW "SYS"."SYSPUBLICATION"
  as select b.publication_id,
    b.object_id,
    b.creator,
    b.publication_name,
    r.remarks,
    b.type,
    b.sync_type
    from SYS.ISYSPUBLICATION as b
      left outer join SYS.ISYSREMARK as r on(b.object_id = r.object_id)
```

See also

- “SYSPUBLICATION system view” on page 1019
- “SYSREMARK system view” on page 1020

SYSREMOTEOPTION2 consolidated view

Presents, in a more readable format, the columns from SYSREMOTEOPTION and SYSREMOTEOPTIONTYPE that do not contain sensitive data.

The tables and columns that make up this view are provided in the SQL statement below. To learn more about a particular table or column, use the links provided beneath the view definition.

```
ALTER VIEW "SYS"."SYSREMOTEOPTION2"
  as select ISYSREMOTEOPTION.option_id,
    ISYSREMOTEOPTION.user_id,
    SYS.HIDE_FROM_NON_DBA(ISYSREMOTEOPTION.setting) as setting
    from SYS.ISYSREMOTEOPTION
```

See also

- "SYSREMOTEOPTION system view" on page 1020

SYSREMOTEOPTIONS consolidated view

Each row of the SYSREMOTEOPTIONS view describes the values of a SQL Remote message link parameter. Some columns in this view contain potentially sensitive data. For that reason, access to this view is restricted to users with DBA authority. The SYSREMOTEOPTION2 view provides public access to the insensitive data.

The tables and columns that make up this view are provided in the SQL statement below. To learn more about a particular table or column, use the links provided beneath the view definition.

```
ALTER VIEW "SYS"."SYSREMOTEOPTIONS"
  as select srt.type_name,
    sup.user_name,
    srot."option",
    SYS.HIDE_FROM_NON_DBA(sro.setting) as setting
    from SYS.ISYSREMOTETYPE as srt
      ,SYS.ISYSREMOTEOPTIONTYPE as srot
      ,SYS.ISYSREMOTEOPTION as sro
      ,SYS.ISYSUSER as sup
    where srt.type_id = srot.type_id
    and srot.option_id = sro.option_id
    and sro.user_id = sup.user_id
```

See also

- "SYSREMOTETYPE system view" on page 1021
- "SYSREMOTEOPTIONTYPE system view" on page 1020
- "SYSREMOTEOPTION system view" on page 1020
- "SYSUSER system view" on page 1039

SYSREMOTETYPES consolidated view

Each row of the SYSREMOTETYPES view describes one of the SQL Remote message types, including the publisher address.

The tables and columns that make up this view are provided in the SQL statement below. To learn more about a particular table or column, use the links provided beneath the view definition.

```
ALTER VIEW "SYS"."SYSREMOTETYPES"
  as select rt.type_id,rt.type_name,rt.publisher_address,rm.remarks
    from SYS.ISYSREMOTETYPE as rt
      left outer join SYS.ISYSREMARK as rm on(rt.object_id = rm.object_id)
```

See also

- "SYSREMOTETYPE system view" on page 1021
- "SYSREMARK system view" on page 1020

SYSREMOTEUSERS consolidated view

Each row of the SYSREMOTEUSERS view describes a user ID with REMOTE permissions (a subscriber), together with the status of SQL Remote messages that were sent to and from that user.

The tables and columns that make up this view are provided in the SQL statement below. To learn more about a particular table or column, use the links provided beneath the view definition.

```
ALTER VIEW "SYS"."SYSREMOTEUSERS"
  as select u.user_name,r.consolidate,t.type_name,r.address,r.frequency,
    r.send_time,
    (if r.frequency = 'A' then null else if r.frequency = 'P' then
        if r.time_sent is null then current timestamp
        else(select min(minutes(a.time_sent,60*hour(a.send_time)
            +minute(seconds(a.send_time,59))))
            from SYS.ISYSREMOTEUSER as a where a.frequency = 'P'
            and a.send_time = r.send_time)
        endif
      else if current date+r.send_time
         > coalesce(r.time_sent,current timestamp) then
          current date+r.send_time else current date+r.send_time+1 endif
      endif endif) as next_send,
    r.log_send,r.time_sent,r.log_sent,r.confirm_sent,r.send_count,
    r.resend_count,r.time_received,r.log_received,
    r.confirm_received,r.receive_count,r.rereceive_count
    from SYS.ISYSREMOTEUSER as r
      join SYS.ISYSUSER as u on(u.user_id = r.user_id)
      join SYS.ISYSREMOTETYPE as t on(t.type_id = r.type_id)
```

See also

- "SYSREMOTEUSER system view" on page 1021
- "SYSUSER system view" on page 1039
- "SYSREMOTETYPE system view" on page 1021

SYSSUBSCRIPTIONS consolidated view

Each row describes a subscription from one user ID (which must have REMOTE permissions) to one publication.

The tables and columns that make up this view are provided in the SQL statement below. To learn more about a particular table or column, use the links provided beneath the view definition.

```
ALTER VIEW "SYS"."SYSSUBSCRIPTIONS"
  as select p.publication_name,u.user_name,s.subscribe_by,s.created,
    s.started
    from SYS.ISYSSUBSCRIPTION as s
      join SYS.ISYSPUBLICATION as p on(p.publication_id = s.publication_id)
      join SYS.ISYSUSER as u on u.user_id = s.user_id
```

See also

- "SYSSUBSCRIPTION system view" on page 1025
- "SYSPUBLICATION system view" on page 1019
- "SYSUSER system view" on page 1039

SYSSYNC2 consolidated view

The SYSSYNC2 view provides public access to the data found in the SYSSYNC system view—information relating to MobiLink synchronization—without exposing potentially sensitive data.

The tables and columns that make up this view are provided in the SQL statement below. To learn more about a particular table or column, use the links provided beneath the view definition.

```
ALTER VIEW "SYS"."SYSSYNC2"
  as select ISYSSYNC.sync_id,
    ISYSSYNC.type,
    ISYSSYNC.publication_id,
    ISYSSYNC.progress,
    ISYSSYNC.site_name,
    SYS.HIDE_FROM_NON_DBA(ISYSSYNC."option") as "option",
    SYS.HIDE_FROM_NON_DBA(ISYSSYNC.server_connect) as server_connect,
    ISYSSYNC.server_conn_type,
    ISYSSYNC.last_download_time,
    ISYSSYNC.last_upload_time,
    ISYSSYNC.created,
    ISYSSYNC.log_sent,
    ISYSSYNC.generation_number,
    ISYSSYNC.extended_state
    from SYS.ISYSSYNC
```

See also

- "SYSSYNC system view" on page 1026

SYSSYNCPUBLICATIONDEFAULTS consolidated view

The SYSSYNCPUBLICATIONDEFAULTS view provides the default synchronization settings associated with publications involved in MobiLink synchronization.

The tables and columns that make up this view are provided in the SQL statement below. To learn more about a particular table or column, use the links provided beneath the view definition.

```
ALTER VIEW "SYS"."SYSSYNCPUBLICATIONDEFAULTS"
  as select s.sync_id,
        p.publication_name,
        SYS.HIDE_FROM_NON_DBA(s."option") as "option",
        SYS.HIDE_FROM_NON_DBA(s.server_connect) as server_connect,
        s.server_conn_type
        from SYS.ISYSSYNC as s join SYS.ISYSPUBLICATION as p
           on(p.publication_id = s.publication_id) where s.site_name is null
```

See also

- "SYSSYNC system view" on page 1026
- "SYSPUBLICATION system view" on page 1019

SYSSYNCS consolidated view

The SYSSYNCS view contains information relating to MobiLink synchronization. Some columns in this view contain potentially sensitive data. For that reason, access to this view is restricted to users with DBA authority.

The tables and columns that make up this view are provided in the SQL statement below. To learn more about a particular table or column, use the links provided beneath the view definition.

```
ALTER VIEW "SYS"."SYSSYNCS"
  as select p.publication_name,s.progress,s.site_name,
    SYS.HIDE_FROM_NON_DBA(s."option") as "option",
    SYS.HIDE_FROM_NON_DBA(s.server_connect) as server_connect,
    s.server_conn_type,s.last_download_time,
    s.last_upload_time,s.created,s.log_sent,s.generation_number,
    s.extended_state
    from SYS.ISYSSYNC as s
      left outer join SYS.ISYSPUBLICATION as p
      on p.publication_id = s.publication_id
```

See also

- "SYSSYNC system view" on page 1026
- "SYSPUBLICATION system view" on page 1019

SYSSYNCSCRIPTS consolidated view

Each row in the SYSSYNCSCRIPTS view identifies a stored procedure for MobiLink scripted upload. This view is almost identical to the SYSSYNCSCRIPT system view, except that the values are in human-readable format, as opposed to raw data.

```
ALTER VIEW "SYS"."SYSSYNCSCRIPTS"
  as select p.publication_name,
    t.table_name,
    case s.type
    when 0 then 'upload insert'
    when 1 then 'upload delete'
    when 2 then 'upload update'
    else 'unknown'
    end as type,
    c.proc_name
    from SYS.ISYSSYNCSCRIPT as s
      join SYS.ISYSPUBLICATION as p on p.object_id = s.pub_object_id
```

```
            join SYS.ISYSTAB as t on t.object_id = s.table_object_id
            join SYS.ISYSPROCEDURE as c on c.object_id = s.proc_object_id
```

See also

- “SYSSYNCSCRIPT system view” on page 1027
- “SYSPUBLICATION system view” on page 1019
- “SYSTAB system view” on page 1028
- “SYSPROCEDURE system view” on page 1015
- “Scripted upload” [*MobiLink - Client Administration*]

SYSSYNCSUBSCRIPTIONS consolidated view

The SYSSYNCSUBSCRIPTIONS view contains the synchronization settings associated with MobiLink synchronization subscriptions.

The tables and columns that make up this view are provided in the SQL statement below. To learn more about a particular table or column, use the links provided beneath the view definition.

```
ALTER VIEW "SYS"."SYSSYNCSUBSCRIPTIONS"
  as select s.sync_id,
    p.publication_name,
    s.progress,
    s.site_name,
    SYS.HIDE_FROM_NON_DBA(s."option") as "option",
    SYS.HIDE_FROM_NON_DBA(s.server_connect) as server_connect,
    s.server_conn_type,
    s.last_download_time,
    s.last_upload_time,
    s.created,
    s.log_sent,
    s.generation_number,
    s.extended_state
    from SYS.ISYSSYNC as s join SYS.ISYSPUBLICATION as p on(p.publication_id
= s.publication_id)
    where s.publication_id is not null and
    s.site_name is not null and exists
    (select 1 from SYS.SYSSYNCUSERS as u
      where s.site_name = u.site_name)
```

See also

- “SYSSYNC system view” on page 1026
- “SYSPUBLICATION system view” on page 1019
- “SYSSYNCUSERS consolidated view” on page 1057

SYSSYNCUSERS consolidated view

A view of synchronization settings associated with MobiLink synchronization users.

The tables and columns that make up this view are provided in the SQL statement below. To learn more about a particular table or column, use the links provided beneath the view definition.

```
ALTER VIEW "SYS"."SYSSYNCUSERS"
  as select ISYSSYNC.sync_id,
    ISYSSYNC.site_name,
    SYS.HIDE_FROM_NON_DBA(ISYSSYNC."option") as "option",
    SYS.HIDE_FROM_NON_DBA(ISYSSYNC.server_connect) as server_connect,
    ISYSSYNC.server_conn_type
    from SYS.ISYSSYNC where
    ISYSSYNC.publication_id is null
```

See also

- "SYSSYNC system view" on page 1026

SYSTABAUTH consolidated view

The SYSTABAUTH view contains information from the SYSTABLEPERM system view, but in a more readable format.

The tables and columns that make up this view are provided in the SQL statement below. To learn more about a particular table or column, use the links provided beneath the view definition.

```
ALTER VIEW "SYS"."SYSTABAUTH"( grantor,
  grantee,screator,stname,tcreator,ttname,
  selectauth,insertauth,deleteauth,
  updateauth,updatecols,alterauth,referenceauth )
  as select u1.user_name,u2.user_name,u3.user_name,tab1.table_name,
    u4.user_name,tab2.table_name,tp.selectauth,tp.insertauth,
    tp.deleteauth,tp.updateauth,tp.updatecols,tp.alterauth,
    tp.referenceauth
    from SYS.ISYSTABLEPERM as tp
      join SYS.ISYSUSER as u1 on u1.user_id = tp.grantor
      join SYS.ISYSUSER as u2 on u2.user_id = tp.grantee
      join SYS.ISYSTAB as tab1 on tab1.table_id = tp.stable_id
      join SYS.ISYSUSER as u3 on u3.user_id = tab1.creator
      join SYS.ISYSTAB as tab2 on tab2.table_id = tp.stable_id
      join SYS.ISYSUSER as u4 on u4.user_id = tab2.creator
```

See also

- "SYSTABLEPERM system view" on page 1032
- "SYSUSER system view" on page 1039
- "SYSTAB system view" on page 1028

SYSTRIGGERS consolidated view

Each row in the SYSTRIGGERS view describes one trigger in the database. This view also contains triggers that are automatically created for foreign key definitions which have a referential triggered action (such as ON DELETE CASCADE).

The tables and columns that make up this view are provided in the SQL statement below. To learn more about a particular table or column, use the links provided beneath the view definition.

```
ALTER VIEW "SYS"."SYSTRIGGERS"( owner,
  trigname,tname,event,trigtime,trigdefn )
  as select u.user_name,trig.trigger_name,tab.table_name,
          if trig.event = 'I' then 'INSERT'
          else if trig.event = 'U' then 'UPDATE'
            else if trig.event = 'C' then 'UPDATE'
              else if trig.event = 'D' then 'DELETE'
                else if trig.event = 'A' then 'INSERT,DELETE'
                  else if trig.event = 'B' then 'INSERT,UPDATE'
                    else if trig.event = 'E' then 'DELETE,UPDATE'
                      else 'INSERT,DELETE,UPDATE'
                      endif
                    endif
                  endif
                endif
              endif
            endif
          endif,if trig.trigger_time = 'B' or trig.trigger_time = 'P' then 'BEFORE'
          else if trig.trigger_time = 'A' or trig.trigger_time = 'S' then 'AFTER'
            else if trig.trigger_time = 'R' then 'RESOLVE'
              else 'INSTEAD OF'
              endif
            endif
          endif,trig.trigger_defn
          from SYS.ISYSTRIGGER as trig
            join SYS.ISYSTAB as tab on(tab.table_id = trig.table_id)
            join SYS.ISYSUSER as u on u.user_id = tab.creator where
          trig.foreign_table_id is null
```

See also

- “SYSTRIGGER system view” on page 1037
- “SYSTAB system view” on page 1028
- “SYSUSER system view” on page 1039

SYSUSEROPTIONS consolidated view

The SYSUSEROPTIONS view contains the option settings that are in effect for each user. If a user has no setting for an option, this view displays the public setting for the option.

The tables and columns that make up this view are provided in the SQL statement below. To learn more about a particular table or column, use the links provided beneath the view definition.

```
ALTER VIEW "SYS"."SYSUSEROPTIONS"( user_name,
  "option",setting )
  as select u.user_name,
    o."option",
    isnull((select s.setting
      from SYS.ISYSOPTION as s
      where s.user_id = u.user_id
      and s."option" = o."option"),
    o.setting)
    from SYS.SYSOPTIONS as o,SYS.ISYSUSER as u
    where o.user_name = 'PUBLIC'
```

See also

- “SYSOPTIONS consolidated view” on page 1050
- “SYSUSER system view” on page 1039

SYSVIEWS consolidated view

Each row of the SYSVIEWS view describes one view, including its view definition.

The tables and columns that make up this view are provided in the SQL statement below. To learn more about a particular table or column, use the links provided beneath the view definition.

```
ALTER VIEW "SYS"."SYSVIEWS"( vcreator,
  viewname,viewtext )
  as select u.user_name,t.table_name,v.view_def
    from SYS.ISYSTAB as t
      join SYS.ISYSVIEW as v on(t.object_id = v.view_object_id)
      join SYS.ISYSUSER as u on(u.user_id = t.creator)
```

See also

- "SYSTAB system view" on page 1028
- "SYSVIEW system view" on page 1041
- "SYSUSER system view" on page 1039

Compatibility views

Compatibility views are views that are provided for compatibility with pre-10.0.0 versions of SQL Anywhere. Where possible you should use system and consolidated views instead, as support may diminish for some compatibility views in future releases.

SYSCOLLATION compatibility view (deprecated)

The SYSCOLLATION compatibility view contains the collation sequence information for the database. It is obtainable via built-in functions and is not kept in the catalog. Following is definition for this view:

```
ALTER VIEW "SYS"."SYSCOLLATION"
  as select 1 as collation_id,
    DB_PROPERTY('Collation') as collation_label,
    DB_EXTENDED_PROPERTY('Collation','Description') as collation_name,
    cast(DB_EXTENDED_PROPERTY('Collation','LegacyData') as binary(1280)) as
collation_order;
```

See also

- “Database properties” [*SQL Anywhere Server - Database Administration*]
- “DB_PROPERTY function [System]” on page 185
- “DB_EXTENDED_PROPERTY function [System]” on page 181

SYSCOLLATIONMAPPINGS compatibility view (deprecated)

The SYSCOLLATIONMAPPINGS compatibility view contains only one row with the database collation mapping. It is obtainable via built-in functions and is not kept in the catalog. Following is definition for this view:

```
ALTER VIEW "SYS"."SYSCOLLATIONMAPPINGS"
  as select DB_PROPERTY('Collation') as collation_label,
    DB_EXTENDED_PROPERTY('Collation','Description') as collation_name,
    DB_PROPERTY('Charset') as cs_label,
    DB_EXTENDED_PROPERTY('Collation','ASESensitiveSortOrder') as
so_case_label,
    DB_EXTENDED_PROPERTY('Collation','ASEInsensitiveSortOrder') as
so_caseless_label,
    DB_EXTENDED_PROPERTY('Charset','java') as jdk_label;
```

See also

- “Database properties” [*SQL Anywhere Server - Database Administration*]
- “DB_PROPERTY function [System]” on page 185
- “DB_EXTENDED_PROPERTY function [System]” on page 181

SYSCOLUMN compatibility view (deprecated)

The SYSCOLUMN view is provided for compatibility with older versions of SQL Anywhere that offered a SYSCOLUMN system table. However, the previous SYSCOLUMN table has been replaced by the ISYSTABCOL system table, and its corresponding "SYSTABCOL system view" on page 1030, which you should use instead.

The tables and columns that make up this view are provided in the SQL statement below. To learn more about a particular table or column, use the links provided beneath the view definition.

```
ALTER VIEW "SYS"."SYSCOLUMN"
  as select b.table_id,
    b.column_id,
    if c.sequence is null then 'N' else 'Y' endif as pkey,
    b.domain_id,
    b.nulls,
    b.width,
    b.scale,
    b.object_id,
    b.max_identity,
    b.column_name,
    r.remarks,
    b."default",
    b.user_type,
    b.column_type
    from SYS.ISYSTABCOL as b
      left outer join SYS.ISYSREMARK as r on(b.object_id = r.object_id)
      left outer join SYS.ISYSIDXCOL as c on(b.table_id = c.table_id
      and b.column_id = c.column_id and c.index_id = 0);
```

See also

- "SYSTABCOL system view" on page 1030
- "SYSREMARK system view" on page 1020
- "SYSIDXCOL system view" on page 1006

SYSFILE compatibility view (deprecated)

Each row in the SYSFILE system view describes a dbspace for a database. Every database consists of one or more dbspaces; each dbspace corresponds to an operating system file.

SQL Anywhere automatically creates dbspaces for the main database file, temporary file, transaction log file, and transaction log mirror file. Information about the transaction log, and transaction log mirror dbspaces does not appear in the SYSFILE system view. See "Pre-defined dbspaces" [*SQL Anywhere Server - Database Administration*].

```
ALTER VIEW "SYS"."SYSFILE"
  as select b.dbfile_id as file_id,
    if b.dbspace_id = 0 and b.dbfile_id = 0 then
      db_property('File')
    else
      if b.dbspace_id = 15 and b.dbfile_id = 15 then
        db_property('TempFileName')
      else
        b.file_name
      endif
```

```
        endif as file_name,
        a.dbspace_name,
        a.store_type,
        b.lob_map,
        b.dbspace_id
        from SYS.ISYSDBSPACE as a
          join SYS.ISYSDBFILE as b on(a.dbspace_id = b.dbspace_id);
```

SYSFKCOL compatibility view (deprecated)

Each row of SYSFKCOL describes the association between a foreign column in the foreign table of a relationship and the primary column in the primary table. This view is deprecated; use the SYSIDX and SYSIDXCOL system views instead.

The tables and columns that make up this view are provided in the SQL statement below. To learn more about a particular table or column, use the links provided beneath the view definition.

```
ALTER VIEW "SYS"."SYSFKCOL"
  as select a.table_id as foreign_table_id,
    a.index_id as foreign_key_id,
    a.column_id as foreign_column_id,
    a.primary_column_id
    from SYS.ISYSIDXCOL as a
      ,SYS.ISYSIDX as b
    where a.table_id = b.table_id
    and a.index_id = b.index_id
    and b.index_category = 2;
```

See also

- “SYSIDX system view” on page 1005
- “SYSIDXCOL system view” on page 1006

SYSFOREIGNKEY compatibility view (deprecated)

The SYSFOREIGNKEY view is provided for compatibility with older versions of SQL Anywhere that offered a SYSFOREIGNKEY system table. However, the previous SYSFOREIGNKEY system table has been replaced by the ISYSFKEY system table, and its corresponding “SYSFKEY system view” on page 1002, which you should use instead.

A foreign key is a relationship between two tables—the foreign table and the primary table. Every foreign key is defined by one row in SYSFOREIGNKEY and one or more rows in SYSFKCOL. SYSFOREIGNKEY contains general information about the foreign key while SYSFKCOL identifies the columns in the foreign key and associates each column in the foreign key with a column in the primary key of the primary table.

The tables and columns that make up this view are provided in the SQL statement below. To learn more about a particular table or column, use the links provided beneath the view definition.

```
ALTER VIEW "SYS"."SYSFOREIGNKEY"
  as select b.foreign_table_id,
    b.foreign_index_id as foreign_key_id,
    a.object_id,
    b.primary_table_id,
    p.root,
        b.check_on_commit,
        b.nulls,
        a.index_name as role,
        r.remarks,
        b.primary_index_id,
        a.not_enforced as fk_not_enforced,
        10 as hash_limit
        from(SYS.ISYSIDX as a left outer join SYS.ISYSPHYSIDX as p on(a.table_id
= p.table_id and a.phys_index_id = p.phys_index_id))
          left outer join SYS.ISYSREMARK as r on(a.object_id = r.object_id)
          ,SYS.ISYSFKEY as b
        where a.table_id = b.foreign_table_id
        and a.index_id = b.foreign_index_id;
```

See also

- "SYSIDX system view" on page 1005
- "SYSPHYSIDX system view" on page 1014
- "SYSREMARK system view" on page 1020
- "SYSFKEY system view" on page 1002

SYSINDEX compatibility view (deprecated)

The SYSINDEX view is provided for compatibility with older versions of SQL Anywhere that offered a SYSINDEX system table. However, the SYSINDEX system table has been replaced by the ISYSIDX system table, and its corresponding "SYSIDX system view" on page 1005, which you should use instead.

The tables and columns that make up this view are provided in the SQL statement below. To learn more about a particular table or column, use the links provided beneath the view definition.

```
ALTER VIEW "SYS"."SYSINDEX"
  as select b.table_id,
    b.index_id,
    b.object_id,
    p.root,
    b.dbspace_id,
    case b."unique"
    when 1 then 'Y'
    when 2 then 'U'
    when 3 then 'M'
    when 4 then 'N'
    else 'I'
    end as "unique",
    t.creator,
    b.index_name,
    r.remarks,
    10 as hash_limit,
    b.dbspace_id as file_id
    from(SYS.ISYSIDX as b left outer join SYS.ISYSPHYSIDX as p on(b.table_id
= p.table_id and b.phys_index_id = p.phys_index_id))
      left outer join SYS.ISYSREMARK as r on(b.object_id = r.object_id)
      ,SYS.ISYSTAB as t
    where t.table_id = b.table_id
    and b.index_category = 3;
```

See also

- “SYSIDX system view” on page 1005
- “SYSPHYSIDX system view” on page 1014
- “SYSTABLE compatibility view (deprecated)” on page 1066
- “SYSREMARK system view” on page 1020

SYSINFO compatibility view (deprecated)

The SYSINFO view indicates the database characteristics, as defined when the database was created. It always contains only one row. This view is obtainable via built-in functions and is not kept in the catalog. Following is the definition for the SYSINFO view:

```
ALTER VIEW "SYS"."SYSINFO"( page_size,
  encryption,
  blank_padding,
  case_sensitivity,
  default_collation,
  database_version )
  as select db_property('PageSize'),
    if db_property('Encryption') <> 'None' then 'Y' else 'N' endif,
    if db_property('BlankPadding') = 'On' then 'Y' else 'N' endif,
    if db_property('CaseSensitive') = 'On' then 'Y' else 'N' endif,
    db_property('Collation'),
    null;
```

See also

- “Database properties” [*SQL Anywhere Server - Database Administration*]
- “DB_PROPERTY function [System]” on page 185
- “DB_EXTENDED_PROPERTY function [System]” on page 181

SYSIXCOL compatibility view (deprecated)

The SYSIXCOL view is provided for compatibility with older versions of SQL Anywhere that offered a SYSIXCOL system table. However, the SYSIXCOL system table has been replaced by the ISYSIDXCOL system table, and its corresponding SYSIDXCOL system view. You should switch to using the “SYSIDXCOL system view” on page 1006.

Each row of the SYSIXCOL describes a column in an index. The tables and columns that make up this view are provided in the SQL statement below. To learn more about a particular table or column, use the links provided beneath the view definition.

```
ALTER VIEW "SYS"."SYSIXCOL"
  as select a.table_id,
    a.index_id,
    a.sequence,
    a.column_id,
    a."order"
    from SYS.ISYSIDXCOL as a
      ,SYS.ISYSIDX as b
    where a.table_id = b.table_id
```

```
        and a.index_id = b.index_id
        and b.index_category = 3;
```

See also

- “SYSIDX system view” on page 1005
- “SYSIDXCOL system view” on page 1006

SYSTABLE compatibility view (deprecated)

The SYSTABLE view is provided for compatibility with older versions of SQL Anywhere that offered a SYSTABLE system table. However, the SYSTABLE system table has been replaced by the ISYSTAB system table, and its corresponding “SYSTAB system view” on page 1028, which you should use instead.

Each row of SYSTABLE view describes one table in the database.

The tables and columns that make up this view are provided in the SQL statement below. To learn more about a particular table or column, use the links provided beneath the view definition.

```
ALTER VIEW "SYS"."SYSTABLE"
  as select b.table_id,
    b.file_id,
    b.count,
    0 as first_page,
    b.commit_action as last_page,
    COALESCE(ph.root,0) as primary_root,
    b.creator,
    0 as first_ext_page,
    0 as last_ext_page,
    b.table_page_count,
    b.ext_page_count,
    b.object_id,
    b.table_name,
    b.table_type_str as table_type,
    v.view_def,
    r.remarks,
    b.replicate,
    p.existing_obj,
    p.remote_location,
    'T' as remote_objtype,
    p.srvid,
    case b.server_type
    when 1 then 'SA'
    when 2 then 'IQ'
    when 3 then 'OMNI'
    else 'INVALID'
    end as server_type,
    10 as primary_hash_limit,
    0 as page_map_start,
    s.source,
    b."encrypted"
    from SYS.SYSTAB as b
      left outer join SYS.ISYSREMARK as r on(b.object_id = r.object_id)
      left outer join SYS.ISYSSOURCE as s on(b.object_id = s.object_id)
      left outer join SYS.ISYSVIEW as v on(b.object_id = v.view_object_id)
      left outer join SYS.ISYSPROXYTAB as p on(b.object_id =
p.table_object_id)
      left outer join(SYS.ISYSIDX as i left outer join SYS.ISYSPHYSIDX as ph
on(i.table_id = ph.table_id
and i.phys_index_id = ph.phys_index_id)) on(b.table_id = i.table_id
and i.index_category = 1 and i.index_id = 0);
```

See also

- “SYSTAB system view” on page 1028
- “SYSREMARK system view” on page 1020
- “SYSSOURCE system view” on page 1024
- “SYSVIEW system view” on page 1041
- “SYSPROXYTAB system view” on page 1018
- “SYSIDX system view” on page 1005
- “SYSPHYSIDX system view” on page 1014

SYSUSERAUTH compatibility view (deprecated)

The SYSUSERAUTH view is provided for compatibility with older versions of SQL Anywhere. Use the SYSUSERAUTHORITY system view instead. See “SYSUSERAUTHORITY system view” on page 1040.

Each row of the SYSUSERAUTH view describes a user, without exposing their user_id. Instead, each user is identified by their user name. Because this view displays passwords, this view does not have PUBLIC select permission.

The tables and columns that make up this view are provided in the SQL statement below. To learn more about a particular table or column, use the links provided beneath the view definition.

```
ALTER VIEW "SYS"."SYSUSERAUTH"( name,
  password,resourceauth,dbaauth,scheduleauth,user_group )
  as select SYSUSERPERM.user_name,
            SYSUSERPERM.password,
            SYSUSERPERM.resourceauth,
            SYSUSERPERM.dbaauth,
            SYSUSERPERM.scheduleauth,
            SYSUSERPERM.user_group
    from SYS.SYSUSERPERM;
```

See also

- “SYSUSERPERM compatibility view (deprecated)” on page 1068

SYSUSERLIST compatibility view (deprecated)

The SYSUSERAUTH view is provided for compatibility with older versions of SQL Anywhere.

Each row of the SYSUSERLIST view describes a user, without exposing their user_id and password. Each user is identified by their user name.

The tables and columns that make up this view are provided in the SQL statement below. To learn more about a particular table or column, use the links provided beneath the view definition.

```
ALTER VIEW "SYS"."SYSUSERLIST"(
      name,
      resourceauth,
            dbaauth,
            scheduleauth,user_group )
        as select SYSUSERPERM.user_name,
                  SYSUSERPERM.resourceauth,
                  SYSUSERPERM.dbaauth,
                  SYSUSERPERM.scheduleauth,
                  SYSUSERPERM.user_group
          from SYS.SYSUSERPERM;
```

See also

- "SYSUSERPERM compatibility view (deprecated)" on page 1068

SYSUSERPERM compatibility view (deprecated)

This view is deprecated because it only shows the authorities and permissions available in previous versions. You should change your application to use the SYSUSERAUTHORITY system view instead.

Each row of the SYSUSERPERM view describes one user ID.

The tables and columns that make up this view are provided in the SQL statement below. To learn more about a particular table or column, use the links provided beneath the view definition.

```
ALTER VIEW "SYS"."SYSUSERPERM"
  as select b.user_id,
    b.object_id,
    b.user_name,
    b.password,
    if exists(select * from SYS.ISYSUSERAUTHORITY
      where ISYSUSERAUTHORITY.user_id = b.user_id and ISYSUSERAUTHORITY.auth
= 'RESOURCE') then
      'Y' else 'N' endif as resourceauth,
    if exists(select * from SYS.ISYSUSERAUTHORITY
      where ISYSUSERAUTHORITY.user_id = b.user_id and ISYSUSERAUTHORITY.auth
= 'DBA') then
      'Y' else 'N' endif as dbaauth,
    'N' as scheduleauth,
    if exists(select * from SYS.ISYSUSERAUTHORITY
      where ISYSUSERAUTHORITY.user_id = b.user_id and ISYSUSERAUTHORITY.auth
= 'PUBLISH') then
      'Y' else 'N' endif as publishauth,
    if exists(select * from SYS.ISYSUSERAUTHORITY
      where ISYSUSERAUTHORITY.user_id = b.user_id and ISYSUSERAUTHORITY.auth
= 'REMOTE DBA') then
      'Y' else 'N' endif as remotedbaauth,
    if exists(select * from SYS.ISYSUSERAUTHORITY
      where ISYSUSERAUTHORITY.user_id = b.user_id and ISYSUSERAUTHORITY.auth
= 'GROUP') then
      'Y' else 'N' endif as user_group,
    r.remarks
    from SYS.ISYSUSER as b
      left outer join SYS.ISYSREMARK as r on(b.object_id = r.object_id);
```

See also

- "SYSUSERAUTHORITY system view" on page 1040
- "SYSUSER system view" on page 1039
- "SYSREMARK system view" on page 1020

SYSUSERPERMS compatibility view (deprecated)

This view is deprecated because it only shows the authorities and permissions available in previous versions. You should change your application to use the SYSUSERAUTHORITY system view instead.

Similar to the SYSUSERPERM view, each row of the SYSUSERPERMS view describes one user ID. However, password information is not included. All users are allowed to read from this view.

The tables and columns that make up this view are provided in the SQL statement below. To learn more about a particular table or column, use the links provided beneath the view definition.

```
ALTER VIEW "SYS"."SYSUSERPERMS"
  as select SYSUSERPERM.user_id,
   SYSUSERPERM.user_name,
   SYSUSERPERM.resourceauth,
   SYSUSERPERM.dbaauth,
   SYSUSERPERM.scheduleauth,
   SYSUSERPERM.user_group,
   SYSUSERPERM.publishauth,
   SYSUSERPERM.remotedbaauth,
   SYSUSERPERM.remarks
    from SYS.SYSUSERPERM;
```

See also

- “SYSUSERPERM compatibility view (deprecated)” on page 1068
- “SYSUSERAUTHORITY system view” on page 1040

Views for Transact-SQL compatibility

The Adaptive Server Enterprise and SQL Anywhere system catalogs are different. The Adaptive Server Enterprise system tables and views are owned by the user dbo, and exist partly in the master database, partly in the sybsecurity database, and partly in each individual database. The SQL Anywhere system tables and views are owned by the special user SYS and exist separately in each database.

To assist in preparing compatible applications, SQL Anywhere provides the following set of views owned by the special user dbo, which correspond to their Adaptive Server Enterprise counterparts. Where architectural differences make the contents of a particular Adaptive Server Enterprise table or view meaningless in a SQL Anywhere context, the view is empty, containing just the column names and data types.

View name	Description
syscolumns	One row for each column in a table or view, and for each parameter in a procedure
syscomments	One or more rows for each view, rule, default, trigger, and procedure, giving the SQL definition statement
sysindexes	One row for each clustered or nonclustered index, one row for each table with no indexes, and an additional row for each table containing text or image data.

View name	Description
sysobjects	One row for each table, view, procedure, rule, trigger default, log, or (in tempdb only) temporary object
systypes	One row for each system-supplied or user-defined data type
sysusers	One row for each user allowed in the database
syslogins	One row for each valid user account

Glossary

Glossary

Adaptive Server Anywhere (ASA)

The relational database server component of SQL Anywhere Studio, intended for use in mobile and embedded environments or as a server for small and medium-sized businesses. In version 10.0.0, Adaptive Server Anywhere was renamed SQL Anywhere Server, and SQL Anywhere Studio was renamed SQL Anywhere.

See also: "SQL Anywhere" on page 1097.

agent ID

See also: "client message store ID" on page 1075.

article

In MobiLink or SQL Remote, an article is a database object that represents a whole table, or a subset of the columns and rows in a table. Articles are grouped together in a publication.

See also:

- "replication" on page 1095
- "publication" on page 1092

atomic transaction

A transaction that is guaranteed to complete successfully or not at all. If an error prevents part of an atomic transaction from completing, the transaction is rolled back to prevent the database from being left in an inconsistent state.

base table

Permanent tables for data. Tables are sometimes called **base tables** to distinguish them from temporary tables and views.

See also:

- "temporary table" on page 1099
- "view" on page 1101

bit array

A bit array is a type of array data structure that is used for efficient storage of a sequence of bits. A bit array is similar to a character string, except that the individual pieces are 0s (zeros) and 1s (ones) instead of characters. Bit arrays are typically used to hold a string of Boolean values.

business rule

A guideline based on real-world requirements. Business rules are typically implemented through check constraints, user-defined data types, and the appropriate use of transactions.

See also:

- "constraint" on page 1077
- "user-defined data type" on page 1101

carrier

A MobiLink object, stored in MobiLink system tables or a Notifier properties file, that contains information about a public carrier for use by server-initiated synchronization.

See also: "server-initiated synchronization" on page 1096.

character set

A character set is a set of symbols, including letters, digits, spaces, and other symbols. An example of a character set is ISO-8859-1, also known as Latin1.

See also:

- "code page" on page 1075
- "encoding" on page 1081
- "collation" on page 1075

check constraint

A restriction that enforces specified conditions on a column or set of columns.

See also:

- "constraint" on page 1077
- "foreign key constraint" on page 1082
- "primary key constraint" on page 1092
- "unique constraint" on page 1100

checkpoint

The point at which all changes to the database are saved to the database file. At other times, committed changes are saved only to the transaction log.

checksum

The calculated number of bits of a database page that is recorded with the database page itself. The checksum allows the database management system to validate the integrity of the page by ensuring that the numbers match as the page is being written to disk. If the counts match, it's assumed that page was successfully written.

client message store

In QAnywhere, a SQL Anywhere database on the remote device that stores messages.

client message store ID

In QAnywhere, a MobiLink remote ID that uniquely identifies a client message store.

client/server

A software architecture where one application (the client) obtains information from and sends information to another application (the server). The two applications often reside on different computers connected by a network.

code page

A code page is an encoding that maps characters of a character set to numeric representations, typically an integer between 0 and 255. An example of a code page is Windows code page 1252. For the purposes of this documentation, code page and encoding are interchangeable terms.

See also:

- “character set” on page 1074
- “encoding” on page 1081
- “collation” on page 1075

collation

A combination of a character set and a sort order that defines the properties of text in the database. For SQL Anywhere databases, the default collation is determined by the operating system and language on which the server is running; for example, the default collation on English Windows systems is 1252LATIN1. A collation, also called a collating sequence, is used for comparing and sorting strings.

See also:

- “character set” on page 1074
- “code page” on page 1075
- “encoding” on page 1081

command file

A text file containing SQL statements. Command files can be built manually, or they can be built automatically by database utilities. The dbunload utility, for example, creates a command file consisting of the SQL statements necessary to recreate a given database.

communication stream

In MobiLink, the network protocol used for communication between the MobiLink client and the MobiLink server.

concurrency

The simultaneous execution of two or more independent, and possibly competing, processes. SQL Anywhere automatically uses locking to isolate transactions and ensure that each concurrent application sees a consistent set of data.

See also:

- “transaction” on page 1099
- “isolation level” on page 1085

conflict resolution

In MobiLink, conflict resolution is logic that specifies what to do when two users modify the same row on different remote databases.

connection ID

A unique number that identifies a given connection between a client application and the database. You can determine the current connection ID using the following SQL statement:

```
SELECT CONNECTION_PROPERTY( 'Number' );
```

connection-initiated synchronization

A form of MobiLink server-initiated synchronization in which synchronization is initiated when there are changes to connectivity.

See also: “server-initiated synchronization” on page 1096.

connection profile

A set of parameters that are required to connect to a database, such as user name, password, and server name, that is stored and used as a convenience.

consolidated database

In distributed database environments, a database that stores the master copy of the data. In case of conflict or discrepancy, the consolidated database is considered to have the primary copy of the data.

See also:

- “synchronization” on page 1099
- “replication” on page 1095

constraint

A restriction on the values contained in a particular database object, such as a table or column. For example, a column may have a uniqueness constraint, which requires that all values in the column be different. A table may have a foreign key constraint, which specifies how the information in the table relates to data in some other table.

See also:

- “check constraint” on page 1074
- “foreign key constraint” on page 1082
- “primary key constraint” on page 1092
- “unique constraint” on page 1100

contention

The act of competing for resources. For example, in database terms, two or more users trying to edit the same row of a database contend for the rights to edit that row.

correlation name

The name of a table or view that is used in the FROM clause of a query—either its original name, or an alternate name, that is defined in the FROM clause.

creator ID

In UltraLite Palm OS applications, an ID that is assigned when the application is created.

cursor

A named linkage to a result set, used to access and update rows from a programming interface. In SQL Anywhere, cursors support forward and backward movement through the query results. Cursors consist of two parts: the cursor result set, typically defined by a SELECT statement; and the cursor position.

See also:

- “cursor result set” on page 1077
- “cursor position” on page 1077

cursor position

A pointer to one row within the cursor result set.

See also:

- “cursor” on page 1077
- “cursor result set” on page 1077

cursor result set

The set of rows resulting from a query that is associated with a cursor.

See also:

- “cursor” on page 1077
- “cursor position” on page 1077

data cube

A multi-dimensional result set with each dimension reflecting a different way to group and sort the same results. Data cubes provide complex information about data that would otherwise require self-join queries and correlated subqueries. Data cubes are a part of OLAP functionality.

data definition language (DDL)

The subset of SQL statements for defining the structure of data in the database. DDL statements create, modify, and remove database objects, such as tables and users.

data manipulation language (DML)

The subset of SQL statements for manipulating data in the database. DML statements retrieve, insert, update, and delete data in the database.

data type

The format of data, such as CHAR or NUMERIC. In the ANSI SQL standard, data types can also include a restriction on size, character set, and collation.

See also: “domain” on page 1080.

database

A collection of tables that are related by primary and foreign keys. The tables hold the information in the database. The tables and keys together define the structure of the database. A database management system accesses this information.

See also:

- “foreign key” on page 1082
- “primary key” on page 1092
- “database management system (DBMS)” on page 1079
- “relational database management system (RDBMS)” on page 1094

database administrator (DBA)

The user with the permissions required to maintain the database. The DBA is generally responsible for all changes to a database schema, and for managing users and groups. The role of database administrator is automatically built into databases as user ID DBA with password sql.

database connection

A communication channel between a client application and the database. A valid user ID and password are required to establish a connection. The privileges granted to the user ID determine the actions that can be carried out during the connection.

database file

A database is held in one or more database files. There is an initial file, and subsequent files are called dbspaces. Each table, including its indexes, must be contained within a single database file.

See also: "dbspace" on page 1080.

database management system (DBMS)

A collection of programs that allow you to create and use databases.

See also: "relational database management system (RDBMS)" on page 1094.

database name

The name given to a database when it is loaded by a server. The default database name is the root of the initial database file.

See also: "database file" on page 1079.

database object

A component of a database that contains or receives information. Tables, indexes, views, procedures, and triggers are database objects.

database owner (dbo)

A special user that owns the system objects not owned by SYS.

See also:

- "database administrator (DBA)" on page 1078
- "SYS" on page 1099

database server

A computer program that regulates all access to information in a database. SQL Anywhere provides two types of servers: network servers and personal servers.

DBA authority

The level of permission that enables a user to do administrative activity in the database. The DBA user has DBA authority by default.

See also: "database administrator (DBA)" on page 1078.

dbspace

An additional database file that creates more space for data. A database can be held in up to 13 separate files (an initial file and 12 dbspaces). Each table, together with its indexes, must be contained in a single database file. The SQL command CREATE DBSPACE adds a new file to the database.

See also: "database file" on page 1079.

deadlock

A state where a set of transactions arrives at a place where none can proceed.

device tracking

In MobiLink server-initiated synchronization, functionality that allows you to address messages using the MobiLink user name that identifies a device.

See also: "server-initiated synchronization" on page 1096.

direct row handling

In MobiLink, a way to synchronize table data to sources other than the MobiLink-supported consolidated databases. You can implement both uploads and downloads with direct row handling.

See also:

- "consolidated database" on page 1076
- "SQL-based synchronization" on page 1097

domain

Aliases for built-in data types, including precision and scale values where applicable, and optionally including DEFAULT values and CHECK conditions. Some domains, such as the monetary data types, are pre-defined in SQL Anywhere. Also called user-defined data type.

See also: "data type" on page 1078.

download

The stage in synchronization where data is transferred from the consolidated database to a remote database.

dynamic SQL

SQL that is generated programmatically by your program before it is executed. UltraLite dynamic SQL is a variant designed for small-footprint devices.

EBF

Express Bug Fix. An express bug fix is a subset of the software with one or more bug fixes. The bug fixes are listed in the release notes for the update. Bug fix updates may only be applied to installed software with the same version number. Some testing has been performed on the software, but the software has not undergone full testing. You should not distribute these files with your application unless you have verified the suitability of the software yourself.

embedded SQL

A programming interface for C programs. SQL Anywhere embedded SQL is an implementation of the ANSI and IBM standard.

encoding

Also known as character encoding, an encoding is a method by which each character in a character set is mapped onto one or more bytes of information, typically represented as a hexadecimal number. An example of an encoding is UTF-8.

See also:

- "character set" on page 1074
- "code page" on page 1075
- "collation" on page 1075

event model

In MobiLink, the sequence of events that make up a synchronization, such as begin_synchronization and download_cursor. Events are invoked if a script is created for them.

external login

An alternate login name and password used when communicating with a remote server. By default, SQL Anywhere uses the names and passwords of its clients whenever it connects to a remote server on behalf of those clients. However, this default can be overridden by creating external logins. External logins are alternate login names and passwords used when communicating with a remote server.

extraction

In SQL Remote replication, the act of unloading the appropriate structure and data from the consolidated database. This information is used to initialize the remote database.

See also: "replication" on page 1095.

failover

Switching to a redundant or standby server, system, or network on failure or unplanned termination of the active server, system, or network. Failover happens automatically.

FILE

In SQL Remote replication, a message system that uses shared files for exchanging replication messages. This is useful for testing and for installations without an explicit message-transport system.

See also:"replication" on page 1095.

file-based download

In MobiLink, a way to synchronize data in which downloads are distributed as files, allowing offline distribution of synchronization changes.

file-definition database

In MobiLink, a SQL Anywhere database that is used for creating download files.

See also: “file-based download” on page 1082.

foreign key

One or more columns in a table that duplicate the primary key values in another table. Foreign keys establish relationships between tables.

See also:

- “primary key” on page 1092
- “foreign table” on page 1082

foreign key constraint

A restriction on a column or set of columns that specifies how the data in the table relates to the data in some other table. Imposing a foreign key constraint on a set of columns makes those columns the foreign key.

See also:

- “constraint” on page 1077
- “check constraint” on page 1074
- “primary key constraint” on page 1092
- “unique constraint” on page 1100

foreign table

The table containing the foreign key.

See also: “foreign key” on page 1082.

full backup

A backup of the entire database, and optionally, the transaction log. A full backup contains all the information in the database and provides protection in the event of a system or media failure.

See also: “incremental backup” on page 1084.

gateway

A MobiLink object, stored in MobiLink system tables or a Notifier properties file, that contains information about how to send messages for server-initiated synchronization.

See also: “server-initiated synchronization” on page 1096.

generated join condition

A restriction on join results that is automatically generated. There are two types: key and natural. Key joins are generated when you specify KEY JOIN or when you specify the keyword JOIN but do not use the keywords CROSS, NATURAL, or ON. For a key join, the generated join condition is based on foreign key relationships between tables. Natural joins are generated when you specify NATURAL JOIN; the generated join condition is based on common column names in the two tables.

See also:

- "join" on page 1086
- "join condition" on page 1086

generation number

In MobiLink, a mechanism for forcing remote databases to upload data before applying any more download files.

See also: "file-based download" on page 1082.

global temporary table

A type of temporary table for which data definitions are visible to all users until explicitly dropped. Global temporary tables let each user open their own identical instance of a table. By default, rows are deleted on commit, and rows are always deleted when the connection is ended.

See also:

- "temporary table" on page 1099
- "local temporary table" on page 1086

grant option

The level of permission that allows a user to grant permissions to other users.

hash

A hash is an index optimization that transforms index entries into keys. An index hash aims to avoid the expensive operation of finding, loading, and then unpacking the rows to determine the indexed value, by including enough of the actual row data with its row ID.

histogram

The most important component of column statistics, histograms are a representation of data distribution. SQL Anywhere maintains histograms to provide the optimizer with statistical information about the distribution of values in columns.

iAnywhere JDBC driver

The iAnywhere JDBC driver provides a JDBC driver that has some performance benefits and feature benefits compared to the pure Java jConnect JDBC driver, but which is not a pure-Java solution. The iAnywhere JDBC driver is recommended in most cases.

See also:

- "JDBC" on page 1085
- "jConnect" on page 1085

identifier

A string of characters used to reference a database object, such as a table or column. An identifier may contain any character from A through Z, a through z, 0 through 9, underscore (_), at sign (@), number sign (#), or dollar sign ($).

incremental backup

A backup of the transaction log only, typically used between full backups.

See also: "transaction log" on page 1099.

index

A sorted set of keys and pointers associated with one or more columns in a base table. An index on one or more columns of a table can improve performance.

InfoMaker

A reporting and data maintenance tool that lets you create sophisticated forms, reports, graphs, cross-tabs, and tables, and applications that use these reports as building blocks.

inner join

A join in which rows appear in the result set only if both tables satisfy the join condition. Inner joins are the default.

See also:

- "join" on page 1086
- "outer join" on page 1090

integrated login

A login feature that allows the same single user ID and password to be used for operating system logins, network logins, and database connections.

integrity

Adherence to rules that ensure that data is correct and accurate, and that the relational structure of the database is intact.

See also: "referential integrity" on page 1094.

Interactive SQL

A SQL Anywhere application that allows you to query and alter data in your database, and modify the structure of your database. Interactive SQL provides a pane for you to enter SQL statements, and panes that return information about how the query was processed and the result set.

isolation level

The degree to which operations in one transaction are visible to operations in other concurrent transactions. There are four isolation levels, numbered 0 through 3. Level 3 provides the highest level of isolation. Level 0 is the default setting. SQL Anywhere also supports three snapshot isolation levels: snapshot, statement-snapshot, and readonly-statement-snapshot.

See also: "snapshot isolation" on page 1097.

JAR file

Java archive file. A compressed file format consisting of a collection of one or more packages used for Java applications. It includes all the resources necessary to install and run a Java program in a single compressed file.

Java class

The main structural unit of code in Java. It is a collection of procedures and variables grouped together because they all relate to a specific, identifiable category.

jConnect

A Java implementation of the JavaSoft JDBC standard. It provides Java developers with native database access in multi-tier and heterogeneous environments. However, the iAnywhere JDBC driver is the preferred JDBC driver for most cases.

See also:

- "JDBC" on page 1085
- "iAnywhere JDBC driver" on page 1084

JDBC

Java Database Connectivity. A SQL-language programming interface that allows Java applications to access relational data. The preferred JDBC driver is the iAnywhere JDBC driver.

See also:

- “jConnect” on page 1085
- “iAnywhere JDBC driver” on page 1084

join

A basic operation in a relational system that links the rows in two or more tables by comparing the values in specified columns.

join condition

A restriction that affects join results. You specify a join condition by inserting an ON clause or WHERE clause immediately after the join. In the case of natural and key joins, SQL Anywhere generates a join condition.

See also:

- “join” on page 1086
- “generated join condition” on page 1083

join type

SQL Anywhere provides four types of joins: cross join, key join, natural join, and joins using an ON clause.

See also: “join” on page 1086.

light weight poller

In MobiLink server-initiated synchronization, a device application that polls for push notifications from a MobiLink server.

See also: “server-initiated synchronization” on page 1096.

Listener

A program, dblsn, that is used for MobiLink server-initiated synchronization. Listeners are installed on remote devices and configured to initiate actions on the device when they receive push notifications.

See also: “server-initiated synchronization” on page 1096.

local temporary table

A type of temporary table that exists only for the duration of a compound statement or until the end of the connection. Local temporary tables are useful when you need to load a set of data only once. By default, rows are deleted on commit.

See also:

- “temporary table” on page 1099
- “global temporary table” on page 1083

lock

A concurrency control mechanism that protects the integrity of data during the simultaneous execution of multiple transactions. SQL Anywhere automatically applies locks to prevent two connections from changing the same data at the same time, and to prevent other connections from reading data that is in the process of being changed.

You control locking by setting the isolation level.

See also:

- "isolation level" on page 1085
- "concurrency" on page 1076
- "integrity" on page 1085

log file

A log of transactions maintained by SQL Anywhere. The log file is used to ensure that the database is recoverable in the event of a system or media failure, to improve database performance, and to allow data replication using SQL Remote.

See also:

- "transaction log" on page 1099
- "transaction log mirror" on page 1100
- "full backup" on page 1082

logical index

A reference (pointer) to a physical index. There is no indexing structure stored on disk for a logical index.

LTM

Log Transfer Manager (LTM) also called Replication Agent. Used with Replication Server, the LTM is the program that reads a database transaction log and sends committed changes to Sybase Replication Server.

See: "Replication Server" on page 1095.

maintenance release

A maintenance release is a complete set of software that upgrades installed software from an older version with the same major version number (version number format is *major.minor.patch.build*). Bug fixes and other changes are listed in the release notes for the upgrade.

materialized view

A materialized view is a view that has been computed and stored on disk. Materialized views have characteristics of both views (they are defined using a query specification), and of tables (they allow most table operations to be performed on them).

See also:

- “base table” on page 1073
- “view” on page 1101

message log

A log where messages from an application such as a database server or MobiLink server can be stored. This information can also appear in a messages window or be logged to a file. The message log includes informational messages, errors, warnings, and messages from the MESSAGE statement.

message store

In QAnywhere, databases on the client and server device that store messages.

See also:

- “client message store” on page 1075
- “server message store” on page 1097

message system

In SQL Remote replication, a protocol for exchanging messages between the consolidated database and a remote database. SQL Anywhere includes support for the following message systems: FILE, FTP, and SMTP.

See also:

- “replication” on page 1095
- “FILE” on page 1081

message type

In SQL Remote replication, a database object that specifies how remote users communicate with the publisher of a consolidated database. A consolidated database may have several message types defined for it; this allows different remote users to communicate with it using different message systems.

See also:

- “replication” on page 1095
- “consolidated database” on page 1076

metadata

Data about data. Metadata describes the nature and content of other data.

See also: “schema” on page 1096.

mirror log

See also: “transaction log mirror” on page 1100.

MobiLink

A session-based synchronization technology designed to synchronize UltraLite and SQL Anywhere remote databases with a consolidated database.

See also:

- "consolidated database" on page 1076
- "synchronization" on page 1099
- "UltraLite" on page 1100

MobiLink client

There are two kinds of MobiLink clients. For SQL Anywhere remote databases, the MobiLink client is the dbmlsync command line utility. For UltraLite remote databases, the MobiLink client is built in to the UltraLite runtime library.

MobiLink Monitor

A graphical tool for monitoring MobiLink synchronizations.

MobiLink server

The computer program that runs MobiLink synchronization, mlsrv11.

MobiLink system table

System tables that are required by MobiLink synchronization. They are installed by MobiLink setup scripts into the MobiLink consolidated database.

MobiLink user

A MobiLink user is used to connect to the MobiLink server. You create the MobiLink user on the remote database and register it in the consolidated database. MobiLink user names are entirely independent of database user names.

network protocol

The type of communication, such as TCP/IP or HTTP.

network server

A database server that accepts connections from computers sharing a common network.

See also: "personal server" on page 1091.

normalization

The refinement of a database schema to eliminate redundancy and improve organization according to rules based on relational database theory.

Notifier

A program that is used by MobiLink server-initiated synchronization. Notifiers are integrated into the MobiLink server. They check the consolidated database for push requests, and send push notifications.

See also:

- "server-initiated synchronization" on page 1096
- "Listener" on page 1086

object tree

In Sybase Central, the hierarchy of database objects. The top level of the object tree shows all products that your version of Sybase Central supports. Each product expands to reveal its own sub-tree of objects.

See also: "Sybase Central" on page 1098.

ODBC

Open Database Connectivity. A standard Windows interface to database management systems. ODBC is one of several interfaces supported by SQL Anywhere.

ODBC Administrator

A Microsoft program included with Windows operating systems for setting up ODBC data sources.

ODBC data source

A specification of the data a user wants to access via ODBC, and the information needed to get to that data.

outer join

A join that preserves all the rows in a table. SQL Anywhere supports left, right, and full outer joins. A left outer join preserves the rows in the table to the left of the join operator, and returns a null when a row in the right table does not satisfy the join condition. A full outer join preserves all the rows from both tables.

See also:

- "join" on page 1086
- "inner join" on page 1084

package

In Java, a collection of related classes.

parse tree

An algebraic representation of a query.

PDB

A Palm database file.

performance statistic

A value reflecting the performance of the database system. The CURRREAD statistic, for example, represents the number of file reads issued by the database server that have not yet completed.

personal server

A database server that runs on the same computer as the client application. A personal database server is typically used by a single user on a single computer, but it can support several concurrent connections from that user.

physical index

The actual indexing structure of an index, as it is stored on disk.

plug-in module

In Sybase Central, a way to access and administer a product. Plug-ins are usually installed and registered automatically with Sybase Central when you install the respective product. Typically, a plug-in appears as a top-level container, in the Sybase Central main window, using the name of the product itself; for example, SQL Anywhere.

See also: “Sybase Central” on page 1098.

policy

In QAnywhere, the way you specify when message transmission should occur.

polling

In MobiLink server-initiated synchronization, the way a light weight poller, such as the MobiLink Listener, requests push notifications from a Notifier.

See also: “server-initiated synchronization” on page 1096.

PowerDesigner

A database modeling application. PowerDesigner provides a structured approach to designing a database or data warehouse. SQL Anywhere includes the Physical Data Model component of PowerDesigner.

PowerJ

A Sybase product for developing Java applications.

predicate

A conditional expression that is optionally combined with the logical operators AND and OR to make up the set of conditions in a WHERE or HAVING clause. In SQL, a predicate that evaluates to UNKNOWN is interpreted as FALSE.

primary key

A column or list of columns whose values uniquely identify every row in the table.

See also: “foreign key” on page 1082.

primary key constraint

A uniqueness constraint on the primary key columns. A table can have only one primary key constraint.

See also:

- “constraint” on page 1077
- “check constraint” on page 1074
- “foreign key constraint” on page 1082
- “unique constraint” on page 1100
- “integrity” on page 1085

primary table

The table containing the primary key in a foreign key relationship.

proxy table

A local table containing metadata used to access a table on a remote database server as if it were a local table.

See also: “metadata” on page 1088.

publication

In MobiLink or SQL Remote, a database object that identifies data that is to be synchronized. In MobiLink, publications exist only on the clients. A publication consists of articles. SQL Remote users can receive a publication by subscribing to it. MobiLink users can synchronize a publication by creating a synchronization subscription to it.

See also:

- “replication” on page 1095
- “article” on page 1073
- “publication update” on page 1092

publication update

In SQL Remote replication, a list of changes made to one or more publications in one database. A publication update is sent periodically as part of a replication message to the remote database(s).

See also:

- “replication” on page 1095
- “publication” on page 1092

publisher

In SQL Remote replication, the single user in a database who can exchange replication messages with other replicating databases.

See also: "replication" on page 1095.

push notification

In QAnywhere, a special message delivered from the server to a QAnywhere client that prompts the client to initiate a message transmission. In MobiLink server-initiated synchronization, a special message delivered from a Notifer to a device that contains push request data and internal information.

See also:

- "QAnywhere" on page 1093
- "server-initiated synchronization" on page 1096

push request

In MobiLink server-initiated synchronization, a row of values in a result set that a Notifier checks to determine if push notifications need to be sent to a device.

See also: "server-initiated synchronization" on page 1096.

QAnywhere

Application-to-application messaging, including mobile device to mobile device and mobile device to and from the enterprise, that permits communication between custom programs running on mobile or wireless devices and a centrally located server application.

QAnywhere agent

In QAnywhere, a process running on the client device that monitors the client message store and determines when message transmission should occur.

query

A SQL statement or group of SQL statements that access and/or manipulate data in a database.

See also: "SQL" on page 1097.

Redirector

A web server plug-in that routes requests and responses between a client and the MobiLink server. This plug-in also implements load-balancing and failover mechanisms.

reference database

In MobiLink, a SQL Anywhere database used in the development of UltraLite clients. You can use a single SQL Anywhere database as both reference and consolidated database during development. Databases made with other products cannot be used as reference databases.

referencing object

An object, such as a view, whose definition directly references another object in the database, such as a table.

See also: "foreign key" on page 1082.

referenced object

An object, such as a table, that is directly referenced in the definition of another object, such as a view.

See also: "primary key" on page 1092.

referential integrity

Adherence to rules governing data consistency, specifically the relationships between the primary and foreign key values in different tables. To have referential integrity, the values in each foreign key must correspond to the primary key values of a row in the referenced table.

See also:

- "primary key" on page 1092
- "foreign key" on page 1082

regular expression

A regular expression is a sequence of characters, wildcards, and operators that defines a pattern to search for within a string.

relational database management system (RDBMS)

A type of database management system that stores data in the form of related tables.

See also: "database management system (DBMS)" on page 1079.

remote database

In MobiLink or SQL Remote, a database that exchanges data with a consolidated database. Remote databases may share all or some of the data in the consolidated database.

See also:

- "synchronization" on page 1099
- "consolidated database" on page 1076

REMOTE DBA authority

In SQL Remote, a level of permission required by the Message Agent (dbremote). In MobiLink, a level of permission required by the SQL Anywhere synchronization client (dbmlsync). When the Message Agent (dbremote) or synchronization client connects as a user who has this authority, it has full DBA access. The user ID has no additional permissions when not connected through the Message Agent (dbremote) or synchronization client (dbmlsync).

See also: "DBA authority" on page 1079.

remote ID

A unique identifier in SQL Anywhere and UltraLite databases that is used by MobiLink. The remote ID is initially set to NULL and is set to a GUID during a database's first synchronization.

replication

The sharing of data among physically distinct databases. Sybase has three replication technologies: MobiLink, SQL Remote, and Replication Server.

Replication Agent

See: “LTM” on page 1087.

replication frequency

In SQL Remote replication, a setting for each remote user that determines how often the publisher's message agent should send replication messages to that remote user.

See also: “replication” on page 1095.

replication message

In SQL Remote or Replication Server, a communication sent between a publishing database and a subscribing database. Messages contain data, passthrough statements, and information required by the replication system.

See also:

- “replication” on page 1095
- “publication update” on page 1092

Replication Server

A Sybase connection-based replication technology that works with SQL Anywhere and Adaptive Server Enterprise. It is intended for near-real time replication between a few databases.

See also: “LTM” on page 1087.

role

In conceptual database modeling, a verb or phrase that describes a relationship from one point of view. You can describe each relationship with two roles. Examples of roles are "contains" and "is a member of."

role name

The name of a foreign key. This is called a role name because it names the relationship between the foreign table and primary table. By default, the role name is the table name, unless another foreign key is already using that name, in which case the default role name is the table name followed by a three-digit unique number. You can also create the role name yourself.

See also: “foreign key” on page 1082.

rollback log

A record of the changes made during each uncommitted transaction. In the event of a ROLLBACK request or a system failure, uncommitted transactions are reversed out of the database, returning the database to its former state. Each transaction has a separate rollback log, which is deleted when the transaction is complete.

See also: "transaction" on page 1099.

row-level trigger

A trigger that executes once for each row that is changed.

See also:

- "trigger" on page 1100
- "statement-level trigger" on page 1098

schema

The structure of a database, including tables, columns, and indexes, and the relationships between them.

script

In MobiLink, code written to handle MobiLink events. Scripts programmatically control data exchange to meet business needs.

See also: "event model" on page 1081.

script-based upload

In MobiLink, a way to customize the upload process as an alternative to using the log file.

script version

In MobiLink, a set of synchronization scripts that are applied together to create a synchronization.

secured feature

A feature specified by the -sf option when a database server is started, so it is not available for any database running on that database server.

server-initiated synchronization

A way to initiate MobiLink synchronization from the MobiLink server.

server management request

A QAnywhere message that is formatted as XML and sent to the QAnywhere system queue as a way to administer the server message store or monitor QAnywhere applications.

server message store

In QAnywhere, a relational database on the server that temporarily stores messages until they are transmitted to a client message store or JMS system. Messages are exchanged between clients via the server message store.

service

In Windows operating systems, a way of running applications when the user ID running the application is not logged on.

session-based synchronization

A type of synchronization where synchronization results in consistent data representation across both the consolidated and remote databases. MobiLink is session-based.

snapshot isolation

A type of isolation level that returns a committed version of the data for transactions that issue read requests. SQL Anywhere provides three snapshot isolation levels: snapshot, statement-snapshot, and readonly-statement-snapshot. When using snapshot isolation, read operations do not block write operations.

See also: "isolation level" on page 1085.

SQL

The language used to communicate with relational databases. ANSI has defined standards for SQL, the latest of which is SQL-2003. SQL stands, unofficially, for Structured Query Language.

SQL Anywhere

The relational database server component of SQL Anywhere that is intended for use in mobile and embedded environments or as a server for small and medium-sized businesses. SQL Anywhere is also the name of the package that contains the SQL Anywhere RDBMS, the UltraLite RDBMS, MobiLink synchronization software, and other components.

SQL-based synchronization

In MobiLink, a way to synchronize table data to MobiLink-supported consolidated databases using MobiLink events. For SQL-based synchronization, you can use SQL directly or you can return SQL using the MobiLink server APIs for Java and .NET.

SQL Remote

A message-based data replication technology for two-way replication between consolidated and remote databases. The consolidated and remote databases must be SQL Anywhere.

SQL statement

A string containing SQL keywords designed for passing instructions to a DBMS.

See also:

- “schema” on page 1096
- “SQL” on page 1097
- “database management system (DBMS)” on page 1079

statement-level trigger

A trigger that executes after the entire triggering statement is completed.

See also:

- “trigger” on page 1100
- “row-level trigger” on page 1096

stored procedure

A stored procedure is a group of SQL instructions stored in the database and used to execute a set of operations or queries on a database server

string literal

A string literal is a sequence of characters enclosed in single quotes.

subquery

A SELECT statement that is nested inside another SELECT, INSERT, UPDATE, or DELETE statement, or another subquery.

There are two types of subquery: correlated and nested.

subscription

In MobiLink synchronization, a link in a client database between a publication and a MobiLink user, allowing the data described by the publication to be synchronized.

In SQL Remote replication, a link between a publication and a remote user, allowing the user to exchange updates on that publication with the consolidated database.

See also:

- “publication” on page 1092
- “MobiLink user” on page 1089

Sybase Central

A database management tool that provides SQL Anywhere database settings, properties, and utilities in a graphical user interface. Sybase Central can also be used for managing other Sybase products, including MobiLink.

synchronization

The process of replicating data between databases using MobiLink technology.

In SQL Remote, synchronization is used exclusively to denote the process of initializing a remote database with an initial set of data.

See also:

- "MobiLink" on page 1089
- "SQL Remote" on page 1097

SYS

A special user that owns most of the system objects. You cannot log in as SYS.

system object

Database objects owned by SYS or dbo.

system table

A table, owned by SYS or dbo, that holds metadata. System tables, also known as data dictionary tables, are created and maintained by the database server.

system view

A type of view, included in every database, that presents the information held in the system tables in an easily understood format.

temporary table

A table that is created for the temporary storage of data. There are two types: global and local.

See also:

- "local temporary table" on page 1086
- "global temporary table" on page 1083

transaction

A sequence of SQL statements that comprise a logical unit of work. A transaction is processed in its entirety or not at all. SQL Anywhere supports transaction processing, with locking features built in to allow concurrent transactions to access the database without corrupting the data. Transactions end either with a COMMIT statement, which makes the changes to the data permanent, or a ROLLBACK statement, which undoes all the changes made during the transaction.

transaction log

A file storing all changes made to a database, in the order in which they are made. It improves performance and allows data recovery in the event the database file is damaged.

transaction log mirror

An optional identical copy of the transaction log file, maintained simultaneously. Every time a database change is written to the transaction log file, it is also written to the transaction log mirror file.

A mirror file should be kept on a separate device from the transaction log, so that if either device fails, the other copy of the log keeps the data safe for recovery.

See also: "transaction log" on page 1099.

transactional integrity

In MobiLink, the guaranteed maintenance of transactions across the synchronization system. Either a complete transaction is synchronized, or no part of the transaction is synchronized.

transmission rule

In QAnywhere, logic that determines when message transmission is to occur, which messages to transmit, and when messages should be deleted.

trigger

A special form of stored procedure that is executed automatically when a user runs a query that modifies the data.

See also:

- "row-level trigger" on page 1096
- "statement-level trigger" on page 1098
- "integrity" on page 1085

UltraLite

A database optimized for small, mobile, and embedded devices. Intended platforms include cell phones, pagers, and personal organizers.

UltraLite runtime

An in-process relational database management system that includes a built-in MobiLink synchronization client. The UltraLite runtime is included in the libraries used by each of the UltraLite programming interfaces, and in the UltraLite engine.

unique constraint

A restriction on a column or set of columns requiring that all non-null values are different. A table can have multiple unique constraints.

See also:

- "foreign key constraint" on page 1082
- "primary key constraint" on page 1092
- "constraint" on page 1077

unload

Unloading a database exports the structure and/or data of the database to text files (SQL command files for the structure, and ASCII comma-separated files for the data). You unload a database with the Unload utility.

In addition, you can unload selected portions of your data using the UNLOAD statement.

upload

The stage in synchronization where data is transferred from a remote database to a consolidated database.

user-defined data type

See "domain" on page 1080.

validate

To test for particular types of file corruption of a database, table, or index.

view

A SELECT statement that is stored in the database as an object. It allows users to see a subset of rows or columns from one or more tables. Each time a user uses a view of a particular table, or combination of tables, it is recomputed from the information stored in those tables. Views are useful for security purposes, and to tailor the appearance of database information to make data access straightforward.

window

The group of rows over which an analytic function is performed. A window may contain one, many, or all rows of data that has been partitioned according to the grouping specifications provided in the window definition. The window moves to include the number or range of rows needed to perform the calculations for the current row in the input. The main benefit of the window construct is that it allows additional opportunities for grouping and analysis of results, without having to perform additional queries.

Windows

The Microsoft Windows family of operating systems, such as Windows Vista, Windows XP, and Windows 200x.

Windows CE

See "Windows Mobile" on page 1101.

Windows Mobile

A family of operating systems produced by Microsoft for mobile devices.

work table

An internal storage area for interim results during query optimization.

Index