SYBASE iAnywhere

SQL Anywhere® Server
SQL Usage

Contents

About this book

This book describes how to add objects to a database; how to import, export, and modify data; how to retrieve data; and how to build stored procedures and triggers.

About the SQL Anywhere documentation

The complete SQL Anywhere documentation is available in four formats:

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

  To access the documentation, go to http://dcx.sybase.com.

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

  To access the documentation, choose **Start** » **Programs** » **SQL Anywhere 12** » **Documentation** » **HTML Help (English)**.

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

- **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 access the PDF documentation on Windows operating systems, choose **Start** » **Programs** » **SQL Anywhere 12** » **Documentation** » **PDF (English)**.

  To access the PDF documentation on Unix operating systems, use a web browser to open */documentation/en/pdf/index.html* under the SQL Anywhere installation directory.

Documentation conventions

This section lists the conventions used in this documentation.

Operating systems

SQL Anywhere runs on a variety of platforms. Typically, the behavior of the software is 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 (IBM 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 platforms that are used primarily on server, desktop, and laptop computers, as well as platforms used on mobile devices. Unless otherwise specified, when the documentation refers to Windows, it refers to all supported Windows-based platforms, including Windows Mobile.

  Windows Mobile is based on the Windows CE operating system, which is also used to build a variety of platforms other than Windows Mobile. Unless otherwise specified, when the documentation refers to Windows Mobile, it refers to all supported platforms built using Windows CE.

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

For the complete list of platforms supported by SQL Anywhere, see "Supported platforms" [*SQL Anywhere 12 - Introduction*].

Directory and file names

Usually 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. You can usually 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 *dbsrv12.exe*. On Unix, it is *dbsrv12*.

- ***install-dir*** During the installation process, you choose where to install SQL Anywhere. The environment variable SQLANY12 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 *%SQLANY12%\readme.txt*. On Unix, this is equivalent to *$SQLANY12/readme.txt* or *${SQLANY12}/readme.txt*.

  For more information about the default location of *install-dir*, see "SQLANY12 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 SQLANYSAMP12 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*, choose **Start** » **Programs** » **SQL Anywhere 12** » **Sample Applications And Projects**.

  For more information about the default location of *samples-dir*, see "SQLANYSAMP12 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)"
  ```

- **Semicolons** On Unix, semicolons should be enclosed in quotes.
- **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}*.

Contacting the documentation team

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

You can 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

Go to http://dcx.sybase.com.

Finding out more and requesting technical support

Newsgroups

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: **dbeng12 -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.

Developer Centers

The **SQL Anywhere Tech Corner** gives developers easy access to product technical documentation. You can browse technical white papers, FAQs, tech notes, downloads, techcasts and more to find answers to your questions as well as solutions to many common issues. See http://www.sybase.com/developer/library/sql-anywhere-techcorner.

The following table contains a list of the developer centers available for use on the SQL Anywhere Tech Corner:

Name	URL	Description
SQL Anywhere .NET Developer Center	www.sybase.com/developer/library/sql-anywhere-techcorner/microsoft-net	Get started and get answers to specific questions regarding SQL Anywhere and .NET development.
PHP Developer Center	www.sybase.com/developer/library/sql-anywhere-techcorner/php	An introduction to using the PHP (PHP Hypertext Preprocessor) scripting language to query your SQL Anywhere database.

Name	URL	Description
SQL Anywhere Windows Mobile Developer Center	www.sybase.com/developer/library/sql-anywhere-techcorner/windows-mobile	Get started and get answers to specific questions regarding SQL Anywhere and Windows Mobile development.

Working with database objects

This sectio provides procedures for adding database objects and setting database properties.

The SQL statements for creating, changing, and dropping database objects are called the **data definition language** (DDL). The definitions of the database objects form the database schema. A schema is the logical framework of the database.

To view information about procedures and triggers, see "Using procedures, triggers, and batches" on page 793.

To view conceptual information about database creation and design, see:

- "Creating a SQL Anywhere database" [*SQL Anywhere Server - Database Administration*]
- "Ensuring data integrity" on page 65
- "Using Sybase Central" [*SQL Anywhere Server - Database Administration*]
- "Using Interactive SQL" [*SQL Anywhere Server - Database Administration*]

Set properties for database objects

You can view or set the properties of a database and most database objects. Some of the properties selected when the database was created are non-configurable.

Use the properties windows in Sybase Central to view and set properties. If you do not use Sybase Central, specify the properties when you create the object with a CREATE statement. If the object already exists, use an ALTER statement to modify the properties.

To view and edit the properties of a database object (Sybase Central)

1. Use the SQL Anywhere 12 plug-in to connect to the database.
2. Open the folder in which the object resides.
3. Select the object. The object's properties appear in the right pane of Sybase Central.
4. In the right pane, click the appropriate tabs to edit the properties.

   You can also view and edit properties on the object's properties window. To view the properties window, right-click the object, and then choose **Properties**.

See also

- "Connection, database, and database server properties" [*SQL Anywhere Server - Database Administration*]

View system objects in a database

System objects such as system tables, system views, stored procedures, and domains hold information about database objects, and how they are related to each other. System views, procedures, and domains largely support Sybase Transact-SQL compatibility.

To display system objects in a database (Sybase Central)

1. Use the SQL Anywhere 12 plug-in to connect to the database as a user with DBA authority.
2. Select the database and choose **File** » **Configure Owner Filter**.
3. Select **SYS**, and **dbo**, and then click **OK**.

To browse system objects (SQL)

1. Connect to a database.
2. Execute a SELECT statement, querying the SYSOBJECT system view for a list of objects.

Example

The following SELECT statement queries the SYSOBJECT system view, and returns the list of all tables and views owned by SYS and dbo. A join is made to the SYSTAB system view to return the object name, and SYSUSER system view to return the owner name.

```
SELECT b.table_name "Object Name",
      c.user_name "Owner",
      b.object_id "ID",
      a.object_type "Type",
      a.status "Status"
  FROM ( SYSOBJECT a JOIN SYSTAB b
         ON a.object_id = b.object_id )
  JOIN SYSUSER c
WHERE c.user_name = 'SYS'
   OR c.user_name = 'dbo'
ORDER BY c.user_name, b.table_name;
```

See also

- “SYSOBJECT system view” [*SQL Anywhere Server - SQL Reference*]
- “SYSTAB system view” [*SQL Anywhere Server - SQL Reference*]
- “SYSUSER system view” [*SQL Anywhere Server - SQL Reference*]

Working with tables

When a database is first created, the only tables in the database are the system tables. System tables hold the database schema.

This section describes how to create, alter, and drop tables. You can execute the examples in Interactive SQL, but the SQL statements are independent of the administration tool you use. See "Editing result sets in Interactive SQL" [*SQL Anywhere Server - Database Administration*].

To make it easier for you to re-create the database schema when necessary, create command files to define the tables in your database. The command files should contain the CREATE TABLE and ALTER TABLE statements.

For more information about groups, tables, and connecting as another user, see "Referring to tables owned by groups" [*SQL Anywhere Server - Database Administration*], and "Database object names and prefixes" [*SQL Anywhere Server - Database Administration*].

Create tables

When a database is first created, the only tables in the database are the system tables, which hold the database schema. You can use SQL statements in Interactive SQL or Sybase Central to create new tables to hold your data.

There are two types of tables that you can create:

- **Base table** A table that holds persistent data. The table and its data continue to exist until you explicitly delete the data or drop the table. It is called a base table to distinguish it from temporary tables and views.
- **Temporary table** Data in a temporary table is held for a single connection only. Global temporary table definitions (but not data) are kept in the database until dropped. Local temporary table definitions and data exist for the duration of a single connection only. See "Working with temporary tables" on page 18.

Tables consist of rows and columns. Each column carries a particular kind of information, such as a phone number or a name, while each row specifies a particular entry.

To create a table (Sybase Central)

1. Use the SQL Anywhere 12 plug-in to connect to the database as a user with DBA authority.
2. In the left pane, right-click **Tables** and choose **New** » **Table**.
3. Follow the instructions in the **Create Table Wizard**.
4. In the right pane, click the **Columns** tab and configure your table.
5. Choose **File** » **Save**.

To create a table (SQL)

1. Connect to the database as a user with DBA authority.

2. Execute a CREATE TABLE statement.

Example

The following statement creates a new table to describe qualifications of employees within a company. The table has columns to hold an identifying number, a name, and a type (technical or administrative) for each skill.

```
CREATE TABLE Skills (
   SkillID INTEGER NOT NULL,
   SkillName CHAR( 20 ) NOT NULL,
   SkillType CHAR( 20 ) NOT NULL
);
```

See "CREATE TABLE statement" [*SQL Anywhere Server - SQL Reference*].

Altering tables

This section describes how to alter the structure or column definitions of a table. For example, you can add columns, change various column attributes, or drop columns entirely.

You can perform table alteration tasks on the **SQL** tab in the right pane of Sybase Central. In Interactive SQL, you can perform them using the ALTER TABLE statement.

For information about altering database object properties, see "Set properties for database objects" on page 1.

For information about granting and revoking table permissions, see "Granting permissions on tables" [*SQL Anywhere Server - Database Administration*], and "Revoking user permissions and authorities" [*SQL Anywhere Server - Database Administration*].

Table alterations and view dependencies

Before altering a table, you may want to determine whether there are views dependent on a table, using the sa_dependent_views system procedure. See "sa_dependent_views system procedure" [*SQL Anywhere Server - SQL Reference*].

If you are altering the schema of a table with dependent views, there may be additional steps to make, as noted in the following sections.

- **Dependent regular views** When you alter the schema of a table, the definition for the table in the database is updated. If there are dependent regular views, the database server automatically recompiles them after you perform the table alteration. If the database server cannot recompile a dependent regular view after making a schema change to a table, it is likely because the change you made invalidated the view definition. In this case, you must correct the view definition. See "Alter regular views" on page 28.

- **Dependent materialized views** If there are dependent materialized views, you must disable them before making the table alteration, and then re-enable them after making the table alteration. If you cannot re-enable a dependent materialized view after making a schema change to a table, it is likely

because the change you made invalidated the materialized view definition. In this case, you must drop the materialized view and then create it again with a valid definition, or make suitable alterations to the underlying table before trying to re-enable the materialized view. See "Create materialized views" on page 43.

For an overview of how altering database objects affects view dependencies, see "View dependencies" on page 22.

Alter tables (Sybase Central)

You can alter tables in Sybase Central on the **Columns** tab in the right pane. For example, you can add or drop columns, change column definitions, or change table or column properties. Altering tables fails if there are any dependent materialized views; you must first disable dependent materialized views. Once your table alterations are complete, you must re-enable the dependent materialized views. See "View dependencies" on page 22.

Use the sa_dependent_views system procedure to determine if there are dependent materialized views. See "sa_dependent_views system procedure" [*SQL Anywhere Server - SQL Reference*].

To alter an existing table (Sybase Central)

1. Use the SQL Anywhere 12 plug-in to connect to the database as a user with DBA authority or as the owner of the table.
2. If you are making a schema change and there are materialized views dependent on the table, disable each one as follows:
   a. In the left pane, double-click **Views**.
   b. Right-click the materialized view and choose **Disable**.
3. Double-click **Tables** and select the table you want to alter.
4. In the right pane, click the **Columns** tab and alter the table settings.
5. Choose **File** » **Save**.
6. If you disabled materialized views, re-enable and initialize each one. See "Enable and disable materialized views" on page 51.

Tips

- You can add columns by selecting a table's **Columns** tab and choosing **File** » **New Column**.
- You can drop columns by selecting the column on the **Columns** tab and choosing **Edit** » **Delete**.
- You can copy a column to a table by selecting the column on the **Columns** tab in the right pane and then clicking **Copy**. Select the table, click the **Columns** tab in the right pane, and then click **Paste**.
- You can undo any unsaved edits to a column, by selecting the column in the **Columns** tab in the right pane, and then choosing **Edit** » **Undo.**.
- It is also necessary to click **Save** or choose **File** » **Save**. Changes are not made to the table until then.

See also

- "Enable and disable materialized views" on page 51
- "Ensuring data integrity" on page 65
- "View dependencies" on page 22

Alter tables (SQL)

You can alter tables in Interactive SQL using the ALTER TABLE statement. If you use a clause 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. See "View dependencies" on page 22.

Use the sa_dependent_views system procedure to determine if there are dependent materialized views. See "sa_dependent_views system procedure" [*SQL Anywhere Server - SQL Reference*].

To alter an existing table (SQL)

1. Connect to the database as a user with DBA authority.
2. If you are performing a schema-altering operation on tables with dependent materialized views, and use a clause other than ADD FOREIGN KEY with the ALTER TABLE statement, use the ALTER MATERIALIZED VIEW ... DISABLE statement to disable each dependent materialized view. You do not need to disable dependent regular views.
3. Execute an ALTER TABLE statement to perform the table alteration.

   The definition for the table in the database is updated.
4. If you disabled any materialized views, use the ALTER MATERIALIZED VIEW ... ENABLE statement to re-enable them.

Examples

These examples show how to change the structure of the database. The ALTER TABLE statement can change just about anything pertaining to a table—you can use it to add or drop foreign keys, change

columns from one type to another, and so on. In all these cases, once you make the change, stored procedures, views, and any other items referring to this table may no longer work.

The following command adds a column to the Skills table to allow space for an optional description of the skill:

```
ALTER TABLE Skills
ADD SkillDescription CHAR( 254 );
```

You can also alter column attributes with the ALTER TABLE statement. The following statement shortens the SkillDescription column from a maximum of 254 characters to a maximum of 80:

```
ALTER TABLE Skills
ALTER SkillDescription CHAR( 80 );
```

By default, an error occurs if there are entries that are longer than 80 characters. The string_rtruncation option can be used to change this behavior. See "string_rtruncation option" [*SQL Anywhere Server - Database Administration*].

The following statement changes the name of the SkillType column to Classification:

```
ALTER TABLE Skills
RENAME SkillType TO Classification;
```

The following statement drops the Classification column.

```
ALTER TABLE Skills
DROP Classification;
```

The following statement changes the name of the entire table:

```
ALTER TABLE Skills
RENAME Qualification;
```

See also

- "ALTER TABLE statement" [*SQL Anywhere Server - SQL Reference*]
- "ALTER VIEW statement" [*SQL Anywhere Server - SQL Reference*]
- "Alter regular views" on page 28
- "Enable and disable materialized views" on page 51
- "ALTER MATERIALIZED VIEW statement" [*SQL Anywhere Server - SQL Reference*]
- "Ensuring data integrity" on page 65
- "View dependencies" on page 22

Dropping tables

You can use either Sybase Central or Interactive SQL to drop tables from a database. In Interactive SQL, deleting a table is also called dropping it.

You cannot drop a table that is being used as an article in a SQL Remote publication. If you try to do this in Sybase Central, an error appears. Also, if you are dropping a table that has dependent views, there may be additional steps to make, as noted in the following sections.

Table deletions and view dependencies

When you drop a table, its definition is removed from the database. If there are dependent regular views, the database server attempts to recompile and re-enable them after you perform the table alteration. If it cannot, it is likely because the table deletion invalidated the definition for the view. In this case, you must correct the view definition. See "Alter regular views" on page 28.

If there are dependent materialized views, subsequent refreshing will fail because its definition is no longer valid. In this case, you must drop the materialized view and then create it again with a valid definition. See "Create materialized views" on page 43.

Before altering a table, you may want to determine whether there are views dependent on a table, using the sa_dependent_views system procedure. See "sa_dependent_views system procedure" [*SQL Anywhere Server - SQL Reference*].

For an overview of how table deletions affect view dependencies, see "View dependencies" on page 22.

Drop tables

To drop a table (Sybase Central)

1. Use the SQL Anywhere 12 plug-in to connect to the database as a user with DBA authority or as the owner of the table.
2. If you are dropping a table on which materialized views depend, disable each materialized view:
   a. In the left pane, double-click **Views**.
   b. Right-click the materialized view and choose **Disable**.
3. Double-click **Tables**.
4. Right-click the table and choose **Delete**.
5. Click **Yes**.

To drop a table (SQL)

1. Connect to the database as a user with DBA authority, or as the owner of the table.
2. If you are dropping a table on which materialized views depend, disable each materialized view using the ALTER MATERIALIZED VIEW ... DISABLE statement.
3. Execute a DROP TABLE statement.

Example

The following DROP TABLE command deletes all the records in the Skills table and then removes the definition of the Skills table from the database.

```
DROP TABLE Skills;
```


Like the CREATE statement, the DROP statement automatically executes a COMMIT statement before and after dropping the table. This makes all changes to the database since the last COMMIT or ROLLBACK permanent. The DROP statement also drops all indexes on the table. See "DROP TABLE statement" [*SQL Anywhere Server - SQL Reference*].

Browsing the data held in tables

You can use Sybase Central or Interactive SQL to browse and the data held within the tables of a database.

If you are working in Sybase Central, view the data in a table by selecting the table and clicking the **Data** tab in the right pane.

If you are working in Interactive SQL, execute the following statement:

```
SELECT * FROM table-name;
```

You can edit the data in the table from the Interactive SQL **Results** tab or from the table's **Data** tab in Sybase Central.

See also

- "Using Interactive SQL" [*SQL Anywhere Server - Database Administration*]
- "Editing result sets in Interactive SQL" [*SQL Anywhere Server - Database Administration*]

Managing primary keys

A **primary key** comprises a single column, or a set of columns combined together, whose values identify unique rows in a table. Primary key values do not change over the life of the data in the row. Because uniqueness is essential to good database design, it is best to specify a primary key when you define the table.

It is recommended that you do not use approximate data types such as FLOAT and DOUBLE for primary keys or for columns with unique constraints. Approximate numeric data types are subject to rounding errors after arithmetic operations.

This section describes how to create and edit primary keys in your database. You can use either Sybase Central or Interactive SQL to perform these tasks.

Column order in multi-column primary keys
Primary key column order is determined by the primary key and foreign key clauses in the CREATE TABLE statement. It is not based on the order of the columns as specified in the primary key declaration of the CREATE TABLE statement.

Manage primary keys (Sybase Central)

A primary key is a column, or set of columns, that is used to identify unique rows in a table. Primary keys are typically created at table creation time; however, they can be modified at a later time. In Sybase Central, you access the primary key for a table in one of two ways:

- right-clicking the table and choosing **Set Primary Key**, which starts the **Set Primary Key Wizard**. The **Set Primary Key Wizard** also allows you to change the order of columns of an existing primary key.
- selecting the table in the left pane, and then choosing the **Constraints** tab in the right pane.

To configure a primary key (Sybase Central)

1. Use the SQL Anywhere 12 plug-in to connect to the database as a user with DBA authority or as the owner of the table.
2. In the left pane, double-click **Tables**.
3. Right-click the table and choose **Set Primary Key**.
4. Follow the instructions in the **Set Primary Key Wizard**.

Manage primary keys (SQL)

You can create and alter the primary key in Interactive SQL using the CREATE TABLE and ALTER TABLE statements. These statements let you set many table attributes, including column constraints and checks.

Columns in the primary key cannot contain NULL values. You must specify NOT NULL on columns in the primary key.

To add a primary key (SQL)

1. Connect to the database as a user with DBA authority.
2. Execute an ALTER TABLE statement for the table on which you want to configure the primary key.

To modify a primary key (SQL)

1. Connect to the database as a user with DBA authority.
2. Execute an ALTER TABLE statement to drop the existing primary key.
3. Execute an ALTER TABLE statement to add a primary key.

To delete a primary key (SQL)

1. Connect to the database as a user with DBA authority.

2. Execute an ALTER TABLE statement using the DELETE PRIMARY KEY clause.

Example 1

The following statement creates a table named Skills, and assigns the SkillID column as the primary key:

```
CREATE TABLE Skills (
   SkillID INTEGER NOT NULL,
   SkillName CHAR( 20 ) NOT NULL,
   SkillType CHAR( 20 ) NOT NULL,
   PRIMARY KEY( SkillID )
);
```

The primary key values must be unique for each row in the table, which in this case means that you cannot have more than one row with a given SkillID. Each row in a table is uniquely identified by its primary key.

If you want to change the primary key to use SkillID and SkillName columns together for the primary key, you must first delete the primary key that you created, and then add the new primary key:

```
ALTER TABLE Skills DELETE PRIMARY KEY
ALTER TABLE Skills ADD PRIMARY KEY ( SkillID, SkillName );
```

See "ALTER TABLE statement" [*SQL Anywhere Server - SQL Reference*], and "Manage primary keys (Sybase Central)" on page 9.

Managing foreign keys

This section describes how to create and edit foreign keys in your database. You can use either Sybase Central or Interactive SQL to perform these tasks.

Foreign keys are used to relate values in a child table (or foreign table) to those in a parent table (or primary table). A table can have multiple foreign keys that refer to multiple parent tables linking various types of information.

Example

The SQL Anywhere sample database has one table holding employee information and one table holding department information. The Departments table has the following columns:

- **DepartmentID** An ID number for the department. This is the primary key for the table.
- **DepartmentName** The name of the department.
- **DepartmentHeadID** The employee ID for the department manager.

To find the name of a particular employee's department, there is no need to put the name of the employee's department into the Employees table. Instead, the Employees table contains a column, DepartmentID, holding a value that matches one of the DepartmentID values in the Departments table.

The DepartmentID column in the Employees table is a foreign key to the Departments table. A foreign key references a particular row in the table containing the corresponding primary key.

The Employees table (which contains the foreign key in the relationship) is therefore called the **foreign table** or **referencing table**. The Departments table (which contains the referenced primary key) is called the **primary table** or the **referenced table**.

Manage foreign keys (Sybase Central)

In Sybase Central, the foreign key of a table appears on the **Constraints** tab, which is located on the right pane when a table is selected.

You create a foreign key relationship when you create the child table (that is, before inserting data in the child table). The foreign key relationship then acts as a constraint; for new rows inserted in the child table, the database server checks to see if the value you are inserting into the foreign key column matches a value in the primary table's primary key.

After you have created a foreign key, you can keep track of it on each table's **Constraints** tab in the right pane; this tab displays any foreign tables that reference the currently selected table.

To create a new foreign key (Sybase Central)

1. Use the SQL Anywhere 12 plug-in to connect to the database as a user with DBA authority or as the owner of the table.
2. In the left pane, double-click **Tables**.
3. Right-click the table and choose **New** » **Foreign Key**.
4. Follow the instructions in the **Create Foreign Key Wizard**.

To delete a foreign key (Sybase Central)

1. Use the SQL Anywhere 12 plug-in to connect to the database as a user with DBA authority or as the owner of the table.
2. In the left pane, double-click **Tables**.
3. Select the table for which you want to delete a foreign key.
4. In the right pane, click the **Constraints** tab.
5. Right-click the foreign key and choose **Delete**.
6. Click **Yes**.

For any given table, you can also view a list of tables that reference the table using a foreign key.

To display a list of tables that reference a given table (Sybase Central)

1. Use the SQL Anywhere 12 plug-in to connect to the database as a user with DBA authority or as the owner of the table.
2. In the left pane, double-click **Tables**.
3. Click the table.
4. In the right pane, click the **Referencing Constraints** tab.

Tips

When you create a foreign key using the wizard, you can set properties for the foreign key. To view properties after the foreign key is created, select the foreign key on the **Constraints** tab and then choose **File** » **Properties**.

You can view the properties of a referencing foreign key by selecting the table on the **Referencing Constraints** tab and then choosing **File** » **Properties**.

Manage foreign keys (SQL)

A table can only have one primary key defined, but it can have many foreign keys. You can create and alter foreign keys in Interactive SQL using the CREATE TABLE and ALTER TABLE statements. These statements let you set many table attributes, including column constraints and checks.

To create a foreign key (SQL)

1. Connect to the database as a user with DBA authority.
2. Execute an ALTER TABLE statement.

Omitting column names at foreign key creation (SQL)

Foreign key column names are paired with primary key column names according to position in the two lists in a one-to-one manner. If the primary table column names are not specified when defining the foreign key, then the primary key columns are used. For example, suppose you create two tables as follows:

```
CREATE TABLE Table1( a INT, b INT, c INT, PRIMARY KEY ( a, b ) );
CREATE TABLE Table2( x INT, y INT, z INT, PRIMARY KEY ( x, y ) );
```

Then, you create a foreign key fk1 as follows, specifying exactly how to pair the columns between the two tables:

```
ALTER TABLE Table2 ADD FOREIGN KEY fk1( x,y ) REFERENCES Table1( a, b );
```

Using the following statement, you create a second foreign key, fk2, by specifying only the foreign table columns. The database server automatically pairs these two columns to the first two columns in the primary key on the primary table.

```
ALTER TABLE Table2 ADD FOREIGN KEY fk2( x, y ) REFERENCES Table1;
```

Using the following statement, you create a foreign key without specifying columns for either the primary or foreign table:

```
ALTER TABLE Table2 ADD FOREIGN KEY fk3 REFERENCES Table1;
```

Since you did not specify referencing columns, the database server looks for columns in the foreign table (Table2) with the same name as columns in the primary table (Table1). If they exist, it ensures that the data types match and then creates the foreign key using those columns. If columns do not exist, they are created in Table2. In this example, Table2 does NOT have columns called a and b so they are created with the same data types as Table1.a and Table1.b. These automatically-created columns cannot become part of the primary key of the foreign table.

Examples

In the following example, you create a table called Skills which contains a list of possible skills, and then create a table called EmployeeSkills that has a foreign key relationship to the Skills table. Notice that EmployeeSkills.SkillID has a foreign key relationship with the primary key column (Id) of the Skills table.

```
CREATE TABLE Skills (
   Id INTEGER PRIMARY KEY,
   SkillName CHAR(40),
   Description CHAR(100)
);
CREATE TABLE EmployeeSkills (
   EmployeeID INTEGER NOT NULL,
   SkillID INTEGER NOT NULL,
   SkillLevel INTEGER NOT NULL,
   PRIMARY KEY( EmployeeID ),
   FOREIGN KEY (SkillID) REFERENCES Skills ( Id )
);
```

You can also add a foreign key to a table after it has been created, using the ALTER TABLE statement. In the following example, you create tables similar to those created in the previous example, except you add the foreign key after creating the table.

```
CREATE TABLE Skills2 (
   ID INTEGER PRIMARY KEY,
   SkillName CHAR(40),
   Description CHAR(100)
);
CREATE TABLE EmployeeSkills2 (
   EmployeeID INTEGER NOT NULL,
   SkillID INTEGER NOT NULL,
   SkillLevel INTEGER NOT NULL,
   PRIMARY KEY( EmployeeID ),
);
ALTER TABLE EmployeeSkills2
   ADD FOREIGN KEY SkillFK ( SkillID )
   REFERENCES Skills2 ( ID );
```

You can specify properties for the foreign key as you create it. For example, the following statement creates the same foreign key as in Example 2, but it defines the foreign key as NOT NULL along with restrictions for when you update or delete.

```
ALTER TABLE Skills2
ADD NOT NULL FOREIGN KEY SkillFK ( SkillID )
REFERENCES Skills2 ( ID )
ON UPDATE RESTRICT
ON DELETE RESTRICT;
```


See also

- “Manage foreign keys (Sybase Central)” on page 12
- “CREATE TABLE statement” [*SQL Anywhere Server - SQL Reference*]
- “ALTER TABLE statement” [*SQL Anywhere Server - SQL Reference*]

Working with computed columns

A computed column is a column whose value is an expression that can refer to the values of other columns, called **dependent columns**, in the same row. Computed columns are especially useful in situations where you want to index a complex expression that can include the values of one or more dependent columns. The database server will use the computed column wherever it see an expression that matches the computed column's COMPUTE expression; this includes the SELECT list and predicates. However, if the query expression contains a special value, such as CURRENT TIMESTAMP, this matching does not occur. For a list of special values that prevent matching, see “Special values” [*SQL Anywhere Server - SQL Reference*].

Do not use TIMESTAMP WITH TIME ZONE columns as computed columns. The value of the time_zone_adjustment option varies between connections based on their location and the time of year, resulting in incorrect results and unexpected behavior when the values are computed.

During query optimization, the SQL Anywhere optimizer automatically attempts to transform a predicate involving a complex expression into one that simply refers to the computed column's definition. For example, suppose that you want to query a table containing summary information about product shipments:

```
CREATE TABLE Shipments(
   ShipmentID INTEGER NOT NULL PRIMARY KEY,
   ShipmentDate TIMESTAMP,
   ProductCode CHAR(20) NOT NULL,
   Quantity INTEGER NOT NULL,
   TotalPrice DECIMAL(10,2) NOT NULL
);
```

In particular, the query is to return those shipments whose average cost is between two and four dollars. The query could be written as follows:

```
SELECT *
   FROM Shipments
   WHERE ( TotalPrice / Quantity ) BETWEEN 2.00 AND 4.00;
```

However, in the query above, the predicate in the WHERE clause is not sargable since it does not refer to a single base column. See “Using predicates in queries” on page 551. If the size of the Shipments table is relatively large, an indexed retrieval might be appropriate rather than a sequential scan. To benefit from an indexed retrieval, create a computed column named AverageCost for the Shipments table, and then create an index on the column, as follows:

```
ALTER TABLE Shipments
   ADD AverageCost DECIMAL(21,13)
   COMPUTE( TotalPrice / Quantity );
 CREATE INDEX IDX_average_cost
   ON Shipments( AverageCost ASC );
```

Choosing the type of the computed column is important; the SQL Anywhere optimizer replaces only complex expressions by a computed column if the data type of the expression in the query precisely matches the data type of the computed column. To determine what the type of any expression is, you can use the EXPRTYPE built-in function that returns the expression's type in ready-to-use SQL terms:

```
SELECT EXPRTYPE(
 'SELECT ( TotalPrice/Quantity ) AS X FROM Shipments', 1 )
   FROM DUMMY;
```

For the Shipments table, the above query returns decimal(21,13). During optimization, the SQL Anywhere optimizer rewrites the query above as follows:

```
SELECT *
   FROM Shipments
   WHERE AverageCost
   BETWEEN 2.00 AND 4.00;
```

In this case, the predicate in the WHERE clause is now a sargable one, making it possible for the optimizer to choose an indexed scan, using the new IDX_average_cost index, for the query's access plan.

Altering computed column expressions

You can change the expression used in a computed column with the ALTER TABLE statement. The following statement changes the expression that a computed column is based on.

```
ALTER TABLE table-name
ALTER column-name
SET COMPUTE ( new-expression );
```

The column is recalculated when this statement is executed. If the new expression is invalid, the ALTER TABLE statement fails.

The following statement stops a column from being a computed column.

```
ALTER TABLE
table-name
ALTER column-name
DROP COMPUTE;
```

Existing values in the column are not changed when this statement is executed, but they are no longer updated automatically.

Inserting into and updating computed columns

Considerations regarding inserting into, and updating, computed columns include the following:

- **Direct inserts and updates** You should not use INSERT or UPDATE statements to put values into computed columns since the values may not reflect the intended computation. Also, manually inserted or updated data in computed columns may be changed later when the column is recomputed.

- **Column dependencies** It is strongly recommended that you not use triggers to set the value of a column referenced in the definition of a computed column (for example, to change a NULL value to a not-NULL value), as this can result in the value of the computed column not reflecting its intended computation.

- **Listing column names** You must always explicitly specify column names in INSERT statements on tables with computed columns.

- **Triggers** If you define triggers on a computed column, any INSERT or UPDATE statement that affects the column fires the triggers.

Although you can use INSERT, UPDATE, or LOAD TABLE statements to insert values in computed columns, this is neither the recommended nor intended application of this feature.

The LOAD TABLE statement permits the *optional* computation of computed columns. Suppressing computation during a load operation may make performing complex unload/reload sequences faster. It can also be useful when the value of a computed column must stay constant, even though the COMPUTE expression refers a non-deterministic value, such as CURRENT TIMESTAMP.

Avoid changing the values of dependent columns in triggers as it may cause the value of the computed column to be inconsistent with the column definition.

If a computed column x depends on a column y that is declared not-NULL, then an attempt to set y to NULL will be rejected with an error before triggers fire.

Recalculating computed columns

Values of computed columns are automatically maintained by the database server as rows are inserted and updated. Most applications should never need to update or insert computed column values directly.

Computed columns are recalculated under the following circumstances:

- Any column is deleted, added, or renamed.

- The table is renamed.

- Any column's data type or COMPUTE clause is modified.

- A row is inserted.

- A row is updated.

Computed columns are *not* recalculated under the following circumstances:

- The computed column is queried.

- Values are changed in columns on which the computed column depends.

Copying tables or columns within or between databases

With Sybase Central, you can copy existing tables or columns and insert them into another location in the same database or into a completely different database. See "Copying database objects in the SQL Anywhere 12 plug-in" [*SQL Anywhere Server - Database Administration*].

If you are not using Sybase Central:

- To insert SELECT statement results into a given location, see "SELECT statement" [*SQL Anywhere Server - SQL Reference*].
- To insert a row or selection of rows from elsewhere in the database into a table, see "INSERT statement" [*SQL Anywhere Server - SQL Reference*].

Working with temporary tables

Temporary tables are stored in the temporary file. Pages from the temporary file can be cached, just as pages from any other dbspace can. Operations on temporary tables are never written to the transaction log. There are two types of temporary tables: **local temporary** tables and **global temporary** tables.

See also

- "CREATE TABLE statement" [*SQL Anywhere Server - SQL Reference*]
- "DECLARE LOCAL TEMPORARY TABLE statement" [*SQL Anywhere Server - SQL Reference*]

Local temporary tables

A local temporary table exists only for the duration of a connection or, if defined inside a compound statement, for the duration of the compound statement.

Two local temporary tables within the same scope cannot have the same name. If you create temporary table with the same name as a base table, the base table only becomes visible within the connection once the scope of the local temporary table ends. A connection cannot create a base table with the same name as an existing temporary table.

Global temporary tables

A global temporary table remains in the database until explicitly removed using a DROP TABLE statement. The term global is used to indicate that multiple connections from the same or different applications can use the table at the same time. The characteristics of global temporary tables are as follows:

- The definition of the table is recorded in the catalog and persists until the table is explicitly dropped.
- Inserts, updates, and deletes on the table are not recorded in the transaction log.
- Column statistics for the table are maintained in memory by the database server.

There are two types of global temporary tables: **non-shared** and **shared**. Normally, a global temporary table is non-shared; that is, each connection sees only its own rows in the table. When a connection ends, rows for that connection are deleted from the table.

When a global temporary table is shared, all the table's data is shared across all connections. To create a shared global temporary table, you specify the SHARE BY ALL clause at table creation. In addition to the general characteristics for global temporary tables, the following characteristics apply to shared global temporary tables:

- The content of the table persists until explicitly deleted or until the database is shut down.
- On database startup, the table is empty.
- Row locking behavior on the table is the same as for a base table.

Non-transactional temporary tables

Temporary tables can be declared as non-transactional using the NOT TRANSACTIONAL clause of the CREATE TABLE statement. 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 COMMIT or ROLLBACK, or if the table contains many rows. Changes to non-transactional temporary tables are not affected by COMMIT or ROLLBACK.

Create temporary tables

You can create temporary tables either with SQL statements or with Sybase Central.

To create a table (Sybase Central)

1. Use the SQL Anywhere 12 plug-in to connect to the database as a user with DBA authority or as the owner of the table.
2. Right-click **Tables** and choose **New** » **Global Temporary Table**.
3. Follow the instructions in the **Create Global Temporary Table Wizard**.
4. In the right pane, click the **Columns** tab and configure the table.
5. Choose **File** » **Save**.

To create a table (SQL)

1. Connect to the database as a user with DBA authority.
2. Execute a CREATE TABLE statement or DECLARE LOCAL TEMPORARY TABLE statement.

See also

- "CREATE TABLE statement" [*SQL Anywhere Server - SQL Reference*]
- "DECLARE LOCAL TEMPORARY TABLE statement" [*SQL Anywhere Server - SQL Reference*]

Referencing temporary tables within procedures

Sharing a temporary table between procedures can cause problems if the table definitions are inconsistent. For example, suppose you have two procedures procA and procB, both of which define a temporary table, temp_table, and call another procedure called sharedProc. Neither procA nor procB has been called yet, so the temporary table does not yet exist.

Now, suppose that the procA definition for temp_table is slightly different than the definition in procB—while both used the same column names and types, the column order is different.

When you call procA, it returns the expected result. However, when you call procB, it returns a different result.

This is because when procA was called, it created temp_table, and then called sharedProc. When sharedProc was called, the SELECT statement inside of it was parsed and validated, and then a parsed representation of the statement is cached so that it can be used again when another SELECT statement is executed. The cached version reflects the column ordering from the table definition in procA.

Calling procB causes the temp_table to be recreated, but with different column ordering. When procB calls sharedProc, the database server uses the cached representation of the SELECT statement. So, the results are different.

You can avoid this from happening by doing one of the following:

- ensure that temporary tables used in this way are defined consistently
- consider using a global temporary table instead

Working with views

A View is a computed tables defined by the result set of its view definition, which is expressed as a SQL query. You can use views to show database users exactly the information you want to present, in a format you can control. SQL Anywhere supports two types of views: **regular views** and **materialized views**.

The definition for each view in the database is stored in the ISYSVIEW system table. See "SYSVIEW system view" [*SQL Anywhere Server - SQL Reference*].

Documentation conventions

In the SQL Anywhere documentation, the term **regular view** is used to describe a view that is recomputed each time you reference the view, and the result set is not stored on disk. This is the most commonly used type of view. Most of the documentation refers to regular views.

The term **materialized view** is used to describe a view whose result set is precomputed and materialized on disk similar to the contents of a base table.

The meaning of the term **view** (by itself) in the documentation is context-based. When used in a section that is talking about common aspects of regular and materialized views, it refers to both regular and materialized views. If the term is used in documentation for materialized views, it refers to materialized views, and likewise for regular views.

Comparing regular views, materialized views, and base tables

The following table compares regular views, materialized views, and base tables:

Capability	Regular views	Materialized views	Base tables
Allow access permissions	Yes	Yes	Yes
Allow SELECT	Yes	Yes	Yes
Allow UPDATE	Some	No	Yes
Allow INSERT	Some	No	Yes
Allow DELETE	Some	No	Yes
Allow dependent views	Yes	Yes	Yes
Allow indexes	No	Yes	Yes
Allow integrity constraints	No	No	Yes
Allow keys	No	No	Yes

Benefits of using views

Views let you tailor access to data in the database. Tailoring access serves several purposes:

- **Efficient resource use** Regular views do not require additional storage space for data; they are recomputed each time you invoke them. Materialized views require disk space, but do not need to be recomputed each time they are invoked. Materialized views can improve response time in

environments where the database is large, and the database server processes frequent, repetitive requests to join the same tables.

- **Improved security** By allowing access to only the information that is relevant.
- **Improved usability** By presenting users and application developers with data in a more easily understood form than in the base tables.
- **Improved consistency** By centralizing the definition of common queries in the database.

View dependencies

A view definition can refer to other objects including columns, tables, and other views. When a view makes a reference to another object, the view is called a **referencing object** and the object to which it refers is called a **referenced object**. Further, a referencing object is said to be **dependent** on the objects to which it refers.

The set of referenced objects for a given view includes all the objects to which it refers either directly or indirectly. For example, a view can indirectly refer to a table, by referring to another view that references that table.

Consider the following set of tables and views:

```
CREATE TABLE t1 ( c1 INT, c2 INT );
CREATE TABLE t2( c3 INT, c4 INT );
CREATE VIEW v1 AS SELECT * FROM t1;
CREATE VIEW v2 AS SELECT c3 FROM t2;
CREATE VIEW v3 AS SELECT c1, c3 FROM v1, v2;
```

The following view dependencies can be determined from the definitions above:

- View v1 is dependent on each individual column of t1, and on t1 itself.
- View v2 is dependent on t2.c3, and on t2 itself.
- View v3 is dependent on columns t1.c1 and t2.c3, tables t1 and t2, and views v1 and v2.

The database server keeps track of columns, tables, and views referenced by a given view. The database server uses this dependency information to ensure that schema changes to referenced objects do not leave a referencing view in an unusable state. The following tables explains how view dependencies affects regular and materialized views.

Dependencies and schema-altering changes

An attempt to alter the schema defined for a table or view requires that the database server consider if there are dependent views impacted by the change. Examples of schema-altering operations include:

- Dropping a table, view, materialized view, or column

- Renaming a table, view, materialized view, or column
- Adding, dropping, or altering columns
- Altering a column's data type, size, or nullability
- Disabling views or table view dependencies

When you attempt a schema-altering operation, the following events occur:

1. The database server generates a list of views that depend directly or indirectly upon the table or view being altered. Views with a DISABLED status are ignored.

   If any of the dependent views are materialized views, the request fails, an error is returned, and the remaining events do not occur. You must explicitly disable dependent materialized views before you can proceed with the schema-altering operation. See "Enable and disable materialized views" on page 51.

2. The database server obtains exclusive schema locks on the object being altered, and on all dependent regular views.
3. The database server sets the status of all dependent regular views to INVALID.
4. The database server performs the schema-altering operation. If the operation fails, the locks are released, the status of dependent regular views is reset to VALID, an error is returned, and the following step does not occur.
5. The database server recompiles the dependent regular views, setting each view's status to VALID when successful. If compilation fails for any regular view, the status of that view remains INVALID. Subsequent requests for an INVALID regular view causes the database server to attempt to recompile the view. If subsequent attempts fail, it is likely that an alteration is required on the INVALID view, or on an object upon which it depends.

Dependencies and schema-altering changes (regular views)

- A regular view can reference tables or views, including materialized views.
- When you change the schema of a table or view, the database automatically attempts to recompile all referencing regular views.
- When you disable or drop a view or table, all dependent regular views are automatically disabled.
- You can use the DISABLE VIEW DEPENDENCIES clause of the ALTER TABLE statement to disable dependent regular views.

Dependencies and schema-altering changes (materialized views)

- A materialized view can only reference base tables.
- Schema changes to a base table are not permitted if it is referenced by any enabled materialized views. You can add foreign keys to the table (for example, ALTER TABLE ADD FOREIGN KEY).

- Before you drop a table, you must disable or drop all dependent materialized views.
- The DISABLE VIEW DEPENDENCIES clause of the ALTER TABLE statement does not impact materialized views. To disable a materialized view, you must use the ALTER MATERIALIZED VIEW ... DISABLE statement.
- Once you disable a materialized view, you must explicitly re-enable it, for example using the ALTER MATERIALIZED VIEW ... ENABLE statement.

View dependency information in the catalog

The database server keeps track of direct dependencies. A direct dependency is when one object directly references another object in its definition. The database server uses direct dependency information to determine indirect dependencies as well. For example, suppose View A references View B, which in turn references Table C. In this case, View A is directly dependent on View B, and indirectly dependent on Table C.

The SYSDEPENDENCY system view stores dependency information. Each row in the SYSDEPENDENCY system view describes a dependency between two database objects. See "SYSDEPENDENCY system view" [*SQL Anywhere Server - SQL Reference*].

You can also use the sa_dependent_views system procedure to return a list of views that are dependent on a given table or view. See "sa_dependent_views system procedure" [*SQL Anywhere Server - SQL Reference*].

Working with regular views

When you browse data, a query operates on one or more database objects and produces a result set. Just like a base table, a result set from a query has columns and rows. A view gives a name to a particular query, and holds the definition in the database system tables.

Suppose you frequently need to list the number of employees in each department. You can get this list with the following statement:

```
SELECT DepartmentID, COUNT(*)
FROM Employees
GROUP BY DepartmentID;
```

You can create a view containing the results of this statement using either Sybase Central or Interactive SQL.

Restrictions on SELECT statements for regular views

There are some restrictions on the SELECT statements you can use as regular views. In particular, you cannot use an ORDER BY clause in the SELECT query. A characteristic of relational tables is that there is no significance to the ordering of the rows or columns, and using an ORDER BY clause would impose an order on the rows of the view. You can use the GROUP BY clause, subqueries, and joins in view definitions.

To develop a view, tune the SELECT query by itself until it provides exactly the results you need in the format you want. Once you have the SELECT statement just right, you can add a phrase in front of the query to create the view:

```
CREATE VIEW view-name AS query;
```

Updating regular views

Updates can be performed on a view using the UPDATE, INSERT, or DELETE statements if the query specification defining the view is updatable. Views are considered inherently *non-updatable* if their definition includes any one of the following in their query specification:

- UNION, EXCEPT, or INTERSECT.
- DISTINCT clause.
- GROUP BY clause.
- WINDOW clause.
- FIRST, TOP, or LIMIT clause.
- aggregate functions.
- more than one table in the FROM clause, when ansi_update_constraints option is set to 'Strict' or Cursor'. See “ansi_update_constraints option” [*SQL Anywhere Server - Database Administration*].
- ORDER BY clause, when ansi_update_constraints option is set to 'Strict' or Cursor'. See “ansi_update_constraints option” [*SQL Anywhere Server - Database Administration*].
- all *select-list* items are not base table columns.

Copying regular views

In Sybase Central, you can copy views between databases. To do so, select the view in the right pane of Sybase Central and drag it to the **Views** folder of another connected database. A new view is then created and the original view's definition is copied to it. Note that only the view definition is copied to the new view. Other view properties, such as permissions, are not copied.

Using the WITH CHECK OPTION option

The WITH CHECK OPTION clause is useful for controlling what data is changed when inserting into, or updating, a base table through a view. The following example illustrates this.

Execute the following statement to create the SalesEmployees view with a WITH CHECK OPTION clause.

```
CREATE VIEW SalesEmployees AS
   SELECT EmployeeID, GivenName, Surname, DepartmentID
   FROM Employees
   WHERE DepartmentID = 200
   WITH CHECK OPTION;
```

Select to view the contents of this view, as follows:

```
SELECT * FROM SalesEmployees;
```

EmployeeID	GivenName	Surname	DepartmentID
129	Philip	Chin	200
195	Marc	Dill	200
299	Rollin	Overbey	200
467	James	Klobucher	200
...	...	...	...

Next, attempt to update DepartmentID to 400 for Philip Chin:

```
UPDATE SalesEmployees
SET DepartmentID = 400
WHERE EmployeeID = 129;
```

Since the WITH CHECK OPTION was specified, the database server evaluates whether the update violates anything in the view definition (in this case, the expression in the WHERE clause). The statement fails (DepartmentID must be 200), and the database server returns the error, "WITH CHECK OPTION violated for insert/update on base table 'Employees'."

If you had not specified the WITH CHECK OPTION in the view definition, the update operation would proceed, causing the Employees table to be modified with the new value, and subsequently causing Philip Chin to disappear from the view.

If a view (for example, View2) is created that references the SalesEmployees view, any updates or inserts on View2 are rejected that would cause the WITH CHECK OPTION criteria on SalesEmployees to fail, even if View2 is defined without a WITH CHECK OPTION clause.

See also

- "SELECT statement" [*SQL Anywhere Server - SQL Reference*]
- "Summarizing, grouping, and sorting query results" on page 366
- "Working with materialized views" on page 33

Regular view statuses

Regular views have a status associated with them. The status reflects the availability of the view for use by the database server. You can view the status of all views by selecting **Views** in the left pane of Sybase Central, and examining the values in the **Status** column in the right pane. Or, to see the status of a single view, right-click the view in Sybase Central and choose **Properties** to examine the **Status** value.

Following are descriptions of the possible statuses for regular views:

- **VALID** The view is valid and is guaranteed to be consistent with its definition. The database server can make use of this view without any additional work. An enabled view has the status VALID.

In the SYSOBJECT system view, the value 1 indicates a status of VALID. See "SYSOBJECT system view" [*SQL Anywhere Server - SQL Reference*].

- **INVALID** An INVALID status occurs after a schema change to a referenced object where the change results in an unsuccessful attempt to enable the view. For example, suppose a view, v1, references a column, c1, in table t. If you alter t to remove c1, the status of v1 is set to INVALID when the database server tries to recompile the view as part of the ALTER operation that drops the column. In this case, v1 can recompile only after c1 is added back to t, or v1 is changed to no longer refer to c1. Views can also become INVALID if a table or view that they reference is dropped.

  An INVALID view is different from a DISABLED view in that each time an INVALID view is referenced, for example by a query, the database server tries to recompile the view. If the compilation succeeds, the query proceeds. The view's status remains INVALID until it is explicitly enabled. If the compilation fails, an error is returned.

  When the database server internally enables an INVALID view, it issues a performance warning.

  In the SYSOBJECT system view, the value 2 indicates a status of INVALID. See "SYSOBJECT system view" [*SQL Anywhere Server - SQL Reference*].

- **DISABLED** Disabled views are not available for use by the database server for answering queries. Any query that attempts to use a disabled view returns an error.

  A regular view has this state if:

  - you explicitly disable the view, for example by executing an ALTER VIEW ... DISABLE statement.
  - you disable a view (materialized or not) upon which the view depends.
  - you disable view dependencies for a table, for example by executing an ALTER TABLE ... DISABLE VIEW DEPENDENCIES statement.

  For information about enabling and disabling regular views, see "Enable and disable regular views" on page 30.

  In the SYSOBJECT system view, the value 4 indicates a status of DISABLED. See "SYSOBJECT system view" [*SQL Anywhere Server - SQL Reference*].

Create regular views

When you create a regular view, the database server stores the view definition in the database; no data is stored for the view. Instead, the view definition is executed only when it is referenced, and only for the duration of time that the view is in use. This means that creating a view does not require storing duplicate data in the database.

To create a new regular view (Sybase Central)

1. Use the SQL Anywhere 12 plug-in to connect to the database as a user with DBA authority.

2. In the left pane, right-click **Views** and choose **New** » **View**.
3. Follow the instructions in the **Create View Wizard**.
4. In the right pane, click the **SQL** tab to edit the view definition. To save your changes, choose **File** » **Save**.

To create a new regular view (SQL)

1. Connect to a database.
2. Execute a CREATE VIEW statement.

Example

Create a view called DepartmentSize that contains the results of the SELECT statement given earlier in this section:

```
CREATE VIEW DepartmentSize AS
   SELECT DepartmentID, COUNT(*)
   FROM Employees
   GROUP BY DepartmentID;
```

See "CREATE VIEW statement" [*SQL Anywhere Server - SQL Reference*].

Alter regular views

You can alter a regular view using Sybase Central or Interactive SQL.

In Sybase Central, you can alter the definition of views, procedures, and functions on the object's **SQL** tab in the right pane. You edit a view in a separate window by selecting the view and then choosing **File** » **Edit In New Window**. In Interactive SQL, you can use the ALTER VIEW statement to alter a view. The ALTER VIEW statement replaces a view definition with a new definition, but it maintains the permissions on the view.

You cannot rename an existing view. Instead, you must create a new view with the new name, copy the previous definition to it, and then drop the old view.

If you use the ALTER VIEW statement to alter a 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 view with that name owned by you and alters it. If there isn't one, it returns an error.

View alterations and view dependencies

If you want to alter the definition for a regular view, and there are other views dependent on the view, there may be additional steps to make after the alteration is complete. For example, after you alter a view, the database server automatically recompiles it, enabling it for use by the database server. If there are dependent regular views, the database server disables and re-enables them as well. If they cannot be enabled, they are given the status INVALID and you must either make the definition of the regular view consistent with the definitions of the dependent regular views, or vice versa.

To determine whether a regular view has dependent views, use the sa_dependent_views system procedure. See "sa_dependent_views system procedure" [*SQL Anywhere Server - SQL Reference*].

For information about how views are impacted by schema alterations to underlying objects, see "View dependencies" on page 22.

To alter a regular view (Sybase Central)

1. Use the SQL Anywhere 12 plug-in to connect to the database as a user with DBA authority or as the owner of the regular view.
2. In the left pane, double-click **Views**.
3. Select the view.
4. In the right pane, click the **SQL** tab and edit the view's definition.

   Tip
   If you want to edit multiple views, you can open separate windows for each view rather than editing each view on the **SQL** tab in the right pane. You can open a separate window by selecting a view and then choosing **File** » **Edit In New Window**.
5. Choose **File** » **Save**.

To alter a regular view (SQL)

1. Use the SQL Anywhere 12 plug-in to connect to the database as a user with DBA authority or as the owner of the regular view.
2. Execute an ALTER VIEW statement.

Examples

This example shows that when you alter a regular view, you are effectively replacing the definition of the view. In this case, the view definition is being changed to have column names that are more informative.

```
CREATE VIEW DepartmentSize ( col1, col2 ) AS
   SELECT DepartmentID, COUNT( * )
   FROM Employees GROUP BY DepartmentID;
ALTER VIEW DepartmentSize ( DepartmentNumber, NumberOfEmployees ) AS
   SELECT DepartmentID, COUNT( * )
   FROM Employees GROUP BY DepartmentID;
```

The next example shows that when you are changing only an attribute of the regular view, you do not need to redefine the view. In this case, the view is being set to have its definition hidden.

```
ALTER VIEW DepartmentSize SET HIDDEN;
```

See "ALTER VIEW statement" [*SQL Anywhere Server - SQL Reference*].

Drop regular views

You can drop a regular view in both Sybase Central and Interactive SQL.

If you drop a regular view that has dependent views, then the dependent views are made INVALID as part of the drop operation. The dependent views are not usable until they are changed or the original dropped view is recreated. See "Alter regular views" on page 28.

To determine whether a regular view has dependent views, use the sa_dependent_views system procedure. See "sa_dependent_views system procedure" [*SQL Anywhere Server - SQL Reference*].

For information about regular views are impacted by changes to their underlying objects, see "View dependencies" on page 22.

To drop a regular view (Sybase Central)

1. Use the SQL Anywhere 12 plug-in to connect to the database as a user with DBA authority or as the owner of the regular view.
2. In the left pane, double-click **Views**.
3. Right-click the view and choose **Delete**.
4. Click **Yes**.

To drop a regular view (SQL)

1. Connect to the database as a user with DBA authority, or as the owner of the regular view.
2. Execute a DROP VIEW statement.

Examples

Remove a regular view called DepartmentSize.

```
DROP VIEW DepartmentSize;
```

See "DROP VIEW statement" [*SQL Anywhere Server - SQL Reference*].

Enable and disable regular views

The section describes enabling and disabling regular views. For information about enabling and disabling materialized views, see "Enable and disable materialized views" on page 51.

You can control whether a regular view is available for use by the database server by enabling or disabling it. When you disable a regular view, the database server keeps the definition of the view in the database; however, the view is not available for use in satisfying a query. If a query explicitly references a disabled view, the query fails and an error is returned. Once a view is disabled, it must be explicitly re-enabled so that the database server can use it.

If you disable a view, other views that reference it, directly or indirectly, are automatically disabled. So, once you re-enable a view, you must re-enable all other views that were dependent on the view when it was disabled. You can determine the list of dependent views before disabling a view using the sa_dependent_views system procedure. See "sa_dependent_views system procedure" [*SQL Anywhere Server - SQL Reference*].

When you enable a regular view, the database server recompiles it using the definition stored for the view in the database. If compilation is successful, the view status changes to VALID. An unsuccessful recompile could indicate that the schema has changed in one or more of the referenced objects. If so, you must change either the view definition or the referenced objects until they are consistent with each other, and then enable the view.

Note
Before you enable a regular view, you must re-enable and disabled views that it references.

You can grant permissions on disabled objects. Permissions to disabled objects are stored in the database and become effective when the object is enabled.

To disable a regular view (Sybase Central)

1. Connect to the database as a user with DBA authority, or as the owner of the regular view.
2. In the left pane, double-click **Views**.
3. Right-click the view and choose **Disable**.

To disable a regular view (SQL)

1. Connect to the database as a user with DBA authority, or as the owner of the regular view.
2. Execute an ALTER VIEW ... DISABLE statement.

Example

The following example disables a regular view called ViewSalesOrders owned by GROUPO.

```
ALTER VIEW GROUPO.ViewSalesOrders DISABLE;
```

To enable a regular view (Sybase Central)

1. Use the SQL Anywhere 12 plug-in to connect to the database as a user with DBA authority or as the owner of the regular view.
2. In the left pane, double-click **Views**.
3. Right-click the view and choose **Recompile And Enable**.

To enable a regular view (SQL)

1. Connect to the database as a user with DBA authority, or as the owner of the regular view.

2. Execute an ALTER VIEW ... ENABLE statement.

Example

The following example re-enables the regular view called ViewSalesOrders owned by GROUPO.

```
ALTER VIEW GROUPO.ViewSalesOrders ENABLE;
```

See also

- "sa_dependent_views system procedure" [*SQL Anywhere Server - SQL Reference*]
- "ALTER VIEW statement" [*SQL Anywhere Server - SQL Reference*]
- "SYSDEPENDENCY system view" [*SQL Anywhere Server - SQL Reference*]

Browsing data in regular views

To browse the data held within the views, you can use Interactive SQL. Interactive SQL lets you execute queries to identify the data you want to view. See "Querying data" on page 255.

If you are working in Sybase Central, you can select a view on which you have permission and then choose **File** » **View Data In Interactive SQL**. This command opens Interactive SQL with the view contents displayed on the **Results** tab in the **Results** pane. To browse the view, Interactive SQL executes a `SELECT * FROM owner.view` statement.

See also

- "Using Interactive SQL" [*SQL Anywhere Server - Database Administration*]

View system table data

Data in the system tables is only viewable by querying the system views; you cannot query a system table directly. With a few exceptions, each system table has a corresponding view.

The system views are named similar to the system tables, but without an I at the beginning. For example, to view the data in the ISYSTAB system table, you query the SYSTAB system view.

For a list of views provided in SQL Anywhere and a description of the type of information they contain, see "System views" [*SQL Anywhere Server - SQL Reference*].

You can either use Sybase Central or Interactive SQL to browse system view data.

To view data for a system table via a system view (Sybase Central)

1. Use the SQL Anywhere 12 plug-in to connect to the database as a user with DBA authority.
2. In the left pane, double-click **Views**.
3. Select the view corresponding to the system table.

4. In the right pane, click the **Data** tab.

To view data for a system table via a system view (SQL)

1. Connect to the database as a user with DBA authority.
2. Execute a SELECT statement that references the system view corresponding to the system table.

Example

Suppose you want to view the data in the ISYSTAB system table. Since you cannot query the table directly, the following statement displays all data in the corresponding SYSTAB system view:

```
SELECT * FROM SYS.SYSTAB;
```

Sometimes, columns that exist in the system table do not exist in the corresponding system view. To extract a text file containing the definition of a specific view, use a statement such as the following:

```
SELECT viewtext
FROM SYS.SYSVIEWS
WHERE viewname = 'SYSTAB';
OUTPUT TO viewtext.sql
FORMAT TEXT
ESCAPES OFF
QUOTE '';
```

Working with materialized views

A **materialized view** is a view whose result set has been computed and stored on disk, similar to a base table. Conceptually, a materialized view is both a view (it has a query specification stored in the catalog) and a table (it has persistent materialized rows). So, many operations that you perform on tables can be performed on materialized views as well. For example, you can build indexes on, and unload from, materialized views.

Consider using materialized views for frequently executed, expensive queries, such as those involving intensive aggregation and join operations. Materialized views provide a queryable structure in which to store aggregated, joined data. Materialized views are designed to improve performance in environments where the database is large, and where frequent queries result in repetitive aggregation and join operations on large amounts of data. For example, materialized views are ideal for use with data warehousing applications.

Materialized views are precomputed using data from the base tables that they refer to. Materialized views are read-only; no data-altering operations such as INSERT, LOAD, DELETE, and UPDATE can be used on them.

Column statistics are generated and maintained for materialized views in exactly the same manner as for tables. See “Optimizer estimates and column statistics” on page 541.

While you can create indexes on materialized views, you cannot create keys, constraints, triggers, or articles on them.

See also

- “Improving performance with materialized views” on page 555
- “CREATE MATERIALIZED VIEW statement” [*SQL Anywhere Server - SQL Reference*]
- “ALTER MATERIALIZED VIEW statement” [*SQL Anywhere Server - SQL Reference*]
- “REFRESH MATERIALIZED VIEW statement” [*SQL Anywhere Server - SQL Reference*]
- “sa_materialized_view_info system procedure” [*SQL Anywhere Server - SQL Reference*]
- “sa_materialized_view_can_be_immediate system procedure” [*SQL Anywhere Server - SQL Reference*]
- “sa_refresh_materialized_views system procedure” [*SQL Anywhere Server - SQL Reference*]

Manual and immediate materialized views

There are two types of materialized views: manual and immediate, which implies the **refresh type** for the materialized view.

- **Manual views** A manual materialized view, or **manual view**, is a materialized view with a refresh type defined as MANUAL REFRESH. Data in manual views can become stale because manual views are not refreshed until a refresh is explicitly requested, for example by using the REFRESH MATERIALIZED VIEW statement or the sa_refresh_materialized_views system procedure. By default, when you create a materialized view, it is a manual view.

  A manual view is considered stale when any of the underlying tables change, even if the change does not impact data in the materialized view. You can determine whether a manual view is considered stale by examining the DataStatus value returned by the sa_materialized_view_info system procedure. If S is returned, the manual view is stale.

- **Immediate views** An immediate materialized view, or **immediate view**, is a materialized view with a refresh type defined as IMMEDIATE REFRESH. Data in an immediate view is automatically refreshed when changes to the underlying tables affect data in the view. If changes to the underlying tables do not impact data in the view, the view is not refreshed.

  Also, when an immediate view is refreshed, only the rows that need to be changed are acted upon. This is different from refreshing a manual view, where all data is dropped and recreated for a refresh.

You can change a manual view to an immediate view, and vice versa. However, the process for changing from a manual view to an immediate view has more steps. See “Change a manual view to an immediate view” on page 47.

Changing the refresh type for a materialized view can impact the status and properties of the view, especially when you change a manual view to an immediate view. See “Materialized view statuses and properties” on page 36.

How to view materialized view information in the database

You can request information, such as the status of a materialized view, using the sa_materialized_view_info system procedure. See "sa_materialized_view_info system procedure" [*SQL Anywhere Server - SQL Reference*], and "Materialized view statuses and properties" on page 36.

When a materialized view is created, the options in force when the index was created is stored with the view for future use. To retrieve the option settings used during creation of a materialized view, execute the following statement:

```
SELECT b.object_id, b.table_name, a.option_id, c.option_name, a.option_value
FROM SYSMVOPTION a, SYSTAB b, SYSMVOPTIONNAME c
WHERE a.view_object_id=b.object_id
AND b.table_type=2;
```

A table_type of 2 in the SYSTAB view is a materialized view.

To determine the list of views dependent on a materialized view, use the sa_dependent_views system procedure. See "sa_dependent_views system procedure" [*SQL Anywhere Server - SQL Reference*].

Dependency information can also be found in the SYSDEPENDENCY system view. See "SYSDEPENDENCY system view" [*SQL Anywhere Server - SQL Reference*].

See also

- "SYSMVOPTION system view" [*SQL Anywhere Server - SQL Reference*]
- "SYSMVOPTIONNAME system view" [*SQL Anywhere Server - SQL Reference*]
- "SYSTAB system view" [*SQL Anywhere Server - SQL Reference*]

When to use materialized views

You should carefully consider the following requirements and settings before using a materialized view:

- **Disk space requirements** Since materialized views contain a duplicate of data from base tables, you may need to allocate additional space on disk for the database to accommodate the materialized views you create. Careful consideration needs to be given to the additional space requirements so that the benefit derived is balanced against the cost of using materialized views.

- **Maintenance costs and data freshness requirements** The data in materialized views needs to be refreshed when data in the underlying tables changes. The frequency at which a materialized view needs to be refreshed needs to be determined by taking into account potentially conflicting factors such as:

  - **Rate at which underlying data changes** Frequently or large changes to data renders manual views stale. Consider using an immediate view if data freshness is important.

  - **Cost of refreshing** Depending on the complexity of the underlying query for each materialized view, and the amount of data involved, the computation required for refreshing may be very expensive, and frequent refreshing of materialized views may impose an unacceptable workload on the database server. Additionally, materialized views are unavailable for use during the refresh operation.

- **Data freshness requirements of applications** If the database server uses stale materialized view, it presents stale data to applications. Stale data is data that no longer represents the current state of data in the underlying tables. The degree of staleness is governed by the frequency at which the materialized view is refreshed. An application must be designed to determine the degree of staleness it can tolerate to achieve improved performance. For more information about managing data staleness in materialized views, see "Setting the optimizer staleness threshold for materialized views" on page 54.

- **Data consistency requirements** When refreshing materialized views, you must determine the consistency with which the materialized views should be refreshed. See the WITH ISOLATION LEVEL clause of the "REFRESH MATERIALIZED VIEW statement" [*SQL Anywhere Server - SQL Reference*].

- **Use in optimization** You should verify that the optimizer considers the materialized views when executing a query. You can see the list of materialized views used for a particular query by looking at the **Advanced Details** window of the query's graphical plan in Interactive SQL. See "Reading execution plans" on page 592, and "Improving performance with materialized views" on page 555.

  You can also use Application Profiling mode in Sybase Central to determine whether a materialized view was considered during the enumeration phase of a query by looking at the access plans enumerated by the optimizer. Tracing must be turned on, and must be configured to include the OPTIMIZATION_LOGGING tracing type, to see the access plans enumerated by the optimizer. See "Application profiling" on page 158, and "Choosing a diagnostic tracing level" on page 171.

See also

- "Manual and immediate materialized views" on page 34
- "Improving performance with materialized views" on page 555

Materialized view statuses and properties

Materialized views are characterized by a combination of their status and properties. The **status** of a materialized view reflects the availability of the view for use by the database server. The **properties** of a materialized view reflect the state of the data within the view.

The best way to determine the status and properties of existing materialized views is to use the sa_materialized_view_info system procedure. See "sa_materialized_view_info system procedure" [*SQL Anywhere Server - SQL Reference*].

You can also view information about materialized views by choosing the **Views** folder in Sybase Central and examining the details provided for the individual views, or by querying the SYSTAB and SYSVIEW system views. See "SYSTAB system view" [*SQL Anywhere Server - SQL Reference*], and "SYSVIEW system view" [*SQL Anywhere Server - SQL Reference*].

Materialized view statuses

There are two possible statuses for materialized views:

- **Enabled** A materialized view has the status **enabled** if it has been successfully compiled and it is available for use by the database server. An enabled materialized view may not have data in it. For example, if you truncate the data from an enabled materialized view, it changes to enabled and uninitialized. A materialized view can be initialized but empty if there is no data in the underlying tables that satisfies the definition for the materialized view. This is not the same as a materialized view that has no data in it because it is not initialized.

- **Disabled** A materialized view has the status **disabled** only if you explicitly disable it, for example by using the ALTER MATERIALIZED VIEW ... DISABLE statement. When you disable a materialized view, the data and indexes for the view are dropped. Also, when you disable an immediate view, it is changed to a manual view.

To determine whether a view is enabled or disabled, use the sa_materialized_view_info system procedure to return the Status property for the view. See "sa_materialized_view_info system procedure" [*SQL Anywhere Server - SQL Reference*].

For information about enabling and disabling materialized views, see "Enable and disable materialized views" on page 51.

Materialized view properties

Materialized view properties are used by the optimizer when evaluating whether to use a view. The following list describes the properties for a materialized view that are returned by the sa_materialized_view_info system procedure:

- **Status** The Status property indicates whether the view is enabled or disabled.

- **DataStatus** The DataStatus property reflects the state of the data in the view. For example, it tells you whether the view is initialized and whether the view is stale. Manual views are stale if data in the underlying tables has changed since the last time the materialized view was refreshed. Immediate views are never stale.

- **ViewLastRefreshed** The ViewLastRefreshed property indicates the last time the view was refreshed.

- **DateLastModified** The DateLastModified property indicates the most recent time the data in any underlying table was modified if the view is stale.

- **AvailForOptimization** The AvailForOptimization property reflects whether the view is available for use by the optimizer.

- **RefreshType** The RefreshType property indicates whether it is a manual view or an immediate view.

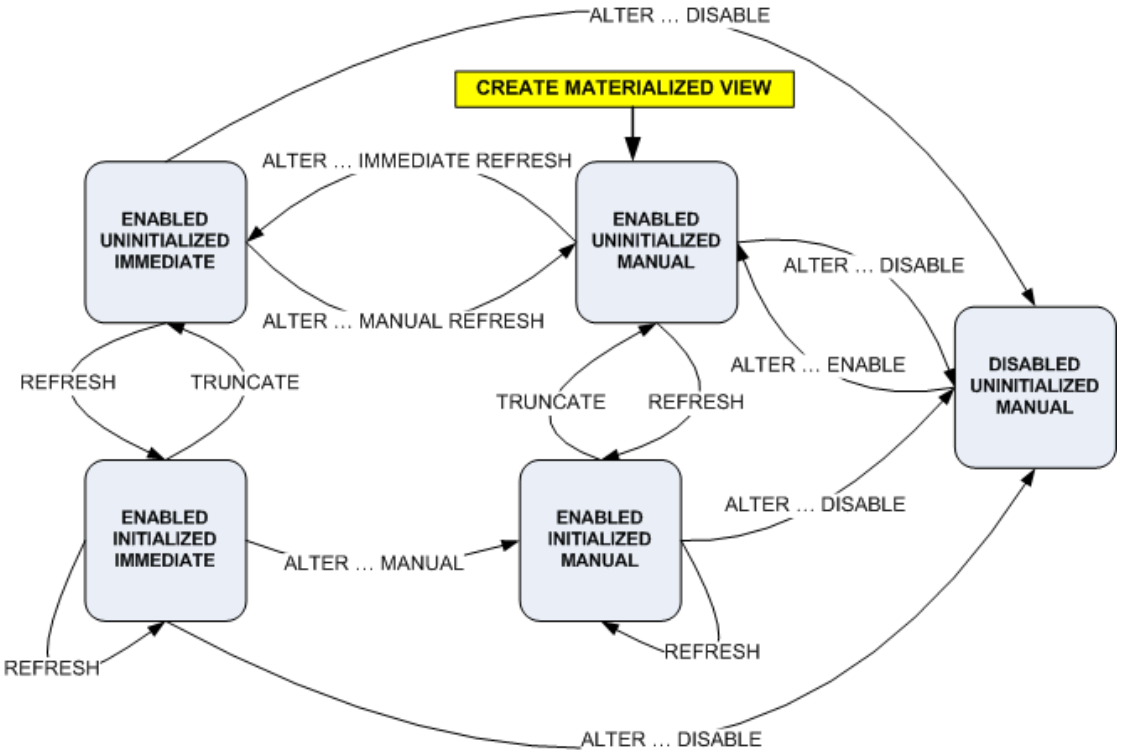
For the list of possible values for each property, see "sa_materialized_view_info system procedure" [*SQL Anywhere Server - SQL Reference*].

While there is no property that tells you whether a manual view can be converted to an immediate view, you can determine this by using the sa_materialized_view_can_be_immediate system procedure. See "sa_materialized_view_can_be_immediate system procedure" [*SQL Anywhere Server - SQL Reference*].

Status and property changes when altering, refreshing, and truncating a materialized view

Operations you perform on a materialized view such as altering, refreshing, and truncating, impact the view's status and properties. The following diagram shows how these tasks impact the status and some of the properties of a materialized view.

In the diagram, each gray square is a materialized view; immediate views are identified by the term IMMEDIATE, and manual views by the term MANUAL. The term ALTER in the connectors between grey boxes is short for ALTER MATERIALIZED VIEW. Although SQL statements are shown for changing the materialized view status, you can also use Sybase Central to perform these activities.



Some important concepts to note from the diagram are as follows:

- When you create a materialized view, it is an enabled manual view and it is uninitialized (contains no data).
- When you refresh an uninitialized view, it becomes initialized (populated with data).
- Changing from a manual view to an immediate view requires several steps, and there are additional restrictions for immediate views. See “Change a manual view to an immediate view” on page 47, and “Additional restrictions for immediate views” on page 40.
- When you disable a materialized view:
  - the data is dropped

- the view reverts to uninitialized
- the indexes are dropped
- an immediate view reverts to manual

See also

- "Working with materialized views" on page 33
- "CREATE MATERIALIZED VIEW statement" [*SQL Anywhere Server - SQL Reference*]
- "ALTER MATERIALIZED VIEW statement" [*SQL Anywhere Server - SQL Reference*]
- "REFRESH MATERIALIZED VIEW statement" [*SQL Anywhere Server - SQL Reference*]
- "TRUNCATE statement" [*SQL Anywhere Server - SQL Reference*]
- "DROP MATERIALIZED VIEW statement" [*SQL Anywhere Server - SQL Reference*]
- "Restrictions on materialized views" on page 39
- "Manual and immediate materialized views" on page 34
- "SYSOBJECT system view" [*SQL Anywhere Server - SQL Reference*]

Restrictions on materialized views

The following restrictions apply when creating, initializing, refreshing, and view matching materialized views:

- When creating a materialized view, the definition for the materialized view must define column names explicitly; you cannot include a `SELECT *` construct as part of the column definition.
- Do not include columns defined as TIMESTAMP WITH TIME ZONE in the materialized view. The value of the time_zone_adjustment option varies between connections based on their location and the time of year, resulting in incorrect results and unexpected behavior.
- When creating a materialized view, the definition for the materialized view cannot contain:
  - references to other views, materialized or not
  - references to remote or temporary tables
  - variables such as CURRENT USER; all expressions must be deterministic
  - calls to stored procedures, user-defined functions, or external functions
  - Transact-SQL outer joins
  - FOR XML clauses

- The following database options must have the specified settings when a materialized view is created; otherwise, an error is returned. These database option values are also required for the view to be used by the optimizer:
  - ansinull=On
  - conversion_error=On
  - divide_by_zero_error=On
  - sort_collation=Internal
  - string_rtruncation=On
- The following database option settings are stored for each materialized view when it is created. The current option values for the connection must match the values stored for a materialized view in order for the view to be used in optimization:
  - date_format
  - date_order
  - default_timestamp_increment
  - first_day_of_week
  - nearest_century
  - precision
  - scale
  - time_format
  - timestamp_format
  - timestamp_with_time_zone_format
  - default_timestamp_increment
  - uuid_has_hyphens
- When a view is refreshed, the connection settings for all the options listed in the bullets above are ignored. Instead, the database option settings (which must match the stored settings for the view) are used.

Specifying an ORDER BY clause in a materialized view definition

Materialized views are similar to base tables in that the rows are not stored in any particular order; the database server orders the rows in the most efficient manner when computing the data. Therefore, specifying an ORDER BY clause in a materialized view definition has no impact on the ordering of rows when the view is materialized. Also, the ORDER BY clause in the view's definition is ignored by the optimizer when performing view matching.

For information about materialized views and view matching by the optimizer, see “Improving performance with materialized views” on page 555.

Additional restrictions for immediate views

The following restrictions are checked when changing a manual view to an immediate view. An error is returned if the view violates any of the restrictions:

Note
You can use the sa_materialized_view_can_be_immediate system procedure to find out if a manual view is eligible to become an immediate view. See "sa_materialized_view_can_be_immediate system procedure" [*SQL Anywhere Server - SQL Reference*].

- The view must be uninitialized. See "Materialized view statuses and properties" on page 36.
- If the view does not contain outer joins, then the view must have a unique index on non nullable columns. If the view contains outer joins, the view must have a unique index on non nullable columns, or a unique index declared as WITH NULLS NOT DISTINCT on nullable columns. See "Create indexes" on page 60.
- If the view definition is a grouped query, the unique index columns must correspond to select list items that are not aggregate functions.
- The view definition cannot contain:
  - GROUPING SETS clauses
  - CUBE clauses
  - ROLLUP clauses
  - DISTINCT clauses
  - row limit clauses
  - non-deterministic expressions
  - self and recursive joins
  - LATERAL, CROSS APPLY, or APPLY clauses
- The view definition must be a single select-project-join or grouped-select-project-join query block, and the grouped-select-project-join query block cannot contain a HAVING clause.
- The grouped-select-project-join query block must contain COUNT(*) in the select list, and is allowed only the SUM and COUNT aggregate functions.

  For a description of these structures, see "Materialized view evaluation" on page 559.
- An aggregate function in the select list cannot be referenced in a complex expression. For example, `SUM(expression) + 1` is not allowed in the select list.
- If the select list contains the `SUM(`*`expression`*`)` aggregate function and *expression* is a nullable expression, then the select list must include a `COUNT(`*`expression`*`)` aggregate function.
- If the view definition contains outer joins (LEFT OUTER JOIN, RIGHT OUTER JOIN, FULL OUTER JOIN) then the view definition must satisfy the following extra conditions:

  1. If a table, T, is referenced in an ON condition of an OUTER JOIN as a preserved side, then T must have a primary key and the primary key columns must be present in the select list of the view. For example, the immediate materialized view V defined as `SELECT T1.pk, R1.X FROM T1, T2 LEFT OUTER JOIN (R1 KEY JOIN R2) ON T1.Y = R.Y` has the preserved table, T1, referenced in the ON clause and its primary key column, T1.pk, is in the select list of the immediate materialized view, V.

2. For each NULL-supplying side of an outer join, there must be at least one base table such that one of its non-nullable columns is present in the select list of the immediate materialized view. For example, for the immediate materialized view, V, defined as `SELECT T1.pk, R1.X FROM T1, T2 LEFT OUTER JOIN (R1 KEY JOIN R2) ON T1.Y = R1.Y`, the NULL-supplying side of the left outer join is the table expression `(R1 KEY JOIN R2)`. The column R1.X is in the select list of the V and R1.X is a non nullable column of the table R1.

3. If the view is a grouped view and the previous condition does not hold, then for each NULL-supplying side of an outer join, there must be at least one base table, T, such that one of its non-nullable columns, T.C, is used in the aggregate function `COUNT(T.C )` in the select list of the immediate materialized view. For example, for the immediate materialized view, V, is defined as `SELECT T1.pk, COUNT(R1.X) FROM T1, T2 LEFT OUTER JOIN (R1 KEY JOIN R2) ON T1.Y = R1.Y GROUP BY T1.pk`, the NULL-supplying side of the left outer join is the table expression `(R1 KEY JOIN R2)`. The aggregate function `COUNT( R1.X )` is in the select list of the V and R1.X is a non-nullable column of the table R1.

4. The following conditions must be satisfied by the predicates of the views with outer joins:

   - The ON clause predicates for LEFT, RIGHT, and FULL OUTER JOINs must refer to both preserved and NULL-supplying table expression. For example, `T LEFT OUTER JOIN R ON R.X = 1` does not satisfy this condition as the predicate R.X=1 references only the NULL-supplying side R.

   - Any predicate must reject NULL-supplied rows produced by a nested outer join. In other words, if a predicate refers to a table expression which is NULL-supplied by a nested outer join, then it must reject all rows which have nulls generated by that outer join.

     For example, the view V1 `SELECT T1.pk, R1.X FROM T1, T2 LEFT OUTER JOIN (R1 KEY JOIN R2) ON (T1.Y = R1.Y) WHERE R1.Z = 10` has the predicate `R1.Z=10` referencing the table R1 which can be NULL-supplied by the `T2 LEFT OUTER JOIN (R1 KEY JOIN R2)`, hence it must reject any NULL-supplied rows. This is true because the predicate evaluates to UNKNOWN when the column R1.Z is NULL.

     However, the view V2 `SELECT T1.pk, R1.X FROM T1, T2 LEFT OUTER JOIN (R1 KEY JOIN R2) ON (T1.Y = R1.Y) WHERE R1.Z IS NULL` does not have this property. The predicate `R1.Z IS NULL` references the NULL-supplying side R1 but it evaluates to TRUE when the table R1 is NULL-supplied (i.e., the R1.Z column is null). Note that the property of rejecting NULL-supplied rows is not as restrictive as a NULL-intolerant property. For example, the predicate `R.X IS NOT DISTINCT FROM T.X and rowid(T) IS NOT NULL` is not NULL-intolerant on the table T as it evaluates to TRUE when T.X is NULL. However, the predicate rejects all the rows which are NULL-supplied on the base table T.

See also

- "Create materialized views" on page 43
- "TimeZoneAdjustment connection property" [*SQL Anywhere Server - Database Administration*]
- "CREATE MATERIALIZED VIEW statement" [*SQL Anywhere Server - SQL Reference*]
- "REFRESH MATERIALIZED VIEW statement" [*SQL Anywhere Server - SQL Reference*]
- For more information about NULL-supplying and preserved sides in outer joins, see "Outer joins" on page 399.

Create materialized views

When you create a materialized view, its definition is stored in the database. The database server validates the definition to make sure it compiles properly. All column and table references are fully qualified by the database server to ensure that all users with access to the view see an identical definition. After successfully creating a materialized view, you populate it with data, also known as **initializing** the view, using a REFRESH MATERIALIZED VIEW statement. See "REFRESH MATERIALIZED VIEW statement" [*SQL Anywhere Server - SQL Reference*].

Before creating, initializing, or refreshing materialized views, ensure that all materialized view restrictions have been met. See "Restrictions on materialized views" on page 39.

To obtain a list of all materialized views in the database, including their status, use the sa_materialized_view_info system procedure. See "sa_materialized_view_info system procedure" [*SQL Anywhere Server - SQL Reference*].

After you finish creating the definition for the materialized view, it appears in the **Views** folder in Sybase Central.

To create a materialized view (Sybase Central)

1. Use the SQL Anywhere 12 plug-in to connect to the database as a user with DBA or RESOURCE authority.
2. In the left pane, right-click **Views** and choose **New** » **Materialized View**.
3. Follow the instructions in the **Create Materialized View Wizard**.
4. Initialize the materialized view so that it contains data. See "Initialize materialized views" on page 44.

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" on page 55.

To create a materialized view (SQL)

1. Connect to the database as a user with DBA or RESOURCE authority.
2. Execute a CREATE MATERIALIZED VIEW statement. The database server creates and stores the view definition in the database, and sets the view's status to ENABLED. See "CREATE MATERIALIZED VIEW statement" [*SQL Anywhere Server - SQL Reference*].
3. You must initialize the materialized view so that it contains data. See "Initialize materialized views" on page 44.

See also

- "CREATE MATERIALIZED VIEW statement" [*SQL Anywhere Server - SQL Reference*]
- "REFRESH MATERIALIZED VIEW statement" [*SQL Anywhere Server - SQL Reference*]
- "SQL Anywhere sample database" [*SQL Anywhere 12 - Introduction*]
- "Restrictions on materialized views" on page 39

Example

The following statement creates a materialized view, EmployeeConfid16, containing information about employees, and then initializes it to populate it with data.

```
CREATE MATERIALIZED VIEW EmployeeConfid16 AS
   SELECT EmployeeID, Employees.DepartmentID, SocialSecurityNumber, Salary,
ManagerID,
      Departments.DepartmentName, Departments.DepartmentHeadID
   FROM Employees, Departments
   WHERE Employees.DepartmentID=Departments.DepartmentID;
REFRESH MATERIALIZED VIEW EmployeeConfid16;
```

Caution
u 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" on page 55.

Initialize materialized views

You must initialize a materialized view to make it available for use by the database server. To initialize it, you refresh it. A failed refresh attempt returns the materialized view to an uninitialized state.

Before creating, initializing, or refreshing materialized views, ensure that all materialized view restrictions have been met. See "Restrictions on materialized views" on page 39.

Note
You can also initialize all uninitialized materialized views at once using the sa_refresh_materialized_views system procedure. See "sa_refresh_materialized_views system procedure" [*SQL Anywhere Server - SQL Reference*].

To initialize a materialized view (Sybase Central)

1. Use the SQL Anywhere 12 plug-in to connect to the database as a user with DBA authority or as a user with INSERT permission on the materialized view.
2. In the left pane, double-click **Views**.
3. Right-click a materialized view and choose **Refresh Data**.
4. Select an isolation level and click **OK**.

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" on page 55.

To initialize a materialized view (SQL)

1. Connect to the database as a user with DBA authority, or as a user with INSERT permission on the materialized view.
2. Execute a REFRESH MATERIALIZED VIEW statement.

Example

The following statements create a materialized view, EmployeeConfid6, and then initializes it:

```
CREATE MATERIALIZED VIEW EmployeeConfid6 AS
   SELECT EmployeeID, Employees.DepartmentID, SocialSecurityNumber, Salary,
ManagerID,
      Departments.DepartmentName, Departments.DepartmentHeadID
   FROM Employees, Departments
   WHERE Employees.DepartmentID=Departments.DepartmentID;
REFRESH MATERIALIZED VIEW EmployeeConfid6;
```

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" on page 55.

See also

- "CREATE MATERIALIZED VIEW statement" [*SQL Anywhere Server - SQL Reference*]
- "REFRESH MATERIALIZED VIEW statement" [*SQL Anywhere Server - SQL Reference*]
- "Enable and disable materialized views" on page 51

Refresh manual views

Manual views become stale when changes occur to their underlying base tables. Refreshing a manual view means that the database server re-executes the query definition for the view and replaces the view data with the new result set of the query. Refreshing makes the view data consistent with the underlying data. You should consider the acceptable degree of data staleness for the manual view and devise a refresh strategy. Your strategy should allow for the time it takes to complete a refresh, since the view is not available for querying during the refresh operation.

You can also set up a strategy in which the view is refreshed using events. For example, you can create an event to refresh at some regular interval.

Immediate views do not need to be refreshed except if they are uninitialized (contain no data), for example after being truncated.

You can also use the sa_refresh_materialized_views system procedure to refresh views. See "sa_refresh_materialized_views system procedure" [*SQL Anywhere Server - SQL Reference*].

You can configure a staleness threshold beyond which the optimizer should not use a materialized view when processing queries, using the materialized_view_optimization database option. See "Setting the optimizer staleness threshold for materialized views" on page 54.

When using the REFRESH MATERIALIZED VIEW statement, you can override the connection isolation level using the WITH ISOLATION LEVEL clause. For more information on how to control concurrency when refreshing a materialized view, see the WITH clause of the "REFRESH MATERIALIZED VIEW statement" [*SQL Anywhere Server - SQL Reference*].

Upgrading databases with materialized views
It is recommended that you refresh materialized views after upgrading your database server, or after rebuilding or upgrading your database to work with an upgraded database server.

To refresh a manual view (Sybase Central)

1. Use the SQL Anywhere 12 plug-in to connect to the database as a user with DBA authority or as a user with INSERT permission on the materialized view. You must also have SELECT permissions on the underlying tables.
2. In the left pane, double-click **Views**.
3. Right-click a materialized view and choose **Refresh Data**.
4. Select an isolation level and click **OK**.

To refresh a manual view (SQL)

1. Connect to the database as a user with DBA authority, or as a user with INSERT permission on the materialized view. You must also have SELECT permissions on the underlying tables.
2. Execute a REFRESH MATERIALIZED VIEW statement.

Example

The following statement creates and then refreshes the EmployeeConfid33 materialized view.


```
CREATE MATERIALIZED VIEW EmployeeConfid33 AS
   SELECT EmployeeID, Employees.DepartmentID, SocialSecurityNumber, Salary,
ManagerID,
      Departments.DepartmentName, Departments.DepartmentHeadID
   FROM Employees, Departments
   WHERE Employees.DepartmentID=Departments.DepartmentID;
REFRESH MATERIALIZED VIEW EmployeeConfid33;
```

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" on page 55.

See also

- "Change a manual view to an immediate view" on page 47
- "Automating tasks using schedules and events" [*SQL Anywhere Server - Database Administration*]
- "materialized_view_optimization option" [*SQL Anywhere Server - Database Administration*]

Change a manual view to an immediate view

When you create a materialized view its refresh type is manual. However, you can change it to immediate. To change from manual to immediate, the view must be in an uninitialized state (contain no data). If the view was just created and has not yet been refreshed, it is uninitialized. If it has data in it, you must truncate the data. The view must also have a unique index, and must conform to the restrictions required for an immediate view. See "Additional restrictions for immediate views" on page 40.

An immediate view can be converted to manual at any time by simply changing its refresh type.

The following procedures explain how to change a manual view to an immediate view. Before performing one of these procedures, verify the manual view has a unique index and is uninitialized. Then, optionally, check its eligibility for immediate refresh type using the sa_materialized_view_can_be_immediate system procedure. See "sa_materialized_view_can_be_immediate system procedure" [*SQL Anywhere Server - SQL Reference*].

To change a manual view to an immediate view (Sybase Central)

1. Use the SQL Anywhere 12 plug-in to connect to the database as a user with DBA authority or as the owner of the view and all of the tables it references.
2. In the left pane, double-click **Views**.
3. Right-click the materialized view and choose **Properties**.
4. In the **Refresh Type** field, choose **Immediate**.
5. Click **OK**.

To change a manual view to an immediate view (SQL)

1. Connect to the database as a user with DBA authority, or as owner of the view and all the tables it references.
2. Change the refresh type to immediate by executing an ALTER MATERIALIZED VIEW ... IMMEDIATE REFRESH statement.

The following procedures explain how to change an immediate view to a manual view.

To change an immediate view to a manual view (Sybase Central)

1. Use the SQL Anywhere 12 plug-in to connect to the database as a user with DBA authority or as the owner of the view.
2. In the left pane, double-click **Views**.
3. Right-click a materialized view and choose **Properties**.
4. In the **Refresh Type** field, choose **Manual**.
5. Click **OK**.

To change an immediate view to a manual view (SQL)

1. Connect to the database as the owner of the view, or as a user with DBA authority.
2. Change the refresh type to manual by executing an ALTER MATERIALIZED VIEW ... MANUAL REFRESH statement.

Example

The following example creates a materialized view and then initializes it. A unique index is then added since immediate views must have a unique index. Since the view must not have data in it when the refresh type is changed, the view is truncated. Finally, the refresh type is changed.

```
CREATE MATERIALIZED VIEW EmployeeConfid44 AS
   SELECT EmployeeID, Employees.DepartmentID,
          SocialSecurityNumber, Salary, ManagerID,
          Departments.DepartmentName, Departments.DepartmentHeadID
     FROM Employees, Departments
     WHERE Employees.DepartmentID=Departments.DepartmentID;
REFRESH MATERIALIZED VIEW EmployeeConfid44;
CREATE UNIQUE INDEX EmployeeIDIdx
   ON EmployeeConfid44 ( EmployeeID );
TRUNCATE MATERIALIZED VIEW EmployeeConfid44;
ALTER MATERIALIZED VIEW EmployeeConfid44
   IMMEDIATE REFRESH;
```

The following statement changes the refresh type back to manual:

```
ALTER MATERIALIZED VIEW EmployeeConfid44
   MANUAL REFRESH;
```


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" on page 55.

See also

- "ALTER MATERIALIZED VIEW statement" [*SQL Anywhere Server - SQL Reference*]
- "Materialized view statuses and properties" on page 36
- "Create indexes" on page 60
- "Initialize materialized views" on page 44

Encrypt and decrypt materialized views

Materialized views can be encrypted for additional security. For example, if a materialized view contains data that was encrypted in the underlying table, you may want to encrypt the materialized view as well. Table encryption must already be enabled in the database to encrypt a materialized view. The encryption algorithm and key specified at database creation are used to encrypt the materialized view. To see the encryption settings in effect for your database, including whether table encryption is enabled, query the Encryption database property using the DB_PROPERTY function, as follows:

```
SELECT DB_PROPERTY( 'Encryption' );
```

As with table encryption, encrypting a materialized view can impact performance since the database server must decrypt data it retrieves from the view.

To encrypt a materialized view (Sybase Central)

1. Use the SQL Anywhere 12 plug-in to connect to the database as a user with DBA authority. as the owner of the view, or as a user with DBA authority.
2. In the left pane, double-click **Views**.
3. Right-click the materialized view and choose **Properties**.
4. Click the **Miscellaneous** tab.
5. Select the **Materialized View Data Is Encrypted** checkbox.
6. Click **OK**.

To encrypt a materialized view (SQL)

1. Connect to the database as a user with DBA authority, or as owner of the materialized view.
2. Execute an ALTER MATERIALIZED VIEW statement using the ENCRYPTED clause.

To decrypt a materialized view (Sybase Central)

1. Use the SQL Anywhere 12 plug-in to connect to the database as a user with DBA authority or as the owner of the view.
2. In the left pane, double-click **Views**.
3. Right-click the materialized view and choose **Properties**.
4. Click the **Miscellaneous** tab.
5. Clear the **Materialized View Data Is Encrypted** checkbox.
6. Click **OK**.

To decrypt a materialized view (SQL)

1. Connect to the database as a user with DBA authority, or as owner of the materialized view.
2. Execute an ALTER MATERIALIZED VIEW statement using the NOT ENCRYPTED clause.

Example of encrypting a materialized view

The following statement creates, initializes, and then encrypts the EmployeeConfid44 materialized view. The database must already be configured to allow encrypted tables for this statement to work:

```
CREATE MATERIALIZED VIEW EmployeeConfid44 AS
   SELECT EmployeeID, Employees.DepartmentID, SocialSecurityNumber, Salary,
ManagerID,
      Departments.DepartmentName, Departments.DepartmentHeadID
   FROM Employees, Departments
   WHERE Employees.DepartmentID=Departments.DepartmentID;
REFRESH MATERIALIZED VIEW EmployeeConfid44;
ALTER MATERIALIZED VIEW GROUPO.EmployeeConfid44 ENCRYPTED;
```

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" on page 55.

Example of decrypting a materialized view

The following statement decrypts the EmployeeConfid44 materialized view:

```
ALTER MATERIALIZED VIEW GROUPO.EmployeeConfid44 NOT ENCRYPTED;
```

See also

- "Enabling table encryption in the database" [*SQL Anywhere Server - Database Administration*]
- "ALTER MATERIALIZED VIEW statement" [*SQL Anywhere Server - SQL Reference*]
- "DB_PROPERTY function [System]" [*SQL Anywhere Server - SQL Reference*]

Enable and disable materialized views

You can control whether a materialized view is available for use by the database server by enabling or disabling it. A disabled materialized view is also not considered by the optimizer during optimization. If a query explicitly references a disabled materialized view, the query fails and an error is returned. When you disable a materialized view, the database server drops the data for the view, but keeps the definition in the database. When you re-enable a materialized view, it is in an uninitialized state and you must refresh it to populate it with data.

Regular views that are dependent on a materialized view are automatically disabled by the database server if the materialized view is disabled. As a result, once you re-enable a materialized view, you must re-enable all dependent views. For this reason, you may want to determine the list of views dependent on the materialized view before disabling it. You can do this using the sa_dependent_views system procedure. This procedure examines the ISYSDEPENDENCY system table and returns the list of dependent views, if any.

When you disable a materialized view, the data and indexes are dropped, and if the view was an immediate view, it is changed to a manual view. So, when you re-enable it, you must refresh it, rebuild the indexes, and change it back to immediate view (if necessary).

You can grant permissions on disabled objects. Permissions to disabled objects are stored in the database and become effective when the object is enabled.

To disable a materialized view (Sybase Central)

1. Use the SQL Anywhere 12 plug-in to connect to the database as a user with DBA authority or as the owner of the materialized view.
2. In the left pane, double-click **Views**.
3. Right-click the materialized view and choose **Disable**.

To disable a materialized view (SQL)

1. Connect to the database as a user with DBA authority, or as owner of the materialized view.
2. Execute an ALTER MATERIALIZED VIEW ... DISABLE statement.

To enable a materialized view (Sybase Central)

1. Use the SQL Anywhere 12 plug-in to connect to the database as a user with DBA authority or as the owner of the materialized view.
2. In the left pane, double-click **Views**.
3. Right-click the materialized view and choose **Recompile And Enable**.
4. Optionally, right-click the view and choose **Refresh Data** to initialize the view and populate it with data.

To enable a materialized view (SQL)

1. Connect to the database as a user with DBA authority, or as owner of the materialized view.
2. Execute an ALTER MATERIALIZED VIEW ... ENABLE statement.
3. Optionally, execute a REFRESH MATERIALIZED VIEW to initialize the view and populate it with data.

Example of disabling a materialized view

The following example creates the EmployeeConfid55 materialized view, initializes it, and then disables it. When it is disabled, the data for the materialized view is dropped, the definition for the materialized view remains in the database, the materialized view is unavailable for use by the database server, and dependent views, if any, are disabled.

```
CREATE MATERIALIZED VIEW EmployeeConfid55 AS
   SELECT EmployeeID, Employees.DepartmentID, SocialSecurityNumber, Salary,
ManagerID,
      Departments.DepartmentName, Departments.DepartmentHeadID
   FROM Employees, Departments
   WHERE Employees.DepartmentID=Departments.DepartmentID;
REFRESH MATERIALIZED VIEW EmployeeConfid55;
ALTER MATERIALIZED VIEW EmployeeConfid55 DISABLE;
```

Example of enabling a materialized view

The following two statements, respectively, re-enable the EmployeeConfid55 materialized view, and then populate it with data.

```
ALTER MATERIALIZED VIEW EmployeeConfid55 ENABLE;
REFRESH MATERIALIZED VIEW EmployeeConfid55;
```

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” on page 55.

See also

- “Change a manual view to an immediate view” on page 47
- “REFRESH MATERIALIZED VIEW statement” [*SQL Anywhere Server - SQL Reference*]
- “CREATE INDEX statement” [*SQL Anywhere Server - SQL Reference*]
- “sa_dependent_views system procedure” [*SQL Anywhere Server - SQL Reference*]
- “ALTER MATERIALIZED VIEW statement” [*SQL Anywhere Server - SQL Reference*]
- “ALTER VIEW statement” [*SQL Anywhere Server - SQL Reference*]
- “View dependencies” on page 22
- “SYSDEPENDENCY system view” [*SQL Anywhere Server - SQL Reference*]

Enable and disable optimizer use of a materialized view

The optimizer maintains a list of materialized views that can be used in the optimization process. A materialized view is not considered a candidate for use in optimization if its definition includes certain elements that the optimizer rejects, or if its data is not considered fresh enough to use. For information about what qualifies a materialized view as a candidate in the optimization process, see "Improving performance with materialized views" on page 555.

By default, materialized views are available for use by the optimizer. However, you can disable optimizer's use of a materialized view unless it is explicitly referenced in a query.

To determine if a materialized view is enabled or disabled for use by the optimizer, use the sa_materialized_view_info system procedure. See "sa_materialized_view_info system procedure" [*SQL Anywhere Server - SQL Reference*].

To enable a materialized view's use in optimization (Sybase Central)

1. Use the SQL Anywhere 12 plug-in to connect to the database as a user with DBA authority.
2. In the left pane, double-click **Views**.
3. Right-click the materialized view and choose **Properties**.
4. Click the **General** tab and select **Used In Optimization**.
5. Click **OK**.

To enable a materialized view's use in optimization (SQL)

1. Connect to the database as a user with DBA authority, or as owner of the materialized view.
2. Execute an ALTER MATERIALIZED VIEW statement with the ENABLE USE IN OPTIMIZATION clause.

Examples

The following statement enables the EmployeeConfid77 view for use in optimization:

```
ALTER MATERIALIZED VIEW GROUPO.EmployeeConfid77 ENABLE USE IN OPTIMIZATION;
```

To disable a materialized view's use in optimization (Sybase Central)

1. Use the SQL Anywhere 12 plug-in to connect to the database as a user with DBA authority.
2. In the left pane, double-click **Views**.
3. Right-click the materialized view and choose **Properties**.
4. Click the **General** tab and clear **Used In Optimization**.
5. Click **OK**.

To disable a materialized view's use in optimization (SQL)

1. Connect to the database as a user with DBA authority, or as owner of the materialized view.
2. Execute an ALTER MATERIALIZED VIEW statement with the DISABLE USE IN OPTIMIZATION clause.

Example

The following statement creates the EmployeeConfid77 materialized view, refreshes it, and then disables it for use in optimization.

```
CREATE MATERIALIZED VIEW EmployeeConfid77 AS
   SELECT EmployeeID, Employees.DepartmentID, SocialSecurityNumber, Salary,
ManagerID,
      Departments.DepartmentName, Departments.DepartmentHeadID
   FROM Employees, Departments
   WHERE Employees.DepartmentID=Departments.DepartmentID;
REFRESH MATERIALIZED VIEW EmployeeConfid77;
ALTER MATERIALIZED VIEW EmployeeConfid77 DISABLE USE IN OPTIMIZATION;
```

See also

- "ALTER MATERIALIZED VIEW statement" [*SQL Anywhere Server - SQL Reference*]

Setting the optimizer staleness threshold for materialized views

Data in a materialized view becomes stale when the data changes in the tables referenced by the materialized view. The materialized_view_optimization database option allows you to configure a staleness threshold beyond which the optimizer should no longer consider using it when processing queries. The materialized_view_optimization database option does not impact how often materialized views are refreshed.

If a query explicitly references a materialized view, the view is used to process the query, regardless of freshness of the data in the view. As well, the OPTION clause of a SELECT statement can be used to override the setting of the materialized_view_optimization database option, forcing the use of the materialized view. See "SELECT statement" [*SQL Anywhere Server - SQL Reference*].

If you notice that the materialized view is not considered by the optimizer, it may be due to staleness. Adjust the interval specified for the event or trigger responsible for refreshing the view.

Note
When snapshot isolation is in use, the optimizer avoids using a materialized view if it was refreshed after the start of the snapshot for a transaction.

For information about how to use the materialized_view_optimization database option, see "materialized_view_optimization option" [*SQL Anywhere Server - Database Administration*].

For information about using events and triggers, see "Automating tasks using schedules and events" [*SQL Anywhere Server - Database Administration*].

For information about determining whether the materialized view has been considered by optimizer, see "Reading execution plans" on page 592 and "Monitor query performance" on page 211.

Hide materialized views

You can hide a materialized view's definition from users. When you hide a materialized view, you obfuscate the view definition stored in the database, making the view invisible in the catalog. The view can still be directly referenced, and is still eligible for use during query processing. When a materialized view is hidden, debugging using the debugger will not show the view definition, nor will the definition be available through procedure profiling, and the view can be still unloaded and reloaded into other databases.

Hiding a materialized view is irreversible, and can only be performed using a SQL statement.

To hide a materialized view (SQL)

1. Connect to the database as a user with DBA authority, or as owner of the materialized view.
2. Execute an ALTER MATERIALIZED VIEW statement with the SET HIDDEN clause.

Example

The following statements create a materialized view, EmployeeConfid3, refreshes it, and then obfuscate its view definition.

```
CREATE MATERIALIZED VIEW EmployeeConfid3 AS
   SELECT EmployeeID, Employees.DepartmentID, SocialSecurityNumber, Salary,
ManagerID,
      Departments.DepartmentName, Departments.DepartmentHeadID
   FROM Employees, Departments
   WHERE Employees.DepartmentID=Departments.DepartmentID;
REFRESH MATERIALIZED VIEW EmployeeConfid3;
ALTER MATERIALIZED VIEW EmployeeConfid3 SET HIDDEN;
```

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" on page 55.

See also

- "ALTER MATERIALIZED VIEW statement" [*SQL Anywhere Server - SQL Reference*]

Drop materialized views

When a materialized view is no longer needed, you can drop it.

Before you can drop a materialized view, you must drop or disable all dependent views. To determine whether there are views dependent on a materialized view, use the sa_dependent_views system procedure. See "sa_dependent_views system procedure" [*SQL Anywhere Server - SQL Reference*].

See also "View dependencies" on page 22.

To drop a materialized view (Sybase Central)

1. Use the SQL Anywhere 12 plug-in to connect to the database as a user with DBA authority or as the owner of the view.
2. In the left pane, double-click **Views**.
3. Right-click the materialized view and choose **Delete**.
4. Click **Yes**.

To drop a materialized view (SQL)

1. Connect to the database as a user with DBA authority, or as the owner of the view.
2. Execute a DROP MATERIALIZED VIEW statement.

Example

The following statement creates the EmployeeConfid4 materialized view, initializes it (populates it with data), and then drops it.

```
CREATE MATERIALIZED VIEW EmployeeConfid4 AS
   SELECT EmployeeID, Employees.DepartmentID, SocialSecurityNumber, Salary,
ManagerID,
      Departments.DepartmentName, Departments.DepartmentHeadID
   FROM Employees, Departments
   WHERE Employees.DepartmentID=Departments.DepartmentID;
REFRESH MATERIALIZED VIEW EmployeeConfid4;
DROP MATERIALIZED VIEW EmployeeConfid4;
```

See also

- "DROP MATERIALIZED VIEW statement" [*SQL Anywhere Server - SQL Reference*]
- "sa_dependent_views system procedure" [*SQL Anywhere Server - SQL Reference*]
- "View dependencies" on page 22

Working with indexes

Performance is an important consideration when designing and creating your database. Indexes can dramatically improve the performance of statements that search for a specific row or a specific subset of the rows. On the other hand, indexes take up additional disk space and can slow inserts, updates, and deletes.

When to use indexes

An index provides an ordering on the rows in a column or columns of a table. An index is like a telephone book that initially sorts people by surname, and then sorts identical surnames by first names. This ordering speeds up searches for phone numbers for a particular surname, but it does not provide help in finding the phone number at a particular address. In the same way, a database index is useful only for searches on a specific column or columns.

Indexes get more useful as the size of the table increases. The average time to find a phone number at a given address increases with the size of the phone book, while it does not take much longer to find the phone number of K. Kaminski in a large phone book than in a small phone book.

The database server query optimizer automatically uses an index when a suitable index exists and when using one will improve performance.

There are some down sides to creating indexes. In particular, any indexes must be maintained along with the table itself when the data in a column is modified, so that the performance of inserts, updates, and deletes can be affected by indexes. For this reason, unnecessary indexes should be dropped. Use the Index Consultant to identify unnecessary indexes. See "Obtain Index Consultant recommendations for a query" on page 165.

Deciding what indexes to create

Choosing an appropriate set of indexes for a database is an important part of optimizing performance. Identifying an appropriate set can also be a demanding problem. The performance benefits from some indexes can be significant, but there are also costs associated with indexes, in both storage space and in overhead when altering data.

The Index Consultant is a tool that assists you in proper selection of indexes. It analyzes either a single query or a set of operations, and recommends which indexes to add to your database. It also notifies you of indexes that are unused. See "Obtain Index Consultant recommendations for a query" on page 165.

Indexes on frequently-searched columns

SQL Anywhere automatically indexes primary key and foreign key columns. So, manually creating an index on a key column is not necessary and is generally not recommended. If a column is only part of a key, an index can help.

Indexes require extra space and can slightly reduce the performance of statements that modify the data in the table, such as INSERT, UPDATE, and DELETE statements. However, they can improve search performance dramatically and are highly recommended whenever you search data frequently. To learn more about how indexes improve performance, see "Using indexes" on page 225.

The optimizer automatically uses indexes to improve the performance of any database statement whenever it is possible to do so. Also, the index is updated automatically when rows are deleted, updated,

or inserted. While you can explicitly refer to indexes using index hints when forming your query, there is no need to.

Index hints

You can supply index hints when forming a query. Index hints override the optimizer's choice of query access plan by forcing the use of a particular index or indexes. Index hints are typically only used when evaluating the optimizer's choice of plans, and should be used only by advanced users and database administrators. Improper application of index hinting can lead to poor query performance.

You specify index hints using subclauses of the FROM clause. For example, the INDEX clause allows you to specify up to four indexes. The optimizer must be able to use all the specified indexes, otherwise an error is returned.

Specify NO INDEX to disable the use of indexes for the query, and force a sequential scan of the table instead. However, sequential scans are very costly, and take longer to execute. Use this clause only for comparison purposes when evaluating the optimizer's index selection.

By default, if a query can be satisfied using only index data (that is, without having to access rows in the table), the database server performs an index-only retrieval. However, you may want to specify INDEX ONLY ON so that an error is returned if the indexes can no longer be used for index-only retrieval (for example, if they are changed or dropped).

For more information about the index hint clauses you can specify in the FROM clause, see "FROM clause" [*SQL Anywhere Server - SQL Reference*].

Using clustered indexes

Although indexes can dramatically improve the performance of statements that search for a specific range of key values, two rows appearing sequentially in the index do not necessarily appear on the same table page in the database.

You can further improve a large index scan by declaring that the index is clustered. Using a clustered index increases the chance that two rows from adjacent index entries will appear on the same page in the database. This can lead to performance benefits by reducing the number of times a table page needs to be read into the buffer pool.

The existence of an index with a clustering property causes the database server to attempt to store table rows in approximately the same order as they appear in the clustered index. However, while the database server attempts to preserve the key order, clustering is approximate and total clustering is not guaranteed. So, the database server cannot sequentially scan the table and retrieve all the rows in a clustered index key sequence. Ensuring that the rows of the table are returned in sorted order requires an access plan that either accesses the rows through the index, or performs a physical sort.

The optimizer exploits an index with a clustering property by modifying the expected cost of indexed retrieval to take into account the expected physical adjacency of table rows with matching or adjacent index key values.

The amount of clustering for a given table may degrade over time, as more and more rows are inserted or updated. The database server automatically keeps track of the amount of clustering for each clustered index in the ISYSPHYSIDX system table. If the database server detects that the rows in a table have become significantly unclustered, the optimizer will adjust its expected index retrieval costs.

If you decide to make one of the indexes on a table clustered, you need to consider the expected query workload. Some experimentation is usually required. Generally, the database server can use a clustered index to improve performance when the following conditions hold for a specified query:

- Many of the table pages required for answering the query are not already in memory. When the table pages are already in memory, the server does not need to read these pages and such clustering is irrelevant.
- The query can be answered by performing an index retrieval that is expected to return a non-trivial number of rows. As an example, clustering is usually irrelevant for simple primary key searches.
- The database server actually needs to read table pages, as opposed to performing an index-only retrieval.

Using SQL statements to cluster the index

The clustering property of an index can be added or removed at any time using SQL statements. Any primary key index, foreign key index, UNIQUE constraint index, or secondary index can be declared with the CLUSTERED property. However, you may declare at most one clustered index per table. You can do this using any of the following statements:

- "CREATE TABLE statement" [*SQL Anywhere Server - SQL Reference*]
- "ALTER DATABASE statement" [*SQL Anywhere Server - SQL Reference*]
- "CREATE INDEX statement" [*SQL Anywhere Server - SQL Reference*]
- "DECLARE LOCAL TEMPORARY TABLE statement" [*SQL Anywhere Server - SQL Reference*]

Several statements work in conjunction with each other to allow you to maintain and restore the clustering effect:

- The UNLOAD TABLE statement allows you to unload a table in the order of the clustered index key. See "UNLOAD statement" [*SQL Anywhere Server - SQL Reference*].
- The LOAD TABLE statement inserts rows into the table in the order of the clustered index key. See "LOAD TABLE statement" [*SQL Anywhere Server - SQL Reference*].
- The INSERT statement attempts to put new rows on the same table page as the one containing adjacent rows, as per the clustered index key. See "INSERT statement" [*SQL Anywhere Server - SQL Reference*].
- The REORGANIZE TABLE statement restores the clustering of a table by rearranging the rows according to the clustered index. If REORGANIZE TABLE is used with tables where clustering is not

specified, the tables are reordered using the primary key. See "REORGANIZE TABLE statement" [*SQL Anywhere Server - SQL Reference*].

Creating clustered indexes in Sybase Central

You can also create clustered indexes in Sybase Central using the Create Index Wizard, and selecting Create A Clustered Index when prompted. See "Create indexes" on page 60.

Reordering rows to match a clustered index

To reorder the rows in a table to match a clustered index, use the REORGANIZE TABLE statement. See "REORGANIZE TABLE statement" [*SQL Anywhere Server - SQL Reference*].

Create indexes

Indexes are created on one or more columns of a specified table. You can create indexes on base tables or temporary tables, but you cannot create an index on a view. To create an individual index, you can use either Sybase Central or Interactive SQL. You can use the Index Consultant to guide you in a proper selection of indexes for your database.

When creating indexes, the order in which you specify the columns becomes the order in which the columns appear in the index. Duplicate references to column names in the index definition is not allowed.

To create a new index (Sybase Central)

1. Use the SQL Anywhere 12 plug-in to connect to the database as a user with DBA authority.
2. In the left pane, double-click **Tables** and select the table for which you want to create an index.
3. In the right pane, click the **Indexes** tab.
4. In the left pane, right-click the table and choose **New** » **Index**.
5. Follow the instructions in the **Create Index Wizard**.

   The new index appears on the **Index** tab for the table. It also appears in **Indexes**.

To create a new index (SQL)

1. Connect to the database as a user with DBA authority, or as the owner of the table on you are creating the index.
2. Execute a CREATE INDEX statement.

In addition to creating indexes on one or more columns in a table, you can create indexes on a built-in function using a computed column. See "CREATE INDEX statement" [*SQL Anywhere Server - SQL Reference*].

Example

The following example creates an index called EmployeeNames on the Employees table, using the Surname and GivenName columns:

```
CREATE INDEX EmployeeNames
ON Employees (Surname, GivenName);
```

See "CREATE INDEX statement" [*SQL Anywhere Server - SQL Reference*], and "Improving database performance" on page 157.

Validate indexes

You can validate an index to ensure that every row referenced in the index actually exists in the table. For foreign key indexes, a validation check also ensures that the corresponding row exists in the primary table. This check complements the validity checking performed by the VALIDATE TABLE statement.

Caution
Validate tables or entire databases only when no connections are making changes to the database.

To validate an index (Sybase Central)

1. Use the SQL Anywhere 12 plug-in to connect to the database as a user with DBA authority or as the owner of the table on which the index is created.
2. In the left pane, double-click **Indexes**.
3. Right-click the index and choose **Validate**.
4. Click **OK**.

To validate an index (SQL)

1. Connect to the database as a user with DBA authority, or as the owner of the table on which the index is created.
2. Execute a VALIDATE INDEX statement.

To validate an index (dbvalid utility)

- Run a dbvalid command with the -i option specified.

Example 1

Validate an index called EmployeeNames. If you supply a table name instead of an index name, the primary key index is validated.

```
VALIDATE INDEX EmployeeNames;
```

See "VALIDATE statement" [*SQL Anywhere Server - SQL Reference*].

Example 2

Validate an index called EmployeeNames. The -I option specifies that each object name given is an index.

```
dbvalid -I EmployeeNames
```

See "Validation utility (dbvalid)" [*SQL Anywhere Server - Database Administration*].

Rebuild indexes

Sometimes it is necessary to rebuild an index because it has become fragmented or skewed due to extensive insertion and deletion operations on the table. When you rebuild an index, you rebuild the physical index. All logical indexes that use the physical index benefit from the rebuild operation. You do not need to perform a rebuild on logical indexes. See "Index sharing using logical indexes" on page 623.

You can rebuild indexes in Sybase Central, or by executing an ALTER INDEX ... REBUILD statement. You can also rebuild indexes as part of an effort to remove table fragmentation using the REORGANIZE TABLE statement. This section describes how to rebuild indexes using Sybase Central and the ALTER INDEX ... REBUILD statement. For more information about using the REORGANIZE TABLE statement, see "REORGANIZE TABLE statement" [*SQL Anywhere Server - SQL Reference*].

To rebuild an index (Sybase Central)

1. Use the SQL Anywhere 12 plug-in to connect to the database as a user with DBA authority or as the owner of the table on which the index is created.
2. In the left pane, double-click **Indexes**.
3. Right-click the index and choose **Rebuild**.
4. Click **OK**.

To rebuild an index (SQL)

1. Connect to the database as a user with DBA authority, or as the owner of the table associated with the index.
2. Execute an ALTER INDEX ... REBUILD statement.

Example

The following statement rebuilds the IX_customer_name index on the Customers table:


```
ALTER INDEX IX_customer_name ON Customers REBUILD;
```

For more information about the syntax for the ALTER INDEX statement, see "ALTER INDEX statement" [*SQL Anywhere Server - SQL Reference*].

See also

For more information about index fragmentation and skew, and how to reduce them, see "Reduce index fragmentation and skew" on page 219.

For more information about how to detect index fragmentation and skew, see "Application Profiling Wizard" on page 158, and "sa_index_density system procedure" [*SQL Anywhere Server - SQL Reference*].

Drop indexes

If an index is no longer required, you can delete it from the database in Sybase Central or in Interactive SQL.

To drop an index (Sybase Central)

1. Use the SQL Anywhere 12 plug-in to connect to the database as a user with DBA authority or as the owner of the table on which the index is created.
2. In the left pane, double-click **Indexes**.
3. Right-click the index and choose **Delete**.
4. Click **Yes**.

To drop an index (SQL)

1. Connect to the database as a user with DBA authority, or as the owner of the table associated with the index.
2. Execute a DROP INDEX statement.

Example

The following statement removes the EmployeeNames index from the database:

```
DROP INDEX EmployeeNames;
```

See "DROP INDEX statement" [*SQL Anywhere Server - SQL Reference*].

Index information in the catalog

The ISYSIDX system table provides a list of all indexes in the database, including primary and foreign key indexes. Additional information about the indexes is found in the ISYSPHYSIDX, ISYSIDXCOL,

and ISYSFKEY system views. You can use Sybase Central or Interactive SQL to browse the views for these tables to see the data they contain.

Following is a brief overview of how index information is stored in the system tables:

- **ISYSIDX system table** The central table for tracking indexes, each row in the ISYSIDX system table defines a logical index (PKEY, FKEY, UNIQUE constraint, Secondary index) in the database. See "SYSIDX system view" [*SQL Anywhere Server - SQL Reference*] and "Index sharing using logical indexes" on page 623.

- **ISYSPHYSIDX system table** Each row in the ISYSPHYSIDX system table defines a physical index in the database. See "SYSPHYSIDX system view" [*SQL Anywhere Server - SQL Reference*] and "Index sharing using logical indexes" on page 623.

- **ISYSIDXCOL system table** Just as each row in the SYSIDX system view describes one index in the database, each row in the SYSIDXCOL system view describes one column of an index described in the SYSIDX system view. See "SYSIDXCOL system view" [*SQL Anywhere Server - SQL Reference*].

- **ISYSFKEY system table** Every foreign key in the database is defined by one row in the ISYSFKEY system table and one row in the ISYSIDX system table. See "SYSFKEY system view" [*SQL Anywhere Server - SQL Reference*].

Ensuring data integrity

If data has integrity, the data is valid—correct and accurate—and the relational structure of the database is intact. Referential integrity constraints enforce the relational structure of the database. These rules maintain the consistency of data between tables. Building integrity constraints into the database is the best way to make sure your data remains consistent.

You can enforce several types of referential integrity checks. For example, you can ensure individual entries are correct by imposing constraints and CHECK constraints on tables and columns. You can also configure column properties by choosing an appropriate data type or setting special default values.

SQL Anywhere supports stored procedures, which give you detailed control over how data enters the database. You can also create triggers, or customized stored procedures that are invoked automatically when a certain action, such as an update of a particular column, occurs.

For more information about procedures and triggers see "Using procedures, triggers, and batches" on page 793.

How your data can become invalid

Data in your database may become invalid if proper checks are not made. You can prevent each of these examples from occurring using facilities described in this section.

Incorrect information

- An operator types the date of a sales transaction incorrectly.
- An employee's salary becomes ten times too small because the operator missed a digit.

Duplicated data

- Two different employees add the same new department (with DepartmentID 200) to the Departments table of the organization's database.

Foreign key relations invalidated

- The department identified by DepartmentID 300 closes down and one employee record inadvertently remains unassigned to a new department.

Building integrity constraints into your database

To ensure the validity of data in a database, you need to formulate checks to define valid and invalid data, and design rules to which data must adhere (also known as business rules). Typically, business rules are implemented through check constraints, user-defined data types, and the appropriate use of transactions.

Constraints that are built into the database are more reliable than constraints that are built into client applications or that are provided as instructions to database users. Constraints built into the database

become part of the definition of the database itself, and the database enforces them consistently across all applications. Setting a constraint once in the database imposes it for all subsequent interactions with the database.

In contrast, constraints built into client applications are vulnerable every time the software changes, and may need to be imposed in several applications, or in several places in a single client application.

How the contents of your database change

Changes occur to information in database tables when you submit SQL statements from client applications. Only a few SQL statements actually modify the information in a database. You can:

- Update information in a row of a table using the UPDATE statement.
- Delete an existing row of a table using the DELETE statement.
- Insert a new row into a table using the INSERT statement.

Tools for maintaining data integrity

To maintain data integrity, you can use defaults, data constraints, and constraints that maintain the referential structure of the database.

Defaults

You can assign default values to columns to make certain kinds of data entry more reliable. For example:

- A column can have a current date default value for recording the date of transactions with any user or client application action.
- Other types of default values allow column values to increment automatically without any specific user action other than entering a new row. With this feature, you can guarantee that items (such as purchase orders for example) are unique, sequential numbers.

For more information about these and other column defaults, see “Using column defaults” on page 67.

Primary keys

Primary keys guarantee that every row of a given table can be uniquely identified in the table. See “Primary keys” [*SQL Anywhere 12 - Introduction*].

Table and column constraints

The following constrains maintain the structure of data in the database, and define the relationship between tables in a relational database:

- **Referential constraints** Data integrity is also maintained using referential constraints, also called **RI constraints** (for referential integrity constraints). RI constraints are data rules that are set on

columns and tables to control what the data can be. RI constraints define the relationship between tables in a relational database.

For more information about enforcing referential integrity, see "Enforcing entity and referential integrity" on page 81.

- **NOT NULL constraint** A NOT NULL constraint prevents a column from containing a NULL entry.
- **CHECK constraint** A CHECK constraint assigned to a column can ensure that every item in the column meets a particular condition. For example, you can ensure that Salary column entries fit within a specified range and are protected from user error when new values are entered.

  CHECK constraints can be made on the relative values in different columns. For example, you can ensure that a DateReturned entry is later than a DateBorrowed entry in a library database.

Column constraints can be inherited from domains. For more information about these and other table and column constraints, see "Using table and column constraints" on page 74.

Triggers for advanced integrity rules

A **trigger** is a procedure stored in the database and executed automatically whenever the information in a specified table changes. Triggers are a powerful mechanism for database administrators and developers to ensure that data remains reliable. You can also triggers to maintain data integrity. Triggers can enforce more sophisticated CHECK conditions. See "Using procedures, triggers, and batches" on page 793.

SQL statements for implementing integrity constraints

The following SQL statements implement integrity constraints:

- **CREATE TABLE statement** This statement implements integrity constraints during creation of the table.
- **ALTER TABLE statement** This statement adds integrity constraints to an existing table, or modifies constraints for an existing table.
- **CREATE TRIGGER statement** This statement creates triggers that enforce more complex business rules.
- **CREATE DOMAIN statement** This statement creates a user-defined data type. The definition of the data type can include constraints.

For more information about the syntax of these statements, see "SQL statements" [*SQL Anywhere Server - SQL Reference*].

Using column defaults

Column defaults automatically assign a specified value to a particular column whenever someone enters a new row into a database table. The default value assigned requires no action on the part of the client application, however if the client application does specify a value for the column, the new value overrides the column default value.

Column defaults can quickly and automatically fill columns with information, such as the date or time a row is inserted, or the user ID of the person entering the information. Using column defaults encourages data integrity, but does not enforce it. Client applications can always override defaults.

When default values are defined using variables that start with @, the value used for the default is value of the variable at the moment the DML or load statement is executed.

Supported default values

SQL supports the following default values:

- A string specified in the CREATE TABLE statement or ALTER TABLE statement.
- A number specified in the CREATE TABLE statement or ALTER TABLE statement.
- AUTOINCREMENT: an automatically incremented number that is one more than the previous highest value in the column.
- Default GLOBAL AUTOINCREMENT, which ensures unique primary keys across multiple databases.
- Universally Unique Identifiers (UUIDs) generated using the NEWID function.
- The current date, time, or timestamp.
- The current user ID of the database user.
- A NULL value.
- A constant expression, as long as it does not reference database objects.

Creating column defaults

You can use the CREATE TABLE statement to create column defaults at the time a table is created, or the ALTER TABLE statement to add column defaults at a later time.

Example

The following statement adds a default to an existing column named ID in the SalesOrders table, so that it automatically increments (unless a client application specifies a value). Note that in the SQL Anywhere sample database, this column is already set to AUTOINCREMENT.

```
ALTER TABLE SalesOrders
ALTER ID DEFAULT AUTOINCREMENT;
```

For more information, see "ALTER TABLE statement" [*SQL Anywhere Server - SQL Reference*] and "CREATE TABLE statement" [*SQL Anywhere Server - SQL Reference*].

Altering and dropping column defaults

You can change or remove column defaults using the same form of the ALTER TABLE statement you used to create the defaults. The following statement changes the default value of a column named OrderDate from its current setting to CURRENT DATE:

```
ALTER TABLE SalesOrders
ALTER OrderDate DEFAULT CURRENT DATE;
```

You can remove column defaults by modifying them to be NULL. The following statement removes the default from the OrderDate column:

```
ALTER TABLE SalesOrders
ALTER OrderDate DEFAULT NULL;
```

Work with column defaults

You can add, alter, and drop column defaults in Sybase Central using the **Value** tab of the **Column Properties** window.

To display the Properties window for a column (Sybase Central)

1. Use the SQL Anywhere 12 plug-in to connect to the database as a user with DBA authority.
2. In the left pane, double-click **Tables**.
3. Click the table.
4. Click the **Columns** tab.
5. Right-click the column and choose **Properties**.

Current date and time defaults

For columns with the DATE, TIME, or TIMESTAMP data type, you can use the current date, current time, or current timestamp as a default. The default you choose must be compatible with the column's data type.

Useful examples of current date default

A current date default might be useful to record:

- dates of phone calls in a contacts database
- dates of orders in a sales entry database
- the date a patron borrows a book in a library database

Current timestamp

The current timestamp is similar to the current date default, but offers greater accuracy. For example, a user of a contact management application may have several interactions with a single customer in one day: the current timestamp default would be useful to distinguish these contacts.

Since it records a date and the time down to a precision of millionths of a second, you may also find the current timestamp useful when the sequence of events is important in a database.

Default timestamp

The default timestamp 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. See the DEFAULT clause in "CREATE TABLE statement" [*SQL Anywhere Server - SQL Reference*].

For more information about timestamps, times, and dates, see "SQL data types" [*SQL Anywhere Server - SQL Reference*].

The user ID defaults

Assigning a DEFAULT USER to a column is an easy and reliable way of identifying the person making an entry in a database. This information may be required; for example, when salespeople are working on commission.

Building a user ID default into the primary key of a table is a useful technique for occasionally connected users, and helps to prevent conflicts during information updates. These users can make a copy of tables relevant to their work on a portable computer, make changes while not connected to a multi-user database, and then apply the transaction log to the server when they return.

The LAST USER special value specifies the name of the user who last modified the row. 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 "LAST USER special value" [*SQL Anywhere Server - SQL Reference*].

The AUTOINCREMENT default

The AUTOINCREMENT default is useful for numeric data fields where the value of the number itself may have no meaning. The feature assigns each new row a unique value larger than any other value in the column. You can use AUTOINCREMENT columns to record purchase order numbers, to identify customer service calls or other entries where an identifying number is required.

Autoincrement columns are typically primary key columns or columns constrained to hold unique values. See "Enforcing entity integrity" on page 81.

You can retrieve the most recent value inserted into an autoincrement column using the @@identity global variable. See "@@identity global variable" [*SQL Anywhere Server - SQL Reference*].

Autoincrement and negative numbers

Autoincrement is intended to work with positive integers.

The initial autoincrement value is set to 0 when the table is created. This value remains as the highest value assigned when inserts are done that explicitly insert negative values into the column. An insert where no value is supplied causes the AUTOINCREMENT to generate a value of 1, forcing any other generated values to be positive.

Autoincrement and the IDENTITY column

A column with the AUTOINCREMENT default is referred to in Transact-SQL applications as an IDENTITY column.

For information about IDENTITY columns, see "The special IDENTITY column" on page 650.

See also

- "Reloading tables with autoincrement columns" [*SQL Anywhere 12 - Changes and Upgrading*]
- "CREATE TABLE statement" [*SQL Anywhere Server - SQL Reference*]
- "The GLOBAL AUTOINCREMENT default" on page 71
- "Choosing between sequences and autoincrement values" on page 150
- "Global autoincrement columns" [*SQL Remote*]

The GLOBAL AUTOINCREMENT default

The GLOBAL AUTOINCREMENT default is intended for use when multiple databases are used in a SQL Remote replication or MobiLink synchronization environment. It ensures unique primary keys across multiple databases.

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 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 option, 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 $(n + 1)\ p$, 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 $(n + 1)\ p$, the next default value is one greater than the previous largest value in 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 synchronization.

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. This situation arises, for example, if the column is contained in the table's primary key.

NULL default values are also generated when the supply of values within the partition has been exhausted. 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. Attempting to insert the NULL value causes an error if the column does not permit NULLs. 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*].

Global autoincrement columns are typically primary key columns or columns constrained to hold unique values. See "Enforcing entity integrity" on page 81.

While using the global autoincrement default in other cases is possible, doing so can adversely affect database performance. For example, when the next value for each column is stored as a 64-bit signed integer, using values greater than $2^{31} - 1$ or large double or numeric values may cause wraparound to negative values.

You can retrieve the most recent value inserted into an autoincrement column using the @@identity global variable. See "@@identity global variable" [*SQL Anywhere Server - SQL Reference*].

See also

- "Using global autoincrement" [*MobiLink - Server Administration*]
- "Global autoincrement columns" [*SQL Remote*]
- "CREATE TABLE statement" [*SQL Anywhere Server - SQL Reference*]
- "Reloading tables with autoincrement columns" [*SQL Anywhere 12 - Changes and Upgrading*]
- "The AUTOINCREMENT default" on page 70
- "Choosing between sequences and autoincrement values" on page 150

The NEWID default

Universally Unique Identifiers (UUIDs), also known as Globally Unique Identifiers (GUIDs), can be used to identify unique rows in a table. The values are generated such that a value produced on one computer

will not match that produced on another. They can therefore be used as keys in replication and synchronization environments.

Using UUID values as primary keys has some tradeoffs when you compare them with using GLOBAL AUTOINCREMENT values. For example:

- UUIDs can be easier to set up than GLOBAL AUTOINCREMENT, since there is no need to assign each remote database a unique database ID. There is also no need to consider the number of databases in the system or the number of rows in individual tables. The Extraction utility (dbxtract) can be used to deal with the assignment of database IDs. This isn't usually a concern for GLOBAL AUTOINCREMENT if the BIGINT data type is used, but it needs to be considered for smaller data types.

- UUID values are considerably larger than those required for GLOBAL AUTOINCREMENT, and will require more table space in both primary and foreign tables. Indexes on these columns will also be less efficient when UUIDs are used. In short, GLOBAL AUTOINCREMENT is likely to perform better.

- UUIDs have no implicit ordering. For example, if A and B are UUID values, A > B does not imply that A was generated after B, even when A and B were generated on the same computer. If you require this behavior, an additional column and index may be necessary.

See also

- "NEWID function [Miscellaneous]" [*SQL Anywhere Server - SQL Reference*]
- "UNIQUEIDENTIFIER data type" [*SQL Anywhere Server - SQL Reference*]

The NULL default

For columns that allow NULL values, specifying a NULL default is exactly the same as not specifying a default at all. If the client inserting the row does not explicitly assign a value, the row automatically receives A NULL value.

You can use NULL defaults when information for some columns is optional or not always available.

For more information about the NULL value, see "NULL value" [*SQL Anywhere Server - SQL Reference*].

String and number defaults

You can specify a specific string or number as a default value, as long as the column has a string or numeric data type. You must ensure that the default specified can be converted to the column's data type.

Default strings and numbers are useful when there is a typical entry for a given column. For example, if an organization has two offices, the headquarters in city_1 and a small office in city_2, you may want to set a default entry for a location column to city_1, to make data entry easier.

Constant expression defaults

You can use a constant expression as a default value, as long as it does not reference database objects. Constant expressions allow column defaults to contain entries such as *the date fifteen days from today*, which would be entered as

```
... DEFAULT ( DATEADD( day, 15, GETDATE() ) );
```

Using table and column constraints

Along with the basic table structure (number, name and data type of columns, name and location of the table), the CREATE TABLE statement and ALTER TABLE statement can specify many different table attributes that allow control over data integrity. Constraints allow you to place restrictions on the values that can appear in a column, or on the relationship between values in different columns. Constraints can be either table-wide constraints, or can apply to individual columns.

This section describes how to use constraints to help ensure the accuracy of data in the table.

Using CHECK constraints on columns

You use a CHECK condition to ensure that the values in a column satisfy some criteria or rule. These rules or criteria may be required to verify that the data is correct, or they may be more rigid rules that reflect organization policies and procedures. CHECK conditions on individual column values are useful when only a restricted range of values are valid for that column.

Once a CHECK condition is in place, future values are evaluated against the condition before a row is modified. When you update a value that has a check constraint, the constraints for that value and for the rest of the row are checked.

Variables are not allowed in CHECK constraints on columns. Any string starting with @ within a column CHECK constraint is replaced with the name of the column the constraint is on.

If the column data type is a domain, the column inherits any CHECK constraints defined for the domain. See “Inheriting column CHECK constraints from domains” on page 76.

Note
Column CHECK tests fail if the condition returns a value of FALSE. If the condition returns a value of UNKNOWN, the behavior is as though it returns TRUE, and the value is allowed.

For more information about valid conditions, see “Search conditions” [*SQL Anywhere Server - SQL Reference*].

Example 1

You can enforce a particular formatting requirement. For example, if a table has a column for phone numbers you may want to ensure that users enter them all in the same manner. For North American phone numbers, you could use a constraint such as:

```
ALTER TABLE Customers
ALTER Phone
CHECK ( Phone LIKE '(___) ___-____' );
```

Once this CHECK condition is in place, if you attempt to set a Phone value to 9835, for example, the change is not allowed.

Example 2

You can ensure that the entry matches one of a limited number of values. For example, to ensure that a City column only contains one of a certain number of allowed cities (such as those cities where the organization has offices), you could use a constraint such as:

```
ALTER TABLE Customers
ALTER City
CHECK ( City IN ( 'city_1', 'city_2', 'city_3' ) );
```

By default, string comparisons are case insensitive unless the database is explicitly created as a case-sensitive database.

Example 3

You can ensure that a date or number falls in a particular range. For example, you may require that the StartDate of an employee be between the date the organization was formed and the current date using the following constraint:

```
ALTER TABLE Employees
ALTER StartDate
CHECK ( StartDate BETWEEN '1983/06/27'
                  AND CURRENT DATE );
```

You can use several date formats. The YYYY/MM/DD format in this example has the virtue of always being recognized regardless of the current option settings.

Using CHECK constraints on tables

A CHECK condition applied as a constraint on the table typically ensures that two values in a row being added or modified have a proper relation to each other.

When you give a name to the constraint, the constraint is held individually in the system tables, and you can replace or drop them individually. Since this is more flexible behavior, it is recommended that you either name a CHECK constraint or use an individual column constraint wherever possible.

For example, you can add a constraint on the Employees table to ensure that the TerminationDate is always later than, or equal to, the StartDate:

```
ALTER TABLE Employees
   ADD CONSTRAINT valid_term_date
   CHECK( TerminationDate >= StartDate );
```

You can specify variables within table CHECK constraints but their names must begin with @. The value used is the value of the variable at the moment the DML or LOAD statement is executed.

For more information, see "ALTER TABLE statement" [*SQL Anywhere Server - SQL Reference*].

Inheriting column CHECK constraints from domains

You can attach CHECK constraints to domains. Columns defined on those domains inherit the CHECK constraints. A CHECK constraint explicitly specified for the column overrides that from the domain. For example, the CHECK clause in this domain definition requires that values inserted into columns only be positive integers.

```
CREATE DATATYPE positive_integer INT
CHECK ( @col > 0 );
```

Any column defined using the positive_integer domain accepts only positive integers unless the column itself has a CHECK constraint explicitly specified. Since any variable prefixed with the @ sign is replaced by the name of the column when the CHECK constraint is evaluated, any variable name prefixed with @ could be used instead of @col.

An ALTER TABLE statement with the DELETE CHECK clause drops all CHECK constraints from the table definition, including those inherited from domains.

Any changes made to a constraint in a domain definition (after a column is defined on that domain) are not applied to the column. The column gets the constraints from the domain when it is created, but there is no further connection between the two.

See also

- "Domains" [*SQL Anywhere Server - SQL Reference*]
- "Using CHECK constraints on columns" on page 74

Manage constraints

In Sybase Central, you add, alter, and drop column constraints on the **Constraints** tab of the table or **Column Properties** window.

To manage constraints (Sybase Central)

1. Use the SQL Anywhere 12 plug-in to connect to the database as a user with DBA authority.
2. In the left pane, double-click **Tables**.
3. Click the table you want to alter.

4. In the right pane, click the **Constraints** tab and modify an existing constraint or add a new constraint.

Manage UNIQUE constraints

For a column, a UNIQUE constraint specifies that the values in the column must be unique. For a table, the UNIQUE constraint identifies one or more columns that identify unique rows 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.

To manage unique constraints (Sybase Central)

1. Use the SQL Anywhere 12 plug-in to connect to the database as a user with DBA authority.
2. In the left pane, double-click **Tables**.
3. Click the table you want to alter.
4. In the right pane, click the **Constraints** tab.
5. Right-click in the **Constraints** tab and choose **New** » **Unique Constraint**.
6. Complete the instructions in the **Create Unique Constraint Wizard**.

Altering and dropping CHECK constraints

Altering tables can interfere with other users of the database. Although you can execute the ALTER TABLE statement while other connections are active, you cannot execute the ALTER TABLE statement if any other connection is using the table you want to alter. For large tables, ALTER TABLE is a time-consuming operation, and all other requests referencing the table being altered are prohibited while the statement is processing. See “ALTER TABLE statement” [*SQL Anywhere Server - SQL Reference*].

There are several ways to alter the existing set of CHECK constraints on a table.

- You can add a new CHECK constraint to the table or to an individual column.
- You can drop a CHECK constraint on a column by setting it to NULL. For example, the following statement removes the CHECK constraint on the Phone column in the Customers table:

```
ALTER TABLE Customers
ALTER Phone CHECK NULL;
```

- You can replace a CHECK constraint on a column in the same way as you would add a CHECK constraint. For example, the following statement adds or replaces a CHECK constraint on the Phone column of the Customers table:

```
ALTER TABLE Customers
ALTER Phone
CHECK ( Phone LIKE '___-___-____' );
```

- You can alter a CHECK constraint defined on the table:
  - You can add a new CHECK constraint using ALTER TABLE with an ADD *table-constraint* clause.
  - If you have defined constraint names, you can alter individual constraints.
  - If you have not defined constraint names, you can drop all existing CHECK constraints (including column CHECK constraints and CHECK constraints inherited from domains) using ALTER TABLE DELETE CHECK, and then add in new CHECK constraints.

    To use the ALTER TABLE statement with the DELETE CHECK clause:

    ```
    ALTER TABLE table-name
    DELETE CHECK;
    ```

Sybase Central lets you add, alter and drop both table and column CHECK constraints. For more information, see "Manage constraints" on page 76.

Dropping a column from a table does not drop CHECK constraints associated with the column held in the table constraint. Not removing the constraints produces a `column not found` error message upon any attempt to insert, or even just query, data in the table.

Note
Table CHECK constraints fail if a value of FALSE is returned. If the condition returns a value of UNKNOWN the behavior is as though it returned TRUE, and the value is allowed.

Using domains

A **domain** is a user-defined data type that, together with other attributes, can restrict the range of acceptable values or provide defaults. A domain extends one of the built-in data types. Normally, the range of permissible values is restricted by a check constraint. In addition, a domain can specify a default value and may or may not allow NULLs.

Defining your own domains provides many benefits including:

- Preventing common errors if inappropriate values are entered. A constraint placed on a domain ensures that all columns and variables intended to hold values in a range or format can hold only the intended values. For example, a data type can ensure that all credit card numbers typed into the database contain the correct number of digits.
- Making the applications and the structure of a database easier to understand.
- Convenience. For example, you may intend that all table identifiers are positive integers that, by default, auto-increment. You could enforce this restriction by entering the appropriate constraints and defaults each time you define a new table, but it is less work to define a new domain, then simply state that the identifier can take only values from the specified domain.

For more information about domains, see "Domains" [*SQL Anywhere Server - SQL Reference*].

Create domains (Sybase Central)

You can use Sybase Central to create a domain or assign it to a column.

To create a new domain (Sybase Central)

1. Use the SQL Anywhere 12 plug-in to connect to the database as a user with DBA authority.
2. In the left pane, right-click **Domains** and choose **New** » **Domain**.
3. Follow the instructions in the **Create Domain Wizard**.

To assign a domain to a column (Sybase Central)

1. Use the SQL Anywhere 12 plug-in to connect to the database as a user with DBA authority.
2. In the left pane, double-click **Tables**.
3. Click the table.
4. In the right pane, click the **Columns** tab.
5. Select a column and in the **Data Type** field click the ellipsis (three dots) button.
6. Click the **Data Type** tab and select **Domain**.
7. In the **Domain** list, select a domain.
8. Click **OK**.

Create domains (SQL)

You can use the CREATE DOMAIN statement to create and define domains. See "CREATE DOMAIN statement" [*SQL Anywhere Server - SQL Reference*].

Some predefined domains are included with SQL Anywhere. For example, the monetary domain MONEY.

To create a new domain (SQL)

1. Connect to a database.
2. Execute a CREATE DOMAIN statement.

Example 1: Simple domains

Some columns in the database are used for employee names and others to store addresses. You might then define the following domains.

```
CREATE DOMAIN persons_name CHAR(30)
CREATE DOMAIN street_address CHAR(35);
```

Having defined these domains, you can use them much as you would the built-in data types. For example, you can use these definitions to define a table, as follows.

```
CREATE TABLE Customers (
   ID INT  DEFAULT AUTOINCREMENT  PRIMARY KEY,
   Name persons_name,
   Street street_address);
```

Example 2: Default values, check constraints, and identifiers

In the above example, the table's primary key is specified to be of type integer. Indeed, many of your tables may require similar identifiers. Instead of specifying that these are integers, it is much more convenient to create an identifier domain for use in these applications.

When you create a domain, you can specify a default value and provide check constraint to ensure that no inappropriate values are typed into any column of this type.

Integer values are commonly used as table identifiers. A good choice for unique identifiers is to use positive integers. Since such identifiers are likely to be used in many tables, you could define the following domain.

```
CREATE DOMAIN identifier UNSIGNED INT
DEFAULT AUTOINCREMENT;
```

Using this definition, you can rewrite the definition of the Customers table, shown above.

```
CREATE TABLE Customers2 (
   ID identifier PRIMARY KEY,
   Name persons_name,
   Street street_address
);
```

Drop domains

You can use either Sybase Central or a DROP DOMAIN statement to drop a domain.

Only a user with DBA authority or the user who created a domain can drop it. In addition, since a domain cannot be dropped if any variable or column in the database uses the domain, you need to first drop any columns or variables of that type before you can drop the domain.

To drop a domain (Sybase Central)

1. Use the SQL Anywhere 12 plug-in to connect to the database as a user with DBA authority.
2. In the left pane, double-click **Domains**.
3. In the right pane, right-click the domain and choose **Delete**.
4. Click **Yes**.

To drop a domain (SQL)

1. Connect to the database as a user with DBA authority.
2. Execute a DROP DOMAIN statement.

Example

The following statement drops the persons_name domain.

```
DROP DOMAIN persons_name;
```

For more information, see "DROP DOMAIN statement" [*SQL Anywhere Server - SQL Reference*].

Enforcing entity and referential integrity

The relational structure of the database enables the database server to identify information within the database, and ensures that all the rows in each table uphold the relationships between tables (described in the database schema).

Enforcing entity integrity

When a user inserts or updates a row, the database server ensures that the primary key for the table is still valid: that each row in the table is uniquely identified by the primary key.

Example 1

The Employees table in the SQL Anywhere sample database uses an employee ID as the primary key. When you add a new employee to the table, the database server checks that the new employee ID value is unique and is not NULL.

Example 2

The SalesOrderItems table in the SQL Anywhere sample database uses two columns to define a primary key.

This table holds information about items ordered. One column contains an ID specifying an order, but there may be several items on each order, so this column by itself cannot be a primary key. An additional LineID column identifies which line corresponds to the item. The columns ID and LineID, taken together, specify an item uniquely, and form the primary key.

If a client application breaches entity integrity

Entity integrity requires that each value of a primary key be unique within the table, and that no NULL values exist. If a client application attempts to insert or update a primary key value, providing values that are not unique would breach entity integrity. A breach in entity integrity prevents the new information from being added to the database, and instead sends the client application an error.

You must decide how to present an integrity breach to the user and enable them to take appropriate action. The appropriate action is usually as simple as asking the user to provide a different, unique value for the primary key.

Primary keys enforce entity integrity

Once you specify the primary key for each table, maintaining entity integrity requires no further action by either client application developers or by the database administrator.

The table owner defines the primary key for a table when they create it. If they modify the structure of a table at a later date, they can also redefine the primary key.

For more information about creating primary keys, see "Managing primary keys" on page 9.

For the detailed syntax of the CREATE TABLE statement, see "CREATE TABLE statement" [*SQL Anywhere Server - SQL Reference*].

For information about changing table structure, see "ALTER TABLE statement" [*SQL Anywhere Server - SQL Reference*].

Enforcing referential integrity

A foreign key (made up of a particular column or combination of columns) relates the information in one table (the **foreign** table) to information in another (**referenced** or **primary**) table. For the foreign key relationship to be valid, the entries in the foreign key must correspond to the primary key values of a row in the referenced table. Occasionally, some other unique column combination may be referenced instead of a primary key.

Example 1

The SQL Anywhere sample database contains an Employees table and a Departments table. The primary key for the Employees table is the employee ID, and the primary key for the Departments table is the department ID. In the Employees table, the department ID is called a **foreign key** for the Departments table because each department ID in the Employees table corresponds exactly to a department ID in the Departments table.

The foreign key relationship is a many-to-one relationship. Several entries in the Employees table have the same department ID entry, but the department ID is the primary key for the Departments table, and so is unique. If a foreign key could reference a column in the Departments table containing duplicate entries, or entries with a NULL value, there would be no way of knowing which row in the Departments table is the appropriate reference. This is a mandatory foreign key.

Example 2

Suppose the database also contained an office table listing office locations. The Employees table might have a foreign key for the office table that indicates which city the employee's office is in. The database designer can choose to leave an office location unassigned at the time the employee is hired, for example,

either because they haven't been assigned to an office yet, or because they don't work out of an office. In this case, the foreign key can allow NULL values, and is optional.

Foreign keys enforce referential integrity

Like primary keys, you use the CREATE TABLE or ALTER TABLE statements to create foreign keys. Once you create a foreign key, the column or columns in the key can contain only values that are present as primary key values in the table associated with the foreign key.

For more information about creating foreign keys, see "Managing primary keys" on page 9.

Losing referential integrity

Your database can lose referential integrity if someone:

- Updates or drops a primary key value. All the foreign keys referencing that primary key would become invalid.
- Adds a new row to the foreign table, and enters a value for the foreign key that has no corresponding primary key value. The database would become invalid.

SQL Anywhere provides protection against both types of integrity loss.

If a client application breaches referential integrity

If a client application updates or deletes a primary key value in a table, and if a foreign key references that primary key value elsewhere in the database, there is a danger of a breach of referential integrity.

Example

If the server allowed the primary key to be updated or dropped, and made no alteration to the foreign keys that referenced it, the foreign key reference would be invalid. Any attempt to use the foreign key reference, for example in a SELECT statement using a KEY JOIN clause, would fail, as no corresponding value in the referenced table exists.

While SQL Anywhere handles breaches of entity integrity in a generally straightforward fashion by simply refusing to enter the data and returning an error message, potential breaches of referential integrity become more complicated. You have several options (known as referential integrity actions) available to help you maintain referential integrity.

Referential integrity actions

Maintaining referential integrity when updating or deleting a referenced primary key can be as simple as disallowing the update or drop. Often, however, it is also possible to take a specific action on each foreign

key to maintain referential integrity. The CREATE TABLE and ALTER TABLE statements allow database administrators and table owners to specify what action to take on foreign keys that reference a modified primary key when a breach occurs.

You can specify each of the available referential integrity actions separately for updates and drops of the primary key:

- **RESTRICT** Generates an error and prevents the modification if an attempt to alter a referenced primary key value occurs. This is the default referential integrity action.
- **SET NULL** Sets all foreign keys that reference the modified primary key to NULL.
- **SET DEFAULT** Sets all foreign keys that reference the modified primary key to the default value for that column (as specified in the table definition).
- **CASCADE** When used with ON UPDATE, this action updates all foreign keys that reference the updated primary key to the new value. When used with ON DELETE, this action deletes all rows containing foreign keys that reference the deleted primary key.

System triggers implement referential integrity actions. The trigger, defined on the primary table, is executed using the permissions of the owner of the secondary table. This behavior means that cascaded operations can take place between tables with different owners, without additional permissions having to be granted.

Referential integrity checking

For foreign keys defined to RESTRICT operations that would violate referential integrity, default checks occur at the time a statement executes. If you specify a CHECK ON COMMIT clause, then the checks occur only when the transaction is committed.

Using a database option to control check time

Setting the wait_for_commit database option controls the behavior when a foreign key is defined to restrict operations that would violate referential integrity. The CHECK ON COMMIT clause can override this option.

With the default wait_for_commit set to Off, operations that would leave the database inconsistent cannot execute. For example, an attempt to DELETE a department that still has employees in it is not allowed. The following statement gives an error:

```
DELETE FROM Departments
WHERE DepartmentID = 200;
```

Setting wait_for_commit to On causes referential integrity to remain unchecked until a commit executes. If the database is in an inconsistent state, the database disallows the commit and reports an error. In this mode, a database user could drop a department with employees in it, however, the user cannot commit the change to the database until they:

- Delete or reassign the employees belonging to that department.
- Insert the DepartmentID row back into the Departments table.
- Roll back the transaction to undo the DELETE operation.

Integrity checks on INSERT

SQL Anywhere performs integrity checks when executing INSERT statements. For example, suppose you attempt to create a department, but supply a DepartmentID value that is already in use:

```
INSERT
INTO Departments ( DepartmentID, DepartmentName, DepartmentHeadID )
VALUES ( 200, 'Eastern Sales', 902 );
```

The INSERT is rejected because the primary key for the table would no longer be unique. Since the DepartmentID column is a primary key, duplicate values are not permitted.

Inserting values that violate relationships

The following statement inserts a new row in the SalesOrders table, but incorrectly supplies a SalesRepresentative ID that does not exist in the Employees table.

```
INSERT
INTO SalesOrders ( ID, CustomerID, OrderDate, SalesRepresentative)
VALUES ( 2700, 186, '2000-10-19', 284 );
```

There is a one-to-many relationship between the Employees table and the SalesOrders table, based on the SalesRepresentative column of the SalesOrders table and the EmployeeID column of the Employees table. Only after a record in the primary table (Employees) has been entered can a corresponding record in the foreign table (SalesOrders) be inserted.

Foreign keys

The primary key for the Employees table is the employee ID number. The sales rep ID number in the SalesRepresentative table is a foreign key for the Employees table, meaning that each sales rep number in the SalesOrders table must match the employee ID number for some employee in the Employees table.

When you try to add an order for sales rep 284 you get an error message similar to the following: `No primary key value for foreign key 'FK_SalesRepresentative_EmployeeID' in table 'SalesOrders'`.

There isn't an employee in the Employees table with that ID number. This prevents you from inserting orders without a valid sales representative ID.

See also

- “Relationships between tables” [*SQL Anywhere 12 - Introduction*]

Integrity checks on DELETE or UPDATE

Foreign key errors can also arise when performing update or delete operations. For example, suppose you try to remove the R&D department from the Departments table. The DepartmentID field, being the primary key of the Departments table, constitutes the ONE side of a one-to-many relationship (the DepartmentID field of the Employees table is the corresponding foreign key, and forms the MANY side). A record on the ONE side of a relationship may not be deleted until all corresponding records on the MANY side are deleted.

Referential integrity error on DELETE

Suppose you attempt to delete the R&D department (DepartmentID 100) in the Departments table. An error is reported indicating that there are other records in the database that reference the R&D department, and the delete operation is not performed. To remove the R&D department, you need to first get rid of all employees in that department, as follows:

```
DELETE
FROM Employees
WHERE DepartmentID = 100;
```

Now that you deleted all the employees that belong to the R&D department, you can now delete the R&D department:

```
DELETE
FROM Departments
WHERE DepartmentID = 100;
```

Cancel these changes to the database by entering a ROLLBACK statement:

```
ROLLBACK;
```

Referential integrity error on UPDATE

Now, suppose you try to change the DepartmentID field from the Employees table. The DepartmentID field, being the foreign key of the Employees table, constitutes the MANY side of a one-to-many relationship (the DepartmentID field of the Departments table is the corresponding primary key, and forms the ONE side). A record on the MANY side of a relationship may not be changed unless it corresponds to a record on the ONE side. That is, unless it has a primary key to reference.

For example, the following UPDATE statement causes an integrity error:

```
UPDATE Employees
SET DepartmentID = 600
WHERE DepartmentID = 100;
```

The error `no primary key value for foreign key 'FK_DepartmentID_DepartmentID' in table 'Employees'` is raised because there is no department with a DepartmentID of 600 in the Departments table.

To change the value of the DepartmentID field in the Employees table, it must correspond to an existing value in the Departments table. For example:

```
UPDATE Employees
SET DepartmentID = 300
WHERE DepartmentID = 100;
```


This statement can be executed because the DepartmentID of 300 corresponds to the existing Finance department.

Cancel these changes to the database by entering a ROLLBACK statement:

```
ROLLBACK;
```

Checking the integrity at commit time

In the previous examples, the integrity of the database was checked as each command was executed. Any operation that would result in an inconsistent database is not performed.

It is possible to configure the database so that the integrity is not checked until commit time using the wait_for_commit option. This is useful if you need to make changes that may cause temporary inconsistencies in the data while the changes are taking place. For example, suppose you want to delete the R&D department in the Employees and Departments tables. Since these tables reference each other, and since the deletions must be performed on one table at a time, there will be inconsistencies between the table during the deletion. In this case, the database cannot perform a COMMIT until the deletion finishes. Set the wait_for_commit option to On to allow data inconsistencies to exist up until a commit is performed. See "wait_for_commit option" [*SQL Anywhere Server - Database Administration*].

You can also define foreign keys in such a way that they are automatically modified to be consistent with changes made to the primary key. In the above example, if the foreign key from Employees to Departments was defined with ON DELETE CASCADE, then deleting the department ID would automatically delete the corresponding entries in the Employees table.

In the above cases, there is no way to have an inconsistent database committed as permanent. SQL Anywhere also supports alternative actions if changes would render the database inconsistent. See "Ensuring data integrity" on page 65.

Integrity rules in the system tables

All the information about database integrity checks and rules is held in system tables. Use their corresponding system views as follows to access this information:

System view	Description
SYS.SYSCONSTRAINT	Each row in the SYS.SYSCONSTRAINT system view describes a constraint in the database. The constraints currently supported include table and column checks, primary keys, foreign keys, and unique constraints. See "SYSCONSTRAINT system view" [*SQL Anywhere Server - SQL Reference*]. For table and column check constraints, the actual CHECK condition is contained in the SYS.ISYSCHECK system table. See "SYSCHECK system view" [*SQL Anywhere Server - SQL Reference*].

System view	Description
SYS.SYSCHECK	Each row in the SYS.SYSCHECK system view defines a check constraint listed in the SYS.SYSCONSTRAINT system view. See "SYSCHECK system view" [*SQL Anywhere Server - SQL Reference*].
SYS.SYSFKEY	Each row in the SYS.SYSFKEY system view describes a foreign key, including the match type defined for the key. See "SYSFKEY system view" [*SQL Anywhere Server - SQL Reference*].
SYS.SYSIDX	Each row in the SYS.SYSIDX system view defines an index in the database. See "SYSIDX system view" [*SQL Anywhere Server - SQL Reference*].
SYS.SYSTRIGGER	Each row in the SYS.SYSTRIGGER system view describes one trigger in the database, including triggers that are automatically created for foreign key constraints that have a referential triggered action (such as ON DELETE CASCADE). The referential_action column holds a single character indicating whether the action is cascade (C), delete (D), set null (N), or restrict (R). The event column holds a single character specifying the event that causes the action to occur: A=insert and delete, B=insert and update, C=update, D=delete, E=delete and update, I=insert, U=update, M=insert, delete and update. The trigger_time column shows whether the action occurs after (A) or before (B) the triggering event. See "SYSTRIGGER system view" [*SQL Anywhere Server - SQL Reference*].

Using transactions and isolation levels

To ensure data integrity, it is essential that you can identify states in which the information in your database is **consistent**. The concept of consistency is best illustrated through an example:

Consistency example

Suppose you use your database to handle financial accounts, and you want to transfer money from one client's account to another. The database is in a consistent state both before and after the money is transferred; but it is not in a consistent state after you have debited money from one account and before you have credited it to the second. During a transfer of money, the database is in a consistent state when the total amount of money in the clients' accounts is as it was before any money was transferred. When the money has been half transferred, the database is in an inconsistent state. Either both or neither of the debit and the credit must be processed.

Transactions are logical units of work

A **transaction** is a logical unit of work. Each transaction is a sequence of logically related commands that do one task and transform the database from one consistent state into another. The nature of a consistent state depends on your database.

The statements within a transaction are treated as an indivisible unit: either all are executed or none is executed. At the end of each transaction, you **commit** your changes to make them permanent. If for any reason some of the commands in the transaction do not process properly, then any intermediate changes are undone, or **rolled back**. Another way of saying this is that transactions are **atomic**.

Grouping statements into transactions is key both to protecting the consistency of your data (even in the event of media or system failure), and to managing concurrent database operations. Transactions may be safely interleaved and the completion of each transaction marks a point at which the information in the database is consistent. You should design each transaction to perform a task that changes your database from one consistent state to another.

In the event of a system failure or database crash during normal operation, SQL Anywhere performs automatic recovery of your data when the database is next started. The automatic recovery process recovers all completed transactions, and rolls back any transactions that were uncommitted when the failure occurred. The atomic character of transactions ensures that databases are recovered to a consistent state.

For more information about database backups and data recovery, see “Backup and data recovery” [*SQL Anywhere Server - Database Administration*].

For more information about concurrent database usage, see “Introduction to concurrency” on page 91.

Using transactions

SQL Anywhere expects you to group your commands into transactions. You commit a transaction to make changes to your database permanent. When you alter data, your alterations are recorded in the transaction log and are not made permanent until you enter the COMMIT command.

Transactions start with one of the following events:

- The first statement following a connection to a database.
- The first statement following the end of a transaction.

Transactions complete with one of the following events:

- A COMMIT statement makes the changes to the database permanent.
- A ROLLBACK statement undoes all the changes made by the transaction.
- A statement with a side effect of an automatic commit is executed: data definition commands, such as ALTER, CREATE, COMMENT, and DROP all have the side effect of an automatic commit.
- A disconnection from a database performs an implicit rollback.
- ODBC and JDBC have an autocommit setting that enforces a COMMIT after each statement. By default, ODBC and JDBC require autocommit to be on, and each statement is a single transaction. If you want to take advantage of transaction design possibilities, then you should turn autocommit off.

  For more information about autocommit, see "Setting autocommit or manual commit mode" [*SQL Anywhere Server - Programming*].
- Setting the chained database option to Off is similar to enforcing an autocommit after each statement. By default, connections that use jConnect or Open Client applications have chained set to Off.

  For more information, see "Setting autocommit or manual commit mode" [*SQL Anywhere Server - Programming*], and "chained option" [*SQL Anywhere Server - Database Administration*].

Options in Interactive SQL

Interactive SQL provides you with two options that let you control when and how transactions end:

- If you set the auto_commit option to On, Interactive SQL automatically commits your results following every successful statement and automatically performs a ROLLBACK after each failed statement. See "auto_commit option [Interactive SQL]" [*SQL Anywhere Server - Database Administration*].
- The setting of the option commit_on_exit controls what happens to uncommitted changes when you exit Interactive SQL. If this option is set to On (the default), Interactive SQL does a COMMIT; otherwise, it undoes your uncommitted changes with a ROLLBACK statement. See "commit_on_exit option [Interactive SQL]" [*SQL Anywhere Server - Database Administration*].

Using a data source in Interactive SQL
By default, ODBC operates in autocommit mode. Even if you have set the auto_commit option to Off in Interactive SQL, the ODBC setting overrides the Interactive SQL settings. You can change ODBC's setting using the SQL_ATTR_AUTOCOMMIT connection attribute. ODBC autocommit is independent of the chained option.

SQL Anywhere also supports Transact-SQL commands, such as BEGIN TRANSACTION, for compatibility with Adaptive Server Enterprise. For more information, see "Transact-SQL compatibility" on page 638.

Determining when a transaction began

The TransactionStartTime database property returns the time the database was first modified after a COMMIT or ROLLBACK. You can use this property to find the start time of the earliest transaction for all active connections. For example:

```
BEGIN
  DECLARE connid int;
  DECLARE earliest char(50);
  DECLARE connstart char(50);
  SET connid=next_connection(null);
  SET earliest = NULL;
  lp: LOOP
  IF connid IS NULL THEN LEAVE lp END IF;
    SET connstart = CONNECTION_PROPERTY('TransactionStartTime',connid);
    IF connstart <> '' THEN
      IF earliest IS NULL
      OR CAST(connstart AS TIMESTAMP) < CAST(earliest AS TIMESTAMP) THEN
        SET earliest = connstart;
      END IF;
    END IF;
    SET connid=next_connection(connid);
  END LOOP;
  SELECT earliest
END
```

Introduction to concurrency

Concurrency is the ability of the database server to process multiple transactions at the same time. Were it not for special mechanisms within the database server, concurrent transactions could interfere with each other to produce inconsistent and incorrect information.

Who needs to know about concurrency

Concurrency is a concern to all database administrators and developers. Even if you are working with a single-user database, you must be concerned with concurrency if you want to process requests from multiple applications or even from multiple connections from a single application. These applications and connections can interfere with each other in exactly the same way as multiple users in a network setting.

Transaction size affects concurrency

The way you group SQL statements into transactions can have significant effects on data integrity and on system performance. If you make a transaction too short and it does not contain an entire logical unit of work, then inconsistencies can be introduced into the database. If you write a transaction that is too long and contains several unrelated actions, then there is a greater chance that a ROLLBACK will unnecessarily undo work that could have been committed quite safely into the database.

If your transactions are long, they can lower concurrency by preventing other transactions from being processed concurrently.

There are many factors that determine the appropriate length of a transaction, depending on the type of application and the environment.

To learn more about running concurrent SQL Anywhere database servers, see "Using SQL Anywhere database servers" [*SQL Anywhere Server - Database Administration*].

Savepoints within transactions

You can identify important states within a transaction and return to them selectively using **savepoints** to separate groups of related statements.

A SAVEPOINT statement defines an intermediate point during a transaction. You can undo all changes after that point using a ROLLBACK TO SAVEPOINT statement. Once a RELEASE SAVEPOINT statement has been executed or the transaction has ended, you can no longer use the savepoint. Note that savepoints do not have an effect on COMMITs. When a COMMIT is executed, all changes within the transaction are made permanent in the database.

No locks are released by the RELEASE SAVEPOINT or ROLLBACK TO SAVEPOINT commands: locks are released only at the end of a transaction.

Naming and nesting savepoints

Using named, nested savepoints, you can have many active savepoints within a transaction. Changes between a SAVEPOINT and a RELEASE SAVEPOINT can be canceled by rolling back to a previous savepoint or rolling back the transaction itself. Changes within a transaction are not a permanent part of the database until the transaction is committed. All savepoints are released when a transaction ends.

Savepoints cannot be used in bulk operations mode. There is very little additional overhead in using savepoints.

Isolation levels and consistency

SQL Anywhere allows you to control the degree to which the operations in one transaction are visible to the operations in other concurrent transactions. You do so by setting a database option called the **isolation level**.

SQL Anywhere also allows you to control the isolation levels of individual tables in a query with corresponding table hints. See "FROM clause" [*SQL Anywhere Server - SQL Reference*].

SQL Anywhere provides the following isolation levels:

This isolation level...	Has these characteristics...
0—read uncommitted	• Read permitted on row with or without write lock • No read locks are applied • No guarantee that concurrent transaction will not modify row or roll back changes to row • Corresponds to table hints NOLOCK and READUNCOMMITTED • Allow dirty reads, non-repeatable reads, and phantom rows
1—read committed	• Read only permitted on row with no write lock • Read lock acquired and held for read on current row only, but released when cursor moves off the row • No guarantee that data will not change during transaction • Corresponds to table hint READCOMMITTED • Prevent dirty reads • Allow non-repeatable reads and phantom rows
2—repeatable read	• Read only permitted on row with no write lock • Read lock acquired as each row in the result set is read, and held until transaction ends • Corresponds to table hint REPEATABLEREAD • Prevent dirty reads and non-repeatable reads • Allow phantom rows
3—serializable	• Read only permitted on rows in result without write lock • Read locks acquired when cursor is opened and held until transaction ends • Corresponds to table hints HOLDLOCK and SERIALIZABLE • Prevent dirty reads, non-repeatable reads, and phantom rows
snapshot[1]	• No read locks are applied • Read permitted on any row • Database snapshot of committed data is taken when the first row is read or updated by the transaction

This isolation level...	Has these characteristics...
statement-snapshot[1]	• No read locks are applied • Read permitted on any row • Database snapshot of committed data is taken when the first row is read by the statement
readonly-statement-snapshot[1]	• No read locks are applied • Read permitted on any row • Database snapshot of committed data is taken when the first row is read by a read-only statement • Uses the isolation level (0, 1, 2, or 3) specified by the updatable_statement_isolation option for an updatable statement

[1] Snapshot isolation must be enabled for the database by setting the allow_snapshot_isolation option to On for the database. See "Enabling snapshot isolation" on page 97.

The default isolation level is 0, except for Open Client, jConnect, and TDS connections, which have a default isolation level of 1.

For information about MobiLink isolation levels, see "MobiLink isolation levels" [*MobiLink - Server Administration*].

Lock-based isolation levels prevent some or all interference. Level 3 provides the highest level of isolation. Lower levels allow more inconsistencies, but typically have better performance. Level 0 (read uncommitted) is the default setting.

The snapshot isolation levels prevent all interference between reads and writes. However, writes can still interfere with each other. Few inconsistencies are possible and contention performance is the same as isolation level 0. Performance not related to contention is worse because of the need to save and use row versions.

Notes
All isolation levels guarantee that each transaction executes completely or not at all, and no updates are lost.

The isolation is between transactions only: multiple cursors within the same transaction cannot interfere with each other.

Snapshot isolation

Blocks and deadlocks can occur when users are reading and writing the same data simultaneously. Snapshot isolation is designed to improve concurrency and consistency by maintaining different versions

of data. When you use snapshot isolation in a transaction, the database server returns a committed version of the data in response to any read requests. It does this without acquiring read locks, and prevents interference with users who are writing data.

A **snapshot** is a set of data that has been committed in the database. When using snapshot isolation, all queries within a transaction use the same set of data. No locks are acquired on database tables, which allows other transactions to access and modify the data without blocking. SQL Anywhere supports three snapshot isolation levels that let you control when a snapshot is taken:

- **snapshot** Use a snapshot of committed data from the time when the first row is read, inserted, updated, or deleted by the transaction.
- **statement-snapshot** Use a snapshot of committed data from the time when the first row is read by the statement. Each statement within the transaction sees a snapshot of data from a different time.
- **readonly-statement-snapshot** For read-only statements, use a snapshot of committed data from the time when the first row is read. Each read-only statement within the transaction sees a snapshot of data from a different time. For insert, update, and delete statements, use the isolation level specified by the updatable_statement_isolation option (can be one of 0 (the default), 1, 2, or 3).

You also have the option of specifying when the snapshot starts for a transaction by using the BEGIN SNAPSHOT statement. See "BEGIN SNAPSHOT statement" [*SQL Anywhere Server - SQL Reference*].

Snapshot isolation is often useful, such as:

- **Applications that perform many reads and few updates** Snapshot transactions acquire write locks only for statements that modify the database. If a transaction is performing mainly read operations, then the snapshot transaction does not acquire read locks that could interfere with other users' transactions.
- **Applications that perform long-running transactions while other users need to access data** Snapshot transactions do not acquire read locks, which makes data available to other users for reading and updating while the snapshot transaction takes place.
- **Applications that must read a consistent set of data from the database** Because a snapshot shows a committed set of data from a specific point in time, you can use snapshot isolation to see consistent data that does not change throughout the transaction, even if other users are making changes to the data while your transaction is running.

Snapshot isolation only affects base tables and global temporary tables that are shared by all users. A read operation on any other table type never sees an old version of the data, and never initiates a snapshot. The only time where an update to another table type initiates a snapshot is if the isolation_level option is set to snapshot, and the update initiates a transaction.

The following statements cannot be executed when there are cursors opened with the WITH HOLD clause that use either statement or transaction snapshots:

- ALTER INDEX
- ALTER TABLE
- CREATE INDEX
- DROP INDEX
- REFRESH MATERIALIZED VIEW
- REORGANIZE TABLE
- CREATE TEXT INDEX
- REFRESH TEXT INDEX

When opening cursors with the WITH HOLD clause, a snapshot of all rows committed at the snapshot start time is visible. Also visible are all modifications completed by the current connection since the start of the transaction within which the cursor was opened.

TRUNCATE TABLE is allowed only when a fast truncation is not performed because in this case, individual DELETEs are then recorded in the transaction log. See "TRUNCATE statement" [*SQL Anywhere Server - SQL Reference*].

In addition, if any of these statements are performed from a non-snapshot transaction, then snapshot transactions that are already in progress that subsequently try to use the table return an error indicating that the schema has changed.

Materialized view matching avoids using a view if it was refreshed after the start of the snapshot for a transaction.

Snapshot isolation levels are supported in all programming interfaces. You can set the isolation level using the SET OPTION statement. For information about using snapshot isolation, see:

- "isolation_level option" [*SQL Anywhere Server - Database Administration*]
- ADO and OLE DB: "Using transactions" [*SQL Anywhere Server - Programming*]
- ADO.NET: "IsolationLevel property" [*SQL Anywhere Server - Programming*]

Row versions

When snapshot isolation is enabled for a database, each time a row is updated, the database server adds a copy of the original row to the version stored in the temporary file. The original row version entries are stored until all the active snapshot transactions complete that might need access to the original row values. A transaction using snapshot isolation sees only committed values, so if the update to a row was not committed or rolled back before a snapshot transaction began, the snapshot transaction needs to be able to access the original row value. This allows transactions using snapshot isolation to view data without placing any locks on the underlying tables.

The VersionStorePages database property returns the number of pages in the temporary file that are currently being used for the version store. To obtain this value, execute the following query:

```
SELECT DB_PROPERTY ( 'VersionStorePages' );
```


Old row version entries are removed when they are no longer needed. Old versions of BLOBs are stored in the original table, not the temporary file, until they are no longer required, and index entries for old row versions are stored in the original index until they are not required.

You can retrieve the amount of free space in the temporary file using the sa_disk_free_space system procedure. See "sa_disk_free_space system procedure" [*SQL Anywhere Server - SQL Reference*].

If a trigger is fired that updates row values, the original values of those rows are also stored in the temporary file.

Designing your application to use shorter transactions and shorter snapshots reduces temporary file space requirements.

If you are concerned about temporary file growth, you can set up a GrowTemp system event that specifies the actions to take when the temporary file reaches a specific size. See "Understanding system events" [*SQL Anywhere Server - Database Administration*].

Understanding snapshot transactions

Snapshot transactions acquire write locks on updates, but read locks are never acquired for a transaction or statement that uses a snapshot. As a result, readers never block writers and writers never block readers, but writers can block writers if they attempt to update the same rows.

With snapshot isolation a transaction does not begin with a BEGIN TRANSACTION statement. Rather, it begins with the first read, insert, update, or delete within the transaction, depending on the snapshot isolation level being used for the transaction. The following example shows when a transaction begins for snapshot isolation:

```
SET OPTION PUBLIC.allow_snapshot_isolation = 'On';
         SET TEMPORARY OPTION isolation_level = 'snapshot';
   SELECT * FROM Products; --transaction begins and the statement only
                           --sees changes that are already committed
   INSERT INTO Products
         SELECT ID + 30, Name, Description,
         'Extra large', Color, 50, UnitPrice, NULL
         FROM Products
         WHERE Name = 'Tee Shirt';
COMMIT; --transaction ends
```

Enabling snapshot isolation

Snapshot isolation is enabled or disabled for a database using the allow_snapshot_isolation option. When the option is set to On, row versions are maintained in the temporary file, and connections are allowed to use any of the snapshot isolation levels. When this option is set to Off, any attempt to use snapshot isolation results in an error.

Enabling a database to use snapshot isolation can affect performance because copies of all modified rows must be maintained, regardless of the number of transactions that use snapshot isolation. See "Cursor sensitivity and isolation levels" [*SQL Anywhere Server - Programming*].

The following statement enables snapshot isolation for a database:

```
SET OPTION PUBLIC.allow_snapshot_isolation = 'On';
```

The setting of the allow_snapshot_isolation option can be changed, even when there are users connected to the database. When you change the setting of this option from Off to On, all current transactions must complete before new transactions can use snapshot isolation. When you change the setting of this option from On to Off, all outstanding transactions using snapshot isolation must complete before the database server stops maintaining row version information.

You can view the current snapshot isolation setting for a database by querying the value of the SnapshotIsolationState database property:

```
SELECT DB_PROPERTY ( 'SnapshotIsolationState' );
```

The SnapshotIsolationState property has one of the following values:

- **On** Snapshot isolation is enabled for the database.
- **Off** Snapshot isolation is disabled for the database.
- **in_transition_to_on** Snapshot isolation will be enabled once the current transactions complete.
- **in_transition_to_off** Snapshot isolation will be disabled once the current transactions complete.

When snapshot isolation is enabled for a database, row versions must be maintained for a transaction until the transaction commits or rolls back, even if snapshots are not being used. Therefore, it is best to set the allow_snapshot_isolation option to Off if snapshot isolation is never used.

Snapshot isolation example

The following example uses two connections to the SQL Anywhere sample database to illustrate how snapshot isolation can be used to maintain consistency without blocking.

To use snapshot isolation

1. Execute the following command to create an Interactive SQL connection (Connection1), to the SQL Anywhere sample database:

   ```
   dbisql -c "DSN=SQL Anywhere 12 Demo;ConnectionName=Connection1"
   ```

2. Execute the following command to create an Interactive SQL connection (Connection2) to the SQL Anywhere sample database:

   ```
   dbisql -c "DSN=SQL Anywhere 12 Demo;ConnectionName=Connection2"
   ```

3. In Connection1, execute the following command to set the isolation level to 1 (read committed), which acquires and holds a read lock on the current row.

   ```
   SET OPTION isolation_level = 1;
   ```

4. In Connection1, execute the following command:

   ```
   SELECT * FROM Products;
   ```


ID	Name	Description	Size	Color	Quantity	...
300	Tee Shirt	Tank Top	Small	White	28	...
301	Tee Shirt	V-neck	Medium	Orange	54	...
302	Tee Shirt	Crew Neck	One size fits all	Black	75	...
400	Baseball Cap	Cotton Cap	One size fits all	Black	112	...
...	...	...	...	...	...	...

5. In Connection2, execute the following command:

```
UPDATE Products
SET Name = 'New Tee Shirt'
WHERE ID = 302;
```

6. In Connection1, execute the SELECT statement again:

```
SELECT * FROM Products;
```

   The SELECT statement is blocked and cannot proceed because the UPDATE statement in Connection2 has not been committed or rolled back. The SELECT statement must wait until the transaction in Connection2 is complete before it can proceed. This ensures that the SELECT statement does not read uncommitted data into its result.

7. In Connection2, execute the following command:

```
ROLLBACK;
```

   The transaction in Connection2 completes, and the SELECT statement in Connection1 proceeds.

8. Using the statement snapshot isolation level achieves the same concurrency as isolation level 1, but without blocking.

   In Connection1, execute the following command to allow snapshot isolation:

```
SET OPTION PUBLIC.allow_snapshot_isolation = 'On';
```

9. In Connection 1, execute the following command to change the isolation level to statement snapshot:

```
SET TEMPORARY OPTION isolation_level = 'statement-snapshot';
```

10. In Connection1, execute the following statement:

```
SELECT * FROM Products;
```

11. In Connection2, execute the following statement:

```
UPDATE Products
SET Name = 'New Tee Shirt'
WHERE ID = 302;
```

12. In Connection1, issue the SELECT statement again:

    ```
    SELECT * FROM Products;
    ```

    The SELECT statement executes without being blocked, but does not include the data from the UPDATE statement executed by Connection2.

13. In Connection2, finish the transaction by executing the following command:

    ```
    COMMIT;
    ```

14. In Connection1, finish the transaction (the query against the Products table), and then execute the SELECT statement again to view the updated data:

    ```
    COMMIT;
    SELECT * FROM Products;
    ```

ID	Name	Descrip-tion	Size	Color	Quantity	...
300	Tee Shirt	Tank Top	Small	White	28	...
301	Tee Shirt	V-neck	Medium	Orange	54	...
302	New Tee Shirt	Crew Neck	One size fits all	Black	75	...
400	Baseball Cap	Cotton Cap	One size fits all	Black	112	...
...	...	...	...	...	...	...

15. Undo the changes to the SQL Anywhere sample database by executing the following statement:

    ```
    UPDATE Products
    SET Name = 'Tee Shirt'
    WHERE id = 302;
    COMMIT;
    ```

For more examples about using snapshot isolation, see:

- “Using snapshot isolation to avoid dirty reads” on page 132
- “Using snapshot isolation to avoid non-repeatable reads” on page 139
- “Using snapshot isolation to avoid phantom rows” on page 143

Update conflicts and snapshot isolation

With snapshot isolation, an update conflict can occur when a transaction sees an old version of a row and tries to update or delete it. When this happens, the server gives an error when it detects the conflict. For a committed change, this is when the update or delete is attempted. For an uncommitted change, the update or delete blocks and the server returns the error when the change commits.

Update conflicts cannot occur when using readonly-statement-snapshot because updatable statements run at a non-snapshot isolation, and always see the most recent version of the database. Therefore, the readonly-statement-snapshot isolation level has many of the benefits of snapshot isolation, without requiring large changes to an application originally designed to run at another isolation level. When using the readonly-statement-snapshot isolation level:

- Read locks are never acquired for read-only statements
- Read-only statements always see a committed state of the database

Typical types of inconsistency

There are three typical types of inconsistency that can occur during the execution of concurrent transactions. This list is not exhaustive as other types of inconsistencies can also occur. These three types are mentioned in the ISO SQL/2008 standard and are important because behavior at lower isolation levels is used to define them.

- **Dirty read** Transaction A modifies a row, but does not commit or roll back the change. Transaction B reads the modified row. Transaction A then either further changes the row before performing a COMMIT, or rolls back its modification. In either case, transaction B has seen the row in a state which was never committed.

  For more information about dirty reads, see "Tutorial: Dirty reads" on page 129.

- **Non-repeatable read** Transaction A reads a row. Transaction B then modifies or deletes the row and performs a COMMIT. If transaction A then attempts to read the same row again, the row will have been changed or deleted.

  For more information about non-repeatable reads, see "Tutorial: Non-repeatable reads" on page 134.

- **Phantom row** Transaction A reads a set of rows that satisfy some condition. Transaction B then executes an INSERT or an UPDATE on a row which did not previously meet A's condition. Transaction B commits these changes. These newly committed rows now satisfy Transaction A's condition. If Transaction A then repeats the read, it obtains the updated set of rows.

  For more information about phantom rows, see "Tutorial: Phantom rows" on page 140.

Isolation levels and dirty reads, non-repeatable reads, and phantom rows

SQL Anywhere allows dirty reads, non-repeatable reads, and phantom rows, depending on the isolation level that is used. An X in the following table indicates that the behavior is allowed for that isolation level.

Isolation level	Dirty reads	Non-repeatable reads	Phantom rows
0-read uncommitted	X	X	X
readonly-statement-snapshot	X[1]	X[2]	X[3]

Isolation level	Dirty reads	Non-repeatable reads	Phantom rows
1-read committed		X	X
statement-snapshot		X[2]	X[3]
2-repeatable read			X
3-serializable			
snapshot			

[1] Dirty reads can occur for updatable statements within a transaction if the isolation level specified by the updatable_statement_isolation option does not prevent them from occurring.

[2] Non-repeatable reads can occur for statements within a transaction if the isolation level specified by the updatable_statement_isolation option does not prevent them from occurring. Non-repeatable reads can occur because each statement starts a new snapshot, so one statement may see changes that another statement does not see.

[3] Phantom rows can occur for statements within a transaction if the isolation level specified by the updatable_statement_isolation option does not prevent them from occurring. Phantom rows can occur because each statement starts a new snapshot, so one statement may see changes that another statement does not see.

This table demonstrates two points:

- Each isolation level eliminates one of the three typical types of inconsistencies.
- Each level eliminates the types of inconsistencies eliminated at all lower levels.
- For statement snapshot isolation levels, non-repeatable reads and phantom rows can occur within a transaction, but not within a single statement in a transaction.

The isolation levels have different names under ODBC. These names are based on the names of the inconsistencies that they prevent. See "The ValuePtr parameter" on page 104.

Cursor instability

Another significant inconsistency is **cursor instability**. When this inconsistency is present, a transaction can modify a row that is being referenced by another transaction's cursor. Cursor stability ensures that applications using cursors do not introduce inconsistencies into the data in the database.

Example

Transaction A reads a row using a cursor. Transaction B modifies that row and commits. Not realizing that the row has been modified, Transaction A modifies it.

Eliminating cursor instability

SQL Anywhere provides **cursor stability** at isolation levels 1, 2, and 3. Cursor stability ensures that no other transactions can modify information that is contained in the present row of your cursor. The information in a row of a cursor may be the copy of information contained in a particular table or may be a combination of data from different rows of multiple tables. More than one table will likely be involved whenever you use a join or sub-selection within a SELECT statement.

For information about programming SQL procedures and cursors, see "Using procedures, triggers, and batches" on page 793.

Cursors are used only when you are using SQL Anywhere through another application. For more information, see "Using SQL in applications" [*SQL Anywhere Server - Programming*].

A related but distinct concern for applications using cursors is whether changes to underlying data are visible to the application. You can control the changes that are visible to applications by specifying the sensitivity of the cursor.

For more information about cursor sensitivity, see "SQL Anywhere cursors" [*SQL Anywhere Server - Programming*].

Set the isolation level

Each connection to the database has its own isolation level. In addition, the database can store a default isolation level for each user or group. The PUBLIC setting of the isolation_level database option enables you to set a single default isolation level for the entire database group.

You can also set the isolation level using table hints, but this is an advanced feature that should be used only when needed. For more information, see "WITH table-hint clause, FROM clause" [*SQL Anywhere Server - SQL Reference*].

You can change the isolation of your connection and the default level associated with your user ID by using the SET OPTION command. If you have permission, you can also change the isolation level for other users or groups.

If you want to use snapshot isolation, you must first enable snapshot isolation for the database.

For information about enabling and setting snapshot isolation levels, see "Enabling snapshot isolation" on page 97.

To set the isolation level for the current user

- Execute the SET OPTION statement. For example, the following statement sets the isolation level to 3 for the current user:

```
SET OPTION isolation_level = 3;
```

To set the isolation level for a user or group

1. Connect to the database as a user with DBA authority.
2. Execute the SET OPTION statement, adding the name of the group and a period before isolation_level. For example, the following command sets the default isolation for the PUBLIC group to 3.

   ```
   SET OPTION PUBLIC.isolation_level = 3;
   ```

To set the isolation level just the current connection

- Execute the SET OPTION statement using the TEMPORARY keyword. For example, the following statement sets the isolation level to 3 for the duration of the current connection:

  ```
  SET TEMPORARY OPTION isolation_level = 3;
  ```

Default isolation level

When you connect to a database, the database server determines your initial isolation level as follows:

1. A default isolation level may be set for each user and group. If a level is stored in the database for your user ID, then the database server uses it.
2. If not, the database server checks the groups to which you belong until it finds a level. All users are members of the special group PUBLIC. If it finds no other setting first, then SQL Anywhere uses the level assigned to that group.

For more information about users and groups, see "Managing user IDs, authorities, and permissions" [*SQL Anywhere Server - Database Administration*].

For more information about the SET OPTION statement syntax, see "SET OPTION statement" [*SQL Anywhere Server - SQL Reference*].

You may want to change the isolation level mid-transaction if, for example, just one or more tables requires serialized access. For information about changing the isolation level within a transaction, see "Changing isolation levels within a transaction" on page 106.

Setting the isolation level from an ODBC-enabled application

ODBC applications call SQLSetConnectAttr with Attribute set to SQL_ATTR_TXN_ISOLATION and ValuePtr set according to the corresponding isolation level:

The ValuePtr parameter

ValuePtr	Isolation level
SQL_TXN_READ_UNCOMMITTED	0
SQL_TXN_READ_COMMITTED	1
SQL_TXN_REPEATABLE_READ	2
SQL_TXN_SERIALIZABLE	3
SA_SQL_TXN_SNAPSHOT	snapshot
SA_SQL_TXN_STATEMENT_SNAPSHOT	statement-snapshot
SA_SQL_TXN_READONLY_STATEMENT_SNAPSHOT	readonly-statement-snapshot

Changing an isolation level via ODBC

You can change the isolation level of your connection via ODBC using the function SQLSetConnectOption in the library *ODBC32.dll*.

The SQLSetConnectOption function takes three parameters: the value of the ODBC connection handle, the fact that you want to set the isolation level, and the value corresponding to the isolation level. These values appear in the table below.

String	Value
SQL_TXN_ISOLATION	108
SQL_TXN_READ_UNCOMMITTED	1
SQL_TXN_READ_COMMITTED	2
SQL_TXN_REPEATABLE_READ	4
SQL_TXN_SERIALIZABLE	8
SA_SQL_TXN_SNAPSHOT	32
SA_SQL_TXN_STATEMENT_SNAPSHOT	64
SA_SQL_TXN_READONLY_STATEMENT_SNAPSHOT	128

Do not use the SET OPTION statement to change an isolation level from within an ODBC application. Since the ODBC driver does not parse the statements, execution of any statement in ODBC is not recognized by the ODBC driver. This could lead to unexpected locking behavior.

Example

The following function call sets the isolation level of the connection MyConnection to isolation level 2:

```
SQLSetConnectOption( MyConnection.hDbc,
                     SQL_TXN_ISOLATION,
                     SQL_TXN_REPEATABLE_READ )
```

ODBC uses the isolation feature to support assorted database lock options. For example, in PowerBuilder you can use the Lock attribute of the transaction object to set the isolation level when you connect to the database. The Lock attribute is a string, and is set as follows:

```
SQLCA.lock = "RU"
```

The Lock option is honored only at the moment the CONNECT occurs. Changes to the Lock attribute after the CONNECT have no effect on the connection.

Changing isolation levels within a transaction

Different isolation levels may be suitable for different parts of a single transaction. SQL Anywhere allows you to change the isolation level of your database in the middle of a transaction.

When you change the isolation_level option in the middle of a transaction, the new setting affects only the following:

- Any cursors opened after the change
- Any statements executed after the change

You may want to change the isolation level during a transaction to control the number of locks your transaction places. You may find a transaction needs to read a large table, but perform detailed work with only a few of the rows. If an inconsistency would not seriously affect your transaction, set the isolation to a low level while you scan the large table to avoid delaying the work of others.

You may also want to change the isolation level mid-transaction if, for example, just one table or group of tables requires serialized access.

For an example in which the isolation level is changed in the middle of a transaction, see "Tutorial: Phantom rows" on page 140.

Note
You can also set the isolation level (levels 0-3 only) using table hints, but this is an advanced feature that you should use only when needed. For more information, see the WITH *table-hint* section in "FROM clause" [*SQL Anywhere Server - SQL Reference*].

Changing isolation levels when using snapshot isolation

When using snapshot isolation, you can change the isolation level within a transaction. This can be done by changing the setting of the isolation_level option or by using table hints that affect the isolation level in a query. You can use statement-snapshot, readonly-statement-snapshot, and isolation levels 0-3 at any time. However, you cannot use the snapshot isolation level in a transaction if it began at an isolation level other than snapshot. A transaction is initiated by an update and continues until the next COMMIT or ROLLBACK. If the first update takes place at some isolation level other than snapshot, then any

statement that tries to use the snapshot isolation level before the transaction commits or rolls back returns error -1065 (SQLE_NON_SNAPSHOT_TRANSACTION). For example:

```
SET OPTION PUBLIC.allow_snapshot_isolation = 'On';

BEGIN TRANSACTION
    SET OPTION isolation_level = 3;
    INSERT INTO Departments
        ( DepartmentID, DepartmentName, DepartmentHeadID )
        VALUES( 700, 'Foreign Sales', 129 );
    SET TEMPORARY OPTION isolation_level = 'snapshot';
    SELECT * FROM Departments;
```

Viewing the isolation level

You can inspect the isolation level of the current connection using the CONNECTION_PROPERTY function.

To view the isolation level for the current connection

- Execute the following statement:

```
SELECT CONNECTION_PROPERTY('isolation_level');
```

Transaction blocking and deadlock

When a transaction is being executed, the database server places locks on rows to prevent other transactions from interfering with the affected rows. **Locks** control the amount and types of interference permitted.

SQL Anywhere uses **transaction blocking** to allow transactions to execute concurrently without interference, or with limited interference. Any transaction can acquire a lock to prevent other concurrent transactions from modifying or even accessing a particular row. This transaction blocking scheme always stops some types of interference. For example, a transaction that is updating a particular row of a table always acquires a lock on that row to ensure that no other transaction can update or delete the same row at the same time.

See also

- "blocking_others_timeout option" [*SQL Anywhere Server - Database Administration*]
- "blocking_timeout option" [*SQL Anywhere Server - Database Administration*]
- "blocking option" [*SQL Anywhere Server - Database Administration*]

Transaction blocking

When a transaction attempts to perform an operation, but is forbidden by a lock held by another transaction, a conflict arises and the progress of the transaction attempting to perform the operation is impeded.

Sometimes a set of transactions arrive at a state where none of them can proceed. For more information, see "Deadlock" on page 108.

See also

- "blocking_others_timeout option" [*SQL Anywhere Server - Database Administration*]
- "blocking_timeout option" [*SQL Anywhere Server - Database Administration*]
- "blocking option" [*SQL Anywhere Server - Database Administration*]

The blocking option

If two transactions have each acquired a read lock on a single row, the behavior when one of them attempts to modify that row depends on the setting of the blocking option. To modify the row, that transaction must block the other, yet it cannot do so while the other transaction has it blocked.

- If the blocking is option is set to On (the default), then the transaction that attempts to write waits until the other transaction releases its read lock. At that time, the write goes through.

- If the blocking option has been set to Off, then the statement that attempts to write receives an error.

When the blocking option is set to Off, the statement terminates instead of waiting and any partial changes it has made are rolled back. In this event, try executing the transaction again, later.

Blocking is more likely to occur at higher isolation levels because more locking and more checking is done. Higher isolation levels usually provide less concurrency. How much less depends on the individual natures of the concurrent transactions.

For more information about the blocking option, see "blocking option" [*SQL Anywhere Server - Database Administration*].

See also

- "blocking_others_timeout option" [*SQL Anywhere Server - Database Administration*]
- "blocking_timeout option" [*SQL Anywhere Server - Database Administration*]

Deadlock

Transaction blocking can lead to **deadlock**, a situation in which a set of transactions arrive at a state where none of them can proceed.

Reasons for deadlocks

A deadlock can arise for two reasons:

- **A cyclical blocking conflict** Transaction A is blocked on transaction B, and transaction B is blocked on transaction A. More time will not solve the problem, and one of the transactions must be canceled, allowing the other to proceed. The same situation can arise with more than two transactions blocked in a cycle.

  To eliminate a transactional deadlock, SQL Anywhere selects a connection from those involved in the deadlock, rolls back the changes for the transaction that is active on that connection and returns an error. SQL Anywhere selects the connection to roll back by using an internal heuristic that prefers the connection with the smallest blocking wait time left as determined by the blocking_timeout option. If all connections are set to wait forever, then the connection that caused the server to detect a deadlock is selected as the victim connection. See "blocking_timeout option" [*SQL Anywhere Server - Database Administration*].

- **All workers are blocked** When a transaction becomes blocked, its worker is not relinquished. For example, if the database server is configured with three workers and transactions A, B, and C are blocked on transaction D which is not currently executing a request, then a deadlock situation has arisen since there are no available workers. This situation is called thread deadlock.

  Suppose that the database server has *n* workers. Thread deadlock occurs when *n*-1 workers are blocked, and the last worker is about to block. The database server's kernel cannot permit this last worker to block, since doing so would result in all workers being blocked, and the database server would hang. Instead, the database server ends the task that is about to block the last worker, rolls back the changes for the transaction active on that connection, and returns an error (SQLCODE -307, SQLSTATE 40W06).

  Database servers with tens or hundreds of connections may experience thread deadlock in cases where there are many long-running requests either because of the size of the database or because of blocking. In this case, increasing the database server's multiprogramming level may be an appropriate solution. The design of your application may also cause thread deadlock because of excessive or unintentional contention. In these cases, scaling the application to larger data sets can make the problem worse, and increasing the database server's multiprogramming level may not solve the problem. See "Configuring the database server's multiprogramming level" [*SQL Anywhere Server - Database Administration*].

  The number of database threads that the server uses depends on the individual database's setting. See "Controlling threading behavior" [*SQL Anywhere Server - Database Administration*].

See also

- "log_deadlocks option" [*SQL Anywhere Server - Database Administration*]
- "sa_report_deadlocks system procedure" [*SQL Anywhere Server - SQL Reference*]

Determining who is blocked

You can use the sa_conn_info system procedure to determine which connections are blocked on which other connections. This procedure returns a result set consisting of a row for each connection. One column of the result set lists whether the connection is blocked, and if so which other connection it is blocked on.

For more information, see "sa_conn_info system procedure" [*SQL Anywhere Server - SQL Reference*].

You can also use the Deadlock event to take action when a deadlock occurs. The event handler can use the sa_report_deadlocks procedure to obtain information about the conditions that led to the deadlock. To retrieve more details about the deadlock from the database server, use the log_deadlocks option and enable the RememberLastStatement feature.

The follow procedure shows you how to set up a table and system event that can be used to obtain information about deadlocks when they occur. If you find that your application has frequent deadlocks, you can use application profiling to help diagnose the cause of the deadlocks. See "Tutorial: Diagnosing deadlocks" on page 234.

To take action when a deadlock occurs

1. Create a table to store the data returned from the sa_report_deadlocks system procedure:

```
CREATE TABLE DeadlockDetails(
  deadlockId INT PRIMARY KEY DEFAULT AUTOINCREMENT,
  snapshotId BIGINT,
  snapshotAt TIMESTAMP,
  waiter INTEGER,
  who VARCHAR(128),
  what LONG VARCHAR,
  object_id UNSIGNED BIGINT,
  record_id BIGINT,
  owner INTEGER,
  is_victim BIT,
  rollback_operation_count UNSIGNED INTEGER );
```

2. Create an event that fires when a deadlock occurs.

   This event copies the results of the sa_report_deadlocks system procedure into a table and notifies an administrator about the deadlock:

```
CREATE EVENT DeadlockNotification
TYPE Deadlock
HANDLER
BEGIN
 INSERT INTO DeadlockDetails WITH AUTO NAME
 SELECT snapshotId, snapshotAt, waiter, who, what, object_id, record_id,
        owner, is_victim, rollback_operation_count
    FROM sa_report_deadlocks ();
 COMMIT;
 CALL xp_startmail ( mail_user ='George Smith',
                     mail_password ='mypwd' );
 CALL xp_sendmail( recipient='DBAdmin',
                   subject='Deadlock details added to the DeadlockDetails
table.' );
 CALL xp_stopmail ( );
END;
```

3. Set the log_deadlocks option to On:

```
SET OPTION PUBLIC.log_deadlocks = 'On';
```

4. Enable logging of the most-recently executed statement:

```
CALL sa_server_option( 'RememberLastStatement', 'YES' );
```


See also

- "log_deadlocks option" [*SQL Anywhere Server - Database Administration*]
- "sa_report_deadlocks system procedure" [*SQL Anywhere Server - SQL Reference*]
- "sa_server_option system procedure" [*SQL Anywhere Server - SQL Reference*]
- "CREATE EVENT statement" [*SQL Anywhere Server - SQL Reference*]

Viewing deadlocks from Sybase Central

When you are connected to a database in Sybase Central, you can see a diagram of any deadlocks that have occurred in the database since the log_deadlocks option was set to On. Deadlock information is recorded in an internal buffer.

To use Sybase Central deadlock reporting

1. Select the database in the left pane of Sybase Central, and then choose **File** » **Options**.
2. Turn on the log_deadlocks option.
   a. In the **Options** list, select **log_deadlocks**.
   b. In the **Value** field, type **On**.
   c. Click **Set Permanent Now**.
   d. Click **Close**.

   For more information, see "log_deadlocks option" [*SQL Anywhere Server - Database Administration*].
3. In the right pane, click the **Deadlocks** tab.

   A deadlock diagram appears if there are deadlocks in the database. Each node in the deadlock diagram represents a connection and gives details about which connection was deadlocked, the user name, and the SQL statement the connection was trying to execute when the deadlock occurred. There are two types of deadlocks: connection deadlocks and thread deadlocks. Connection deadlocks are characterized by a circular dependency for the nodes. A thread deadlock is indicated by nodes that are not connected in a circular dependency, and the number of nodes is equal to the thread limit on the database plus one.

How locking works

A lock is 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. Locks improve the consistency of query result by protecting information that is in the process of being updated.

The database server places these locks automatically and needs no explicit instruction. It holds all the locks acquired by a transaction until the transaction is completed, for example by either a COMMIT or

ROLLBACK statement, with a single exception. For information about this exception, see "Lock duration" on page 125.

The transaction that has access to the row is said to hold the lock. Depending on the type of lock, other transactions may have limited access to the locked row, or none at all.

Objects that can be locked

To ensure database consistency and to support appropriate isolation levels between transactions, SQL Anywhere uses the following types of locks:

- **Schema locks** These locks control the ability to make schema changes. For example, a transaction can lock the schema of a table, preventing other transactions from modifying the table's structure.
- **Row locks** These locks are used to ensure consistency between concurrent transactions at a row level. For example, a transaction can lock a particular row to prevent another transaction from changing it, and a transaction must place a write lock on a row if it intends to modify the row. To maximize concurrency, the key and non-key portions of the row can be locked independently. Updating non-key columns of a row does not interfere with the insertion and deletion of foreign rows referencing that row.
- **Table locks** These locks are used to ensure consistency between concurrent transactions at a table level. For example, a transaction that is changing the structure of a table by inserting a new column can lock a table so that other transactions are not affected by the schema change. In such a case, it is essential to limit the access of other transactions to prevent errors.
- **Position locks** These locks are used to ensure consistency within a sequential or indexed scan of a table. Transactions typically scan rows using the ordering imposed by an index, or scan rows sequentially. In either case, a lock can be placed on the scan position. For example, placing a lock in an index can prevent another transaction from inserting a row with a specific value or range of values.

Schema locks provide a mechanism to prevent schema changes from inadvertently affecting executing transactions. Row locks, table locks, and position locks each have a separate purpose, but they do interact. Each lock type prevents a particular set of inconsistencies. Depending on the isolation level you select, the database server uses some or all these lock types to maintain the degree of consistency you require.

Lock duration

The different classes of locks can be held for different durations:

- **Position** Short-term locks, such as read locks on specific rows used to implement cursor stability at isolation level 1.
- **Transaction** Row, table, and position locks that are held until the end of a transaction.
- **Connection** Schema locks that are held beyond the end of a transaction, such as schema locks created when WITH HOLD cursors are used.

Obtaining information about locks

To diagnose a locking issue in the database it may be useful to know the contents of the rows that are locked. You can view the locks currently held in the database using either the sa_locks system procedure, or using the **Locks** tab in Sybase Central. Both methods provide the information you need, including the connection holding the lock, lock duration, and lock type.

Note
Due to the transient nature of locks in the database, the rows visible in Sybase Central, or returned by the sa_locks system procedure, may no longer exist by the time a query completes.

Viewing locks using Sybase Central

You can view locks in Sybase Central. Select the database in the left pane and then click the **Locks** tab in the right pane. For each lock, this tab shows you the connection ID, user ID, table name, lock type, and lock name.

Viewing locks using the sa_locks system procedure

The result set of the sa_locks system procedure contains the row_identifier column that allows you to identify the row in a table the lock refers to. To determine the actual values stored in the locked row, you can join the results of the sa_locks system procedure to a particular table, using the rowID of the table in the join predicate. For example:

```
SELECT S.conn_id, S.user_id, S.lock_class, S.lock_type, E.*
  FROM sa_locks() S JOIN Employees E WITH( NOLOCK )
     ON RowId(E) = S.row_identifier
  WHERE S.table_name = 'Employees';
```

Note
It may not be necessary to specify the NOLOCK table hint; however, if the query is issued at isolation levels other than 0, the query may block until the locks are released, which will reduce the usefulness of this method of checking.

See also

For more information about the sa_locks system procedure, see "sa_locks system procedure" [*SQL Anywhere Server - SQL Reference*].

For information about the NOLOCK table hint, see "FROM clause" [*SQL Anywhere Server - SQL Reference*].

For more information about the ROWID function, see "ROWID function [Miscellaneous]" [*SQL Anywhere Server - SQL Reference*].

Schema locks

Schema locks are used to serialize changes to a database schema, and to ensure that transactions using a table are not affected by schema changes initiated by other connections. For example, a schema lock

prevents an ALTER TABLE statement from dropping a column from a table when that table is being read by an open cursor on another connection.

There are two classes of schema locks:

- **Shared locks** The table schema is locked in shared (read) mode.
- **Exclusive locks** The table schema is locked for the exclusive use of a single connection.

A shared schema lock is acquired when a transaction refers directly or indirectly to a table in the database. Shared schema locks do not conflict with each other; any number of transactions can acquire shared locks on the same table at the same time. The shared schema lock is held until the transaction completes via a COMMIT or ROLLBACK.

Any connection holding a shared schema lock is allowed to change table data, providing the change does not conflict with other connections.

An exclusive schema lock is acquired when the schema of a table is modified, usually through the use of a DDL statement. The ALTER TABLE statement is one example of a DDL statement that acquires an exclusive lock on the table before modifying it. Only one connection can acquire an exclusive schema lock on a table at any time—all other attempts to lock the table's schema (shared or exclusive) will either block or fail with an error. This means that a connection executing at isolation level 0, which is the least restrictive isolation level, will be blocked from reading rows from a table whose schema has been locked in exclusive mode.

Only the connection holding the exclusive table schema lock can change the table data.

Row locks

Row locks are used to prevent lost updates and other types of transaction inconsistencies by ensuring that any row modified by a transaction cannot be modified by another transaction until the first transaction completes, either by committing the changes by issuing an implicit or explicit COMMIT statement, or by aborting the changes via a ROLLBACK statement.

There are three classes of row locks: read (shared) locks, write (exclusive) locks, and intent locks. The database server acquires these locks automatically for each transaction.

Read locks

When a transaction reads a row, the isolation level of the transaction determines if a read lock is acquired. Once a row is read locked, no other transaction can obtain a write lock on it. Acquiring a read lock ensures that a different transaction does not modify or delete a row while it is being read. Any number of transactions can acquire read locks on any row at the same time, so read locks are sometimes referred to as shared locks, or non-exclusive locks.

Read locks can be held for different durations. At isolation levels 2 and 3, any read locks acquired by a transaction are held until the transaction completes through a COMMIT or a ROLLBACK. These read locks are called long-term read locks.

For transactions executing at isolation level 1, the database server acquires a short-term read lock on the row upon which a cursor is positioned. As the application scrolls through the cursor, the short-term read lock on the previously-positioned row is released, and a new short-term read lock is acquired on the subsequent row. This technique is called **cursor stability**. Because the application holds a read lock on the current row, another transaction cannot make changes to the row until the application moves off the row. Note that more than one lock can be acquired if the cursor is over a query involving multiple tables. Short-term read locks are acquired only when the position within a cursor must be maintained across requests (ordinarily, these would be FETCH statements issued by the application). For example, short-term read locks are not acquired when processing a SELECT COUNT(*) query since a cursor opened over this statement will never be positioned on a particular base table row. In this case, the database server only needs to guarantee read committed semantics, that is, that the rows processed by the statement have been committed by other transactions.

Transactions executing at isolation level 0 (read uncommitted) do not acquire long-term or short-term read locks, and do not conflict with other transactions (except for exclusive schema locks). However, isolation level 0 transactions may process uncommitted changes made by other concurrent transactions. You can avoid processing uncommitted changes by using snapshot isolation. See "Snapshot isolation" on page 94.

Write locks

A transaction acquires a write lock whenever it inserts, updates, or deletes a row. This is true for transactions at all isolation levels, including isolation level 0 and snapshot isolation levels. No other transaction can obtain a read, intent, or write lock on the same row after a write lock is acquired. Write locks are also referred to as exclusive locks because only one transaction can hold an exclusive lock on a row at any time. No transaction can obtain a write lock while any other transaction holds a lock of any type on the same row. Similarly, once a transaction acquires a write lock, requests to lock the row by other transactions are denied.

Intent locks

Intent locks, also known as intent-for-update locks, indicate an intent to modify a particular row. Intent locks are acquired when a transaction:

- issues a FETCH FOR UPDATE statement
- issues a SELECT ... FOR UPDATE BY LOCK statement
- uses SQL_CONCUR_LOCK as its concurrency basis in an ODBC application (set by using the SQL_ATTR_CONCURRENCY parameter of the SQLSetStmtAttr ODBC API call)

Intent locks do not conflict with read locks, so acquiring an intent lock does not block other transactions from reading the same row. However, intent locks do prevent other transactions from acquiring either an

intent lock or a write lock on the same row, guaranteeing that the row cannot be changed by any other transaction before an update.

If an intent lock is requested by a transaction executing at snapshot isolation, the intent lock is only acquired if the row is an unmodified row in the database and common to all concurrent transactions. If the row is a snapshot copy, however, an intent lock is not acquired since the original row has already been modified by another transaction. Any attempt by the snapshot transaction to update that row fails and a snapshot update conflict error is returned.

Table locks

In addition to locks on rows, SQL Anywhere also supports locks on tables. Table locks are different than schema locks: a table lock places a lock on all the rows in the table, as opposed to a lock on the table's schema. There are three types of table locks:

- shared
- intent to write
- exclusive

Table locks are only released at the end of a transaction when a COMMIT or ROLLBACK occurs.

The following table identifies the combinations of table locks that conflict.

	Shared	Intent	Exclusive
Shared		conflict	conflict
Intent	conflict		conflict
Exclusive	conflict	conflict	conflict

Shared table locks

A shared table lock allows multiple transactions to read the data of a base table. A transaction that has a shared table lock on a base table can modify the table provided no other transaction holds a lock of any kind on the rows being modified.

A shared table lock is acquired, for example, by executing a LOCK TABLE ... IN SHARED MODE statement. The REFRESH MATERIALIZED VIEW and REFRESH TEXT INDEX statements also provide a WITH SHARE MODE clause that you can use to create shared table locks on the underlying tables while the refresh operation takes place.

See also

- "LOCK TABLE statement" [*SQL Anywhere Server - SQL Reference*]
- "REFRESH MATERIALIZED VIEW statement" [*SQL Anywhere Server - SQL Reference*]
- "REFRESH TEXT INDEX statement" [*SQL Anywhere Server - SQL Reference*]

Intent to write table locks

An intent to write table lock, also known as an intent table lock, is implicitly acquired the first time a write lock on a row is acquired by a transaction. As with shared table locks, intent table locks held until the transaction completes via a COMMIT or a ROLLBACK. Intent table locks conflict with shared and exclusive table locks, but not with other intent table locks.

Exclusive locks

An exclusive table lock prevents any other transaction from accessing the table for any operation (reads, writes, schema modifications, and so on). Only one transaction can hold an exclusive lock on any table at one time. Exclusive table locks conflict with all other table and row locks. However, unlike an exclusive schema lock, transactions executing at isolation level 0 can still read the rows in a table whose table lock is held exclusively.

You can acquire an exclusive table lock explicitly by using of the LOCK TABLE ... IN EXCLUSIVE MODE statement. The REFRESH MATERIALIZED VIEW and REFRESH TEXT INDEX statements also provide a WITH EXCLUSIVE MODE clause that you can use to create exclusive table locks on the underlying tables while the refresh operation takes place.

See also

- "LOCK TABLE statement" [*SQL Anywhere Server - SQL Reference*]
- "REFRESH MATERIALIZED VIEW statement" [*SQL Anywhere Server - SQL Reference*]
- "REFRESH TEXT INDEX statement" [*SQL Anywhere Server - SQL Reference*]

Position locks

In addition to row locks, SQL Anywhere also implements a form of key-range locking designed to prevent anomalies because of the presence of phantoms, or phantom rows. Position locks are only relevant only when the database server is processing transactions operating at isolation level 3.

Transactions that operate at isolation level 3 are said to be serializable. This means that a transaction's behavior at isolation level 3 should not be impacted by concurrent update activity by other transactions. In particular, at isolation level 3, transactions cannot be affected by INSERTs or UPDATEs—phantoms—that introduce rows that can affect the result of a computation. SQL Anywhere uses position locks to prevent such updates from occurring. It is this additional locking that differentiates isolation level 2 (repeatable read) from isolation level 3.

To prevent the creation of phantoms rows, SQL Anywhere acquires locks on positions within a physical scan of a table. For a sequential scan, the scan position is based on the row identifier of the current row. For an index scan, the scan's position is based on the current row's index key value (which can be unique or non-unique). Through locking a scan position, a transaction prevents insertions by other transactions relating to a particular range of values in that ordering of the rows. This includes INSERT statements and UPDATE statements that change the value of an indexed attribute. When a scan position is locked, an UPDATE statement is considered a request to DELETE the index entry followed immediately by an INSERT request.

There are two types of position locks supported by SQL Anywhere: phantom locks and anti-phantom locks. Both types of locks are shared, in that any number of transactions can acquire the same type of lock on the same row. However, phantom and anti-phantom locks conflict.

Phantom locks

A phantom lock, sometimes called an anti-insert lock, is placed on a scan position to prevent the subsequent creation of phantom rows by other transactions. When a phantom lock is acquired, it prevents other transactions from inserting a row into a table immediately before the row that is anti-insert locked. A phantom lock is a long-term lock, that is held until the end of the transaction.

Phantom locks are acquired only by transactions operating at isolation level 3; it is the only isolation level that guarantees consistency with phantoms.

For an index scan, phantom locks are acquired on each row read through the index, and one additional phantom lock is acquired at the end of the index scan to prevent insertions into the index at the end of the satisfying index range. Phantom locks with index scans prevent phantoms from being created by the insertion of new rows to the table, or the update of an indexed value that would cause the creation of an index entry at a point covered by a phantom lock.

With a sequential scan, phantom locks are acquired on every row in a table to prevent any insertion from altering the result set. So, isolation level 3 scans often have a negative effect on database concurrency. While one or more phantom locks conflict with an insert lock, and one or more read locks conflict with a write lock, no interaction exists between phantom/insert locks and read/write locks. For example, although a write lock cannot be acquired on a row that contains a read lock, it can be acquired on a row that has only a phantom lock. More options are open to the database server because of this flexible arrangement, but it means that the server must generally take the extra precaution of acquiring a read lock when acquiring a phantom lock. Otherwise, another transaction could delete the row.

Insert locks

An insert lock, sometimes termed an anti-phantom lock, is a very short-term lock placed on a scan position to reserve the right to insert a row. The lock is held only for the duration of the insertion itself; once the row is properly inserted within a database page it is write-locked to ensure consistency, and the insert lock is released. A transaction that acquires an insert lock on a row prevents other transactions from acquiring a phantom lock on the same row. Insert locks are necessary because the server must anticipate an isolation level 3 scan operation by any active connection, which could potentially occur with any new

request. Note that phantom and insert locks do not conflict with each other when they are held by the same transaction.

Locking conflicts

SQL Anywhere uses schema, row, table, and position locks as necessary to ensure the level of consistency that you require. You do not need to explicitly request the use of a particular lock. Instead, you control the level of consistency that is maintained by choosing the isolation level that best fits your requirements. Knowledge of the types of locks will guide you in choosing isolation levels and understanding the impact of each level on performance. Keep in mind that any one transaction cannot block itself by acquiring locks; a locking conflict can only occur between two (or more) transactions.

Which locks conflict?

While each of the four types of locks have specific purposes, all the types interact and therefore may cause a locking conflict between transactions. To ensure database consistency, only one transaction should change any one row at any one time. Otherwise, two simultaneous transactions might try to change one value to two different new ones. So, it is important that a row write lock be exclusive. In contrast, no difficulty arises if more than one transaction wants to read a row. Since neither is changing it, there is no conflict. So, row read locks may be shared across many connections.

The following table identifies the combination of locks that conflict. Schema locks are not included because they do not apply to rows.

Row locks	readpk	read	intent	writepk	write
readpk					conflict
read				conflict	conflict
intent			conflict	conflict	conflict
write		conflict	conflict	conflict	conflict
write	conflict	conflict	conflict	conflict	conflict

Table locks	shared	intent	exclusive
shared		conflict	conflict
intent	conflict		conflict
exclusive	conflict	conflict	conflict

Position locks	phantom	insert
phantom		conflict
insert	conflict	

Locking during queries

The locks that SQL Anywhere uses when a user enters a SELECT statement depend on the transaction's isolation level. All SELECT statements, regardless of isolation level, acquire schema locks on the referenced tables.

SELECT statements at isolation level 0

No locking operations are required when executing a SELECT statement at isolation level 0. Each transaction is not protected from changes introduced by other transactions. It is your responsibility or that of the database user to interpret the result of these queries with this limitation in mind.

SELECT statements at isolation level 1

SQL Anywhere does not use many more locks when running a transaction at isolation level 1 than it does at isolation level 0. The database server modifies its operation in only two ways.

The first difference in operation has nothing to do with acquiring locks, but rather with respecting them. At isolation level 0, a transaction can read any row, even if another transaction has acquired a write lock. By contrast, before reading each row, an isolation level 1 transaction must check whether a write lock is in place. It cannot read past any write-locked rows because doing so might entail reading dirty data. The use of the READPAST hint permits the server to ignore write-locked rows, but while the transaction will no longer block, its semantics no longer coincide with those of isolation level 1. See READPAST hint in "FROM clause" [*SQL Anywhere Server - SQL Reference*].

The second difference in operation affects cursor stability. Cursor stability is achieved by acquiring a short-term read lock on the current row of a cursor. This read lock is released when the cursor is moved. More than one row may be affected if the contents of the cursor is the result of a join. In this case, the database server acquires short-term read locks on all rows which have contributed information to the cursor's current row, and releases these locks when another row of the cursor is selected as the current row.

SELECT statements at isolation level 2

At isolation level 2, the database server modifies its operation to ensure repeatable read semantics. If a SELECT statement returns values from every row in a table, then the database server acquires a read lock on each row of the table as it reads it. If, instead, the SELECT contains a WHERE clause, or another condition which restricts the rows in the result, then the database server instead reads each row, tests the values in the row against that condition, and then acquires a read lock on the row if it meets that condition. The read locks that are acquired are long-term read locks and are held until the transaction completes via an implicit or explicit COMMIT or ROLLBACK statement. As with isolation level 1, cursor stability is assured at isolation level 2, and dirty reads are not permitted.

SELECT statements at isolation level 3

When operating at isolation level 3, the database server is obligated to ensure that all transaction schedules are serializable. In particular, in addition to the requirements imposed at isolation level 2, it must prevent phantom rows so that re-executing the same statement is guaranteed to return the same results in all circumstances.

To accommodate this requirement, the database server uses read locks and phantom locks. When executing a SELECT statement at isolation level 3, the database server acquires a read lock on each row that is processed during the computation of the result set. Doing so ensures that no other transactions can modify those rows until the transaction completes.

This requirement is similar to the operations that the database server performs at isolation level 2, but differs in that a lock must be acquired for each row read, whether those rows satisfy any predicates in the SELECT's WHERE, ON, or HAVING clauses. For example, if you select the names of all employees in the sales department, then the server must lock all the rows which contain information about a sales person, whether the transaction is executing at isolation level 2 or 3. At isolation level 3, however, the server must also acquire read locks on each of the rows of employees which are not in the sales department. Otherwise, another transaction could potentially transfer another employee to the sales department while the first transaction was still executing.

There are two implications when a read lock must be acquired for each row read:

- The database server may need to place many more locks than would be necessary at isolation level 2. The number of phantom locks acquired is one more than the number of read locks that are acquired for the scan. This doubling of the lock overhead adds to the execution time of the request.
- The acquisition of read locks on each row read has a negative impact on the concurrency of database update operations to the same table.

The number of phantom locks the database server acquires can vary greatly and depends upon the execution strategy chosen by the query optimizer. The SQL Anywhere query optimizer will attempt to avoid sequential scans at isolation level 3 because of the potentially adverse affects on overall system concurrency, but the optimizer's ability to do so depends upon the predicates in the statement and on the relevant indexes available on the referenced tables.

As an example, suppose you want to select information about the employee with Employee ID 123. As EmployeeID is the primary key of the employee table, the query optimizer will almost certainly choose an indexed strategy, using the primary key index, to locate the row efficiently. In addition, there is no danger that another transaction could change another Employee's ID to 123 because primary key values must be unique. The server can guarantee that no second employee is assigned that ID number simply by acquiring a read lock on the row containing information about employee 123.

In contrast, the database server would acquire more locks were you instead to select all the employees in the sales department. In the absence of a relevant index, the database server must read every row in the employee table and test whether each employee is in sales. If this is the case, both read and phantom locks must be acquired for each row in the table.

SELECT statements and snapshot isolation

SELECT statements that execute at snapshot, statement-snapshot, or readonly-statement-snapshot do not acquire read locks. This is because each snapshot transaction (or statement) sees a snapshot of a committed state of the database at some previous point in time. The specific point in time is determined by which of the three snapshot isolation levels is being used by the statement. As such, read transactions never block update transactions and update transactions never block readers. Therefore, snapshot isolation can give considerable concurrency benefits in addition to the obvious consistency benefits. However, there is a tradeoff; snapshot isolation can be very expensive. This is because the consistency guarantee of snapshot isolation means that copies of changed rows must be saved, tracked, and (eventually) deleted for other concurrent transactions.

Locking during inserts

INSERT operations create new rows. SQL Anywhere utilizes various types of locks during insertions to ensure data integrity. The following sequence of operations occurs for INSERT statements executing at any isolation level.

1. Acquire a shared schema lock on the table, if one is not already held.
2. Acquire an intent-to-write table lock on the table, if one is not already held.
3. Find an unlocked position in a page to store the new row. To minimize lock contention, the server will not immediately reuse space made available by deleted (but as yet uncommitted) rows. A new page may be allocated to the table (and the database file may grow) to accommodate the new row.
4. Fill the new row with any supplied values.
5. Place an insert lock in the table to which the row is being added. Recall that insert locks are exclusive, so once the insert lock is acquired, no other isolation level 3 transaction can block the insertion by acquiring a phantom lock.
6. Write lock the new row. The insert lock is released once the write lock has been obtained.
7. Insert the row into the table. Other transactions at isolation level 0 can now, for the first time, see that the new row exists. However, these other transactions cannot modify or delete the new row because of the write lock acquired earlier.
8. Update all affected indexes and verify uniqueness where appropriate. Primary key values must be unique. Other columns may also be defined to contain only unique values, and if any such columns exist, uniqueness is verified.
9. If the table is a foreign table, acquire a shared schema lock on the primary table (if not already held), and acquire a read lock on the matching primary row in the primary table if the foreign key column values being inserted are not NULL. The database server must ensure that the primary row still exists when the inserting transaction COMMITs. It does so by acquiring a read lock on the primary row. With the read lock in place, any other transaction is still free to read that row, but none can delete or update it.

If the corresponding primary row does not exist a referential integrity constraint violation is given.

After the last step, any AFTER INSERT triggers defined on the table may fire. Processing within triggers follows the identical locking behavior as for applications. Once the transaction is committed (assuming all referential integrity constraints are satisfied) or rolled back, all long-term locks are released.

Uniqueness

You can ensure that all values in a particular column, or combination of columns, are unique. The database server always performs this task by building an index for the unique column, even if you do not explicitly create one.

In particular, all primary key values must be unique. The database server automatically builds an index for the primary key of every table. Do not ask the database server to create an index on a primary key, as that index would be a redundant index.

Orphans and referential integrity

A foreign key is a reference to a primary key or UNIQUE constraint, usually in another table. When that primary key does not exist, the offending foreign key is called an **orphan**. SQL Anywhere automatically ensures that your database contains no orphans. This process is referred to as **verifying referential integrity**. The database server verifies referential integrity by counting orphans.

wait_for_commit

You can instruct the database server to delay verifying referential integrity to the end of your transaction. In this mode, you can insert a row which contains a foreign key, then subsequently insert a primary row which contains the missing primary key. Both operations must occur in the same transaction.

To request that the database server delay referential integrity checks until commit time, set the value of the option wait_for_commit to On. By default, this option is Off. To turn it on, issue the following command:

```
SET OPTION wait_for_commit = On;
```

If the server does not find a matching primary row when a new foreign key value is inserted, and wait_for_commit is On, then the server permits the insertion as an orphan. For orphaned foreign rows, upon insertion the following series of steps occurs:

- The server acquires a shared schema lock on the primary table (if not already held). The server also acquires an intent-to-write lock on the primary table.
- The server inserts a surrogate row into the primary table. An actual row is not inserted into the primary table, but the server manufactures a unique row identifier for that row for locking, and a write lock is acquired on this surrogate row. Subsequently, the server inserts the appropriate values into the primary table's primary key index

Before committing a transaction, the database server verifies that referential integrity is maintained by checking the number of orphans your transaction has created. At the end of every transaction, that number must be zero.

Locking during updates

The database server modifies the information contained in a particular record using the following procedure. As with insertions, this sequence of operations is followed for all transactions regardless of their isolation level.

1. Acquire a shared schema lock on the table, if one is not already held.
2. Acquire an intent-to-write table lock on the table, if one is not already held.
   a. Identify candidate rows to be updated. As rows are scanned, they are locked. The default locking behavior is described in "Isolation levels and consistency" on page 92.

      At isolation levels 2 and 3 the following differences occur that are different from the default locking behavior: intent-to-write row-level locks are acquired instead of read locks, and intent-to-write locks may be acquired on rows that are ultimately rejected as candidates for update.

   b. For each candidate row identified in step 2.a, follow the rest of the sequence.
3. Write lock the affected row.
4. Update each of the affected column values as per the UPDATE statement.
5. If indexed values were changed, add new index entries. The original index entries for the row remain, but are marked as deleted. New index entries for the new values are inserted while a short-term insert lock is held. The server verifies index uniqueness where appropriate.
6. If any foreign key values in the row were altered, acquire a shared schema lock on the primary table(s) and follow the procedure for inserting new foreign key values as outlined in "Locking during inserts" on page 122. Similarly, follow the procedure for WAIT_FOR_COMMIT if applicable.
7. If the table is a primary table in a referential integrity relationship, and the relationship's UPDATE action is not RESTRICT, determine the affected row(s) in the foreign table(s) by first acquiring a shared schema lock on the table(s), an intent-to-write table lock on each, and acquire write locks on all the affected rows, modifying each as appropriate. Note that this process may cascade through a nested hierarchy of referential integrity constraints.

After the last step, any AFTER UPDATE triggers may fire. Upon COMMIT, the server verifies referential integrity by ensuring that the number of orphans produced by this transaction is 0, and release all locks.

Modifying a column value can necessitate a large number of operations. The amount of work that the database server needs to do is much less if the column being modified is not part of a primary or foreign key. It is lower still if it is not contained in an index, either explicitly or implicitly because the column has been declared as unique.

The operation of verifying referential integrity during an UPDATE operation is no less simple than when the verification is performed during an INSERT. In fact, when you change the value of a primary key, you may create orphans. When you insert the replacement value, the database server must check for orphans once more.

Locking during deletes

The DELETE operation follows almost the same steps as the INSERT operation, except in the opposite order. As with insertions and updates, this sequence of operations is followed for all transactions regardless of their isolation level.

1. Acquire a shared schema lock on the table, if one is not already held.
2. Acquire an intent-to-write table lock on the table, if one is not already held.
    a. Identify candidate rows to be updated. As rows are scanned, they are locked. The default locking behavior is described in "Isolation levels and consistency" on page 92.

       At isolation levels 2 and 3 the following differences occur that are different from the default locking behavior: intent-to-write row-level locks are acquired instead of read locks, and intent-to-write locks may be acquired on rows that are ultimately rejected as candidates for update.

    b. For each candidate row identified in step 2.a, follow the rest of the sequence.
3. Write lock the row to be deleted.
4. Remove the row from the table so that it is no longer visible to other transactions. The row cannot be destroyed until the transaction is committed because doing so would remove the option of rolling back the transaction. Index entries for the deleted row are preserved, though marked as deleted, until transaction completion. This prevents other transactions from re-inserting the same row.
5. If the table is a primary table in a referential integrity relationship, and the relationship's DELETE action is not RESTRICT, determine the affected row(s) in the foreign table(s) by first acquiring a shared schema lock on the table(s), an intent-to-write table lock on each, and acquire write locks on all the affected rows, modifying each as appropriate. Note that this process may cascade through a nested hierarchy of referential integrity constraints.

The transaction can be committed provided referential integrity is not violated by doing so. To verify referential integrity, the database server also keeps track of any orphans created as a side effect of the deletion. Upon COMMIT, the server records the operation in the transaction log file and release all locks.

Lock duration

Locks are typically held by a transaction until it completes. This behavior prevents other transactions from making changes that would make it impossible to roll back the original transaction. At isolation level three, all locks must be held until a transaction ends to guarantee transaction serializability.

The only locks that are not held until the end of a transaction are cursor stability locks. These row locks are held for as long as the row in question is the current row of a cursor. In most cases, this amount of time is shorter than the lifetime of the transaction, but for WITH HOLD cursors, cursor stability locks can be held for the lifetime of the connection.

See also

- "LOCK TABLE statement" [*SQL Anywhere Server - SQL Reference*]
- "OPEN statement [ESQL] [SP]" [*SQL Anywhere Server - SQL Reference*]

Choosing isolation levels

The choice of isolation level depends on the kind of task an application is performing. This section gives some guidelines for choosing isolation levels.

To choose an appropriate isolation level, you must balance the need for consistency and accuracy with the need for concurrent transactions to proceed unimpeded. If a transaction involves only one or two specific values in one table, it is unlikely to interfere as much with other processes compared to one that searches many large tables and therefore may need to lock many rows or entire tables and may take a very long time to complete.

For example, if your transactions involve transferring money between bank accounts, you likely want to ensure that the information you return is correct. On the other hand, if you just want a rough estimate of the proportion of inactive accounts, then you may not care whether your transaction waits for others or not, and you may be willing to sacrifice some accuracy to avoid interfering with other users of the database.

Furthermore, a transfer may affect only the two rows which contain the two account balances, whereas all the accounts must be read to calculate the estimate. For this reason, the transfer is less likely to delay other transactions.

SQL Anywhere provides four isolation levels: levels 0, 1, 2, and 3. Level 3 provides complete isolation and ensures that transactions are interleaved in such a manner that the schedule is serializable.

If you have enabled snapshot isolation for a database, then three additional isolation levels are available: snapshot, statement-snapshot, and readonly-statement-snapshot.

Choosing a snapshot isolation level

Snapshot isolation offers both concurrency and consistency benefits. Using snapshot isolation incurs a cost penalty since old versions of rows are saved as long as they may be needed by running transactions. Therefore, long running snapshots can require storage of many old row versions. Usually, snapshots used for statement-snapshot do not last as long as those for snapshot. Therefore, statement-snapshot may have some space advantages over snapshot at the cost of less consistency (every statement within the transaction sees the database at a different point in time).

For more information about the performance implications of using snapshot isolation, see "Cursor sensitivity and isolation levels" [*SQL Anywhere Server - Programming*].

For most purposes, the snapshot isolation level is recommended because it provides a single view of the database for the entire transaction.

The statement-snapshot isolation level provides less consistency, but may be useful when long running transactions result in too much space being used in the temporary file by the version store.

The readonly-statement-snapshot isolation level provides less consistency than statement-snapshot, but avoids the possibility of update conflicts. Therefore, it is most appropriate for porting applications originally intended to run under different isolation levels.

For more information about snapshot isolation, see “Snapshot isolation” on page 94.

Serializable schedules

To process transactions concurrently, the database server must execute some component statements of one transaction, then some from other transactions, before continuing to process further operations from the first. The order in which the component operations of the various transactions are interleaved is called the **schedule**.

Applying transactions concurrently in this manner can result in many possible outcomes, including the three particular inconsistencies described in the previous section. Sometimes, the final state of the database also could have been achieved had the transactions been executed sequentially, meaning that one transaction was always completed in its entirety before the next was started. A schedule is called **serializable** whenever executing the transactions sequentially, in some order, could have left the database in the same state as the actual schedule.

Serializability is the commonly accepted criterion for correctness. A serializable schedule is accepted as correct because the database is not influenced by the concurrent execution of the transactions.

The isolation level affects a transaction's serializability. At isolation level 3, all schedules are serializable. The default setting is 0.

Serializable means that concurrency has added no effect

Even when transactions are executed sequentially, the final state of the database can depend upon the order in which these transactions are executed. For example, if one transaction sets a particular cell to the value 5 and another sets it to the number 6, then the final value of the cell is determined by which transaction executes last.

Knowing a schedule is serializable does not settle which order transactions would best be executed, but rather states that concurrency has added no effect. Outcomes which may be achieved by executing the set of transactions sequentially in some order are all assumed correct.

Unserializable schedules introduce inconsistencies

The inconsistencies introduced in “Typical types of inconsistency” on page 101 are typical of the types of problems that appear when the schedule is not serializable. In each case, the inconsistency appeared because of the way the statements were interleaved; the result produced would not be possible if all transactions were executed sequentially. For example, a dirty read can only occur if one transaction can select rows while another transaction is in the middle of inserting or updating data in the same row.

Typical transactions at various isolation levels

Various isolation levels lend themselves to particular types of tasks. Use the information below to help you decide which level is best suited to each particular operation.

Typical level 0 transactions

Transactions that involve browsing or performing data entry may last several minutes, and read a large number of rows. If isolation level 2 or 3 is used, concurrency can suffer. Isolation level of 0 or 1 is typically used for this kind of transaction.

For example, a decision support application that reads large amounts of information from the database to produce statistical summaries may not be significantly affected if it reads a few rows that are later modified. If high isolation is required for such an application, it may acquire read locks on large amounts of data, not allowing other applications write access to it.

Typical level 1 transactions

Isolation level 1 is useful in conjunction with cursors, because this combination ensures cursor stability without greatly increasing locking requirements. SQL Anywhere achieves this benefit through the early release of read locks acquired for the present row of a cursor. These locks must persist until the end of the transaction at either levels two or three to guarantee repeatable reads.

For example, a transaction that updates inventory levels through a cursor is suited to this level, because each of the adjustments to inventory levels as items are received and sold would not be lost, yet these frequent adjustments would have minimal impact on other transactions.

Typical level 2 transactions

At isolation level 2, rows that match your criterion cannot be changed by other transactions. You can employ this level when you must read rows more than once and rely that rows contained in your first result set won't change.

Because of the relatively large number of read locks required, you should use this isolation level with care. As with level 3 transactions, careful design of your database and indexes reduce the number of locks acquired and can improve the performance of your database.

Typical level 3 transactions

Isolation level 3 is appropriate for transactions that demand the most in security. The elimination of phantom rows lets you perform multi-step operations on a set of rows without fear that new rows will appear partway through your operations and corrupt the result.

However much integrity it provides, isolation level 3 should be used sparingly on large systems that are required to support a large number of concurrent transactions. SQL Anywhere places more locks at this level than at any other, raising the likelihood that one transaction will impede the process of many others.

Improving concurrency at isolation levels 2 and 3

Isolation levels 2 and 3 use a lot of locks and so good design is of particular importance for databases that make regular use of these isolation levels. When you must make use of serializable transactions, it is important that you design your database, in particular the indexes, with the business rules of your project

in mind. You may also improve performance by breaking large transactions into several smaller ones, and shorten the length of time that rows are locked.

Although serializable transactions have the most potential to block other transactions, they are not necessarily less efficient. When processing these transactions, SQL Anywhere can perform certain optimizations that may improve performance, in spite of the increased number of locks. For example, since all rows read must be locked whether they match the search criteria, the database server is free to combine the operation of reading rows and placing locks.

Reducing the impact of locking

To avoid placing a large number of locks that might impact the execution of other concurrent transactions, it is recommended that you avoid running transactions at isolation level 3.

When the nature of an operation demands that it run at isolation level 3, you can lower its impact on concurrency by designing the query to read as few rows and index entries as possible. These steps will help the level 3 transaction run more quickly and, of possibly greater importance, will reduce the number of locks it places.

When at least one operation executes at isolation level 3, you may find that adding an index improves transaction speed. An index can have two benefits:

- An index enables rows to be located in an efficient manner
- Searches that make use of the index may need fewer locks.

For more information about the details of the locking methods employed by SQL Anywhere is located in "How locking works" on page 111.

For more information about performance and how SQL Anywhere plans its access of information to execute your commands, see "Improving database performance" on page 157.

Isolation level tutorials

The different isolation levels behave in very different ways, and which one you will want to use depends on your database and on the operations you are performing. The following set of tutorials will help you determine which isolation levels are suitable for different tasks.

Tutorial: Dirty reads

The following tutorial demonstrates one type of inconsistency that can occur when multiple transactions are executed concurrently. Two employees at a small merchandising company access the corporate database at the same time. The first person is the company's Sales Manager. The second is the Accountant.

The Sales Manager wants to increase the price of tee shirts sold by their firm by $0.95, but is having a little trouble with the syntax of the SQL language. At the same time, unknown to the Sales Manager, the Accountant is trying to calculate the retail value of the current inventory to include in a report he volunteered to bring to the next management meeting.

Tip
Before altering your database in the following way, it is prudent to test the change by using SELECT in place of UPDATE.

Note
For this tutorial to work properly, the **Automatically Release Database Locks** option must not be selected in Interactive SQL (**Tools** » **Options** » **SQL Anywhere**).

In this example, you assume the role of two employees, both using the SQL Anywhere sample database concurrently.

1. Start Interactive SQL.
2. In the **Connect** window, connect to the SQL Anywhere sample database as the Sales Manager:
   - From the **Action** dropdown list, choose **Connect With An ODBC Data Source**.
   - In the **ODBC Data Source Name** field, choose **SQL Anywhere 12 Demo**.
   - Click **Advanced**, to reveal **Advanced Options** tab.
   - Click the **Advanced Options** tab, and type **Sales Manager** in the **ConnectionName** field.
   - Click **Connect**.
3. Start a second instance of Interactive SQL.
4. In the **Connect** window, connect to the SQL Anywhere sample database as the Accountant:
   - From the **Action** dropdown list, choose **Connect With An ODBC Data Source**.
   - In the **ODBC Data Source Name** field, choose **SQL Anywhere 12 Demo**.
   - If necessary, click **Advanced**, to reveal the **Advanced Options** tab.
   - Click the **Advanced Options** tab and type **Accountant** in the **ConnectionName** field.
   - Click **Connect**.
5. As the Sales Manager, raise the price of all tee shirts by $0.95:
   - In the **Sales Manager** window, execute the following commands:

```
UPDATE Products
   SET UnitPrice = UnitPrice + 95
   WHERE Name = 'Tee Shirt';
```



```
SELECT ID, Name, UnitPrice
   FROM Products;
```

The result is:

ID	name	UnitPrice
300	Tee Shirt	104.00
301	Tee Shirt	109.00
302	Tee Shirt	109.00
400	Baseball Cap	9.00
...	...	...

You observe immediately that you should have entered 0.95 instead of 95, but before you can fix your error, the Accountant accesses the database from another office.

6. The company's Accountant is worried that too much money is tied up in inventory. As the Accountant, execute the following commands to calculate the total retail value of all the merchandise in stock:

```
SELECT SUM( Quantity * UnitPrice )
 AS Inventory
   FROM Products;
```

The result is:

Inventory
21453.00

Unfortunately, this calculation is not accurate. The Sales Manager accidentally raised the price of the tee shirt by $95, and the result reflects this erroneous price. This mistake demonstrates one typical type of inconsistency known as a **dirty read**. You, as the Accountant, accessed data which the Sales Manager has entered, but has not yet committed.

You can eliminate dirty reads and other inconsistencies explained in “Isolation levels and consistency” on page 92.

7. As the Sales Manager, fix the error by rolling back your first changes and entering the correct UPDATE command. Check that your new values are correct.

```
ROLLBACK;
UPDATE Products
SET UnitPrice = UnitPrice + 0.95
WHERE NAME = 'Tee Shirt';
```

ID	name	UnitPrice
300	Tee Shirt	9.95
301	Tee Shirt	14.95
302	Tee Shirt	14.95
400	Baseball Cap	9.00
...	...	...

8. The Accountant does not know that the amount he calculated was in error. You can see the correct value by executing the SELECT statement again in the Accountant's window.

```
SELECT SUM( Quantity * UnitPrice )
 AS Inventory
   FROM Products;
```

Inventory
6687.15

9. Finish the transaction in the Sales Manager's window. The Sales Manager would enter a COMMIT statement to make the changes permanent, but you should execute a ROLLBACK, instead, to avoid changing the local copy of the SQL Anywhere sample database.

```
ROLLBACK;
```

   The Accountant unknowingly receives erroneous information from the database because the database server is processing the work of both the Sales Manager and the Accountant concurrently.

10. (optional) Restore the sample database (*demo.db*) to its original state by following the steps found here: "Recreate the sample database (demo.db)" [*SQL Anywhere 12 - Introduction*].

Using snapshot isolation to avoid dirty reads

When you use snapshot isolation, other database connections see only committed data in response to their queries. Setting the isolation level to statement-snapshot or snapshot prevents the possibility of dirty reads occurring. The Accountant can use snapshot isolation to ensure that they only see committed data when executing their queries.

1. Start Interactive SQL.
2. In the **Connect** window, connect to the SQL Anywhere sample database as the Sales Manager:
   - From the **Action** dropdown list, choose **Connect With An ODBC Data Source**.
   - In the **ODBC Data Source Name** field, choose **SQL Anywhere 12 Demo**.
   - If necessary, click **Advanced**, to reveal the **Advanced Options** tab.

- Click the **Advanced Options** tab and type **Sales Manager** in the **ConnectionName** field.
- Click **Connect**.

3. Execute the following statement to enable snapshot isolation for the database:

```
SET OPTION PUBLIC.allow_snapshot_isolation = 'ON';
```

4. Start a second instance of Interactive SQL.
5. In the **Connect** window, connect to the SQL Anywhere sample database as the Accountant:
   - From the **Action** dropdown list, choose **Connect With An ODBC Data Source**.
   - In the **ODBC Data Source Name** field, choose **SQL Anywhere 12 Demo**.
   - If necessary, click **Advanced**, to reveal the **Advanced Options** tab.
   - Click the **Advanced Options** tab and type **Accountant** in the **ConnectionName** field.
   - Click **Connect**.
6. As the Sales Manager, raise the price of all the tee shirts by $0.95:
   - In the window labeled Sales Manager, execute the following command to:

```
UPDATE Products
SET UnitPrice = UnitPrice + 0.95
WHERE Name = 'Tee Shirt';
```

   - Calculate the total retail value of all merchandise in stock using the new tee shirt price for the Sales Manager:

```
SELECT SUM( Quantity * UnitPrice )
 AS Inventory
   FROM Products;
```

   The result is:

Inventory
6687.15

7. As the Accountant, execute the following command to calculate the total retail value of all the merchandise in stock. Because this transaction uses the snapshot isolation level, the result is calculated only for data that has been committed to the database.

```
SET OPTION isolation_level = 'Snapshot';
SELECT SUM( Quantity * UnitPrice )
 AS Inventory
   FROM Products;
```

   The result is:

Inventory
6538.00

8. As the Sales Manager, commit your changes to the database by executing the following statement:

```
COMMIT;
```

9. As the Accountant, execute the following statements to view the updated retail value of the current inventory:

```
COMMIT;
SELECT SUM( Quantity * UnitPrice )
 AS Inventory
   FROM Products;
```

The result is:

Inventory
6687.15

Because the snapshot used for the Accountant's transaction began with the first read operation, you must execute a COMMIT to end the transaction and allow the Accountant to see changes made to the data after the snapshot transaction began. See “Understanding snapshot transactions” on page 97.

10. As the Sales Manager, execute the following statement to undo the tee shirt price changes and restore the SQL Anywhere sample database to its original state:

```
UPDATE Products
SET UnitPrice = UnitPrice - 0.95
WHERE Name = 'Tee Shirt';
COMMIT;
```

11. (optional) Restore the sample database (*demo.db*) to its original state by following the steps found here: “Recreate the sample database (demo.db)” [*SQL Anywhere 12 - Introduction*].

Tutorial: Non-repeatable reads

The example in “Tutorial: Dirty reads” on page 129 demonstrated the first type of inconsistency, namely the dirty read. In that example, an Accountant made a calculation while the Sales Manager was in the process of updating a price. The Accountant's calculation used erroneous information which the Sales Manager had entered and was in the process of fixing.

The following example demonstrates another type of inconsistency: non-repeatable reads. In this example, you assume the role of the same two employees, both using the SQL Anywhere sample database concurrently. The Sales Manager wants to offer a new sales price on plastic visors. The Accountant wants to verify the prices of some items that appear on a recent order.

This example begins with both connections at isolation level 1, rather than at isolation level 0, which is the default for the SQL Anywhere sample database supplied with SQL Anywhere. By setting the isolation

level to 1, you eliminate the type of inconsistency which the previous tutorial demonstrated, namely the dirty read.

Note
For this tutorial to work properly, the **Automatically Release Database Locks** option must not be selected in Interactive SQL (**Tools** » **Options** » **SQL Anywhere**).

1. Start Interactive SQL.
2. In the **Connect** window, connect to the SQL Anywhere sample database as the Sales Manager:
   a. From the **Action** dropdown list, choose **Connect With An ODBC Data Source**.
   b. In the **ODBC Data Source Name** field, choose **SQL Anywhere 12 Demo**.
   c. If necessary, click **Advanced**, to reveal the **Advanced Options** tab.
   d. Click the **Advanced Options** tab and type **Sales Manager** in the **ConnectionName** field.
   e. Click **Connect**.
3. Start a second instance of Interactive SQL.
4. In the **Connect** window, connect to the SQL Anywhere sample database as the Accountant:
   a. From the **Action** dropdown list, choose **Connect With An ODBC Data Source**.
   b. In the **ODBC Data Source Name** field, choose **SQL Anywhere 12 Demo**.
   c. If necessary, click **Advanced**, to reveal the **Advanced Options** tab.
   d. Click the **Advanced Options** tab and type **Accountant** in the **ConnectionName** field.
   e. Click **Connect**.
5. Set the isolation level to 1 for the Accountant's connection by executing the following command.

   ```
   SET TEMPORARY OPTION isolation_level = 1;
   ```

6. Set the isolation level to 1 in the Sales Manager's window by executing the following command:

   ```
   SET TEMPORARY OPTION isolation_level = 1;
   ```

7. The Accountant decides to list the prices of the visors. As the Accountant, execute the following command:

   ```
   SELECT ID, Name, UnitPrice FROM Products;
   ```

ID	Name	UnitPrice
300	Tee Shirt	9.00

ID	Name	UnitPrice
301	Tee Shirt	14.00
302	Tee Shirt	14.00
400	Baseball Cap	9.00
401	Baseball Cap	10.00
500	Visor	7.00
501	Visor	7.00
...	...	...

8. The Sales Manager decides to introduce a new sale price for the plastic visor. As the Sales Manager, execute the following command:

```
SELECT ID, Name, UnitPrice FROM Products
WHERE Name = 'Visor';
UPDATE Products
SET UnitPrice = 5.95 WHERE ID = 501;
COMMIT;
SELECT ID, Name, UnitPrice FROM Products
WHERE Name = 'Visor';
```

ID	Name	UnitPrice
500	Visor	7.00
501	Visor	5.95

9. Compare the price of the visor in the Sales Manager window with the price for the same visor in the Accountant window. The Accountant executes the SELECT statement again, and sees the Sales Manager's new sale price.

```
SELECT ID, Name, UnitPrice
FROM Products;
```

ID	Name	UnitPrice
300	Tee Shirt	9.00
301	Tee Shirt	14.00
302	Tee Shirt	14.00
400	Baseball Cap	9.00

ID	Name	UnitPrice
401	Baseball Cap	10.00
500	Visor	7.00
501	Visor	5.95
...	...	...

This inconsistency is called a **non-repeatable read** because when the Accountant executes the same SELECT a second time in the *same transaction*, and did not get the same results.

Of course if the Accountant had finished his transaction, for example by issuing a COMMIT or ROLLBACK command before using SELECT again, it would be a different matter. The database is available for simultaneous use by multiple users and it is completely permissible for someone to change values either before or after the Accountant's transaction. The change in results is only inconsistent because it happens in the middle of his transaction. Such an event makes the schedule unserializable.

10. The Accountant notices this behavior and decides that from now on he doesn't want the prices changing while he looks at them. Non-repeatable reads are eliminated at isolation level 2. As the Accountant, execute the following statements:

    ```
    SET TEMPORARY OPTION isolation_level = 2;
    SELECT ID, Name, UnitPrice
    FROM Products;
    ```

11. The Sales Manager decides that it would be better to delay the sale on the plastic visor until next week so that she won't have to give the lower price on a big order that she's expecting will arrive tomorrow. In her window, try to execute the following statements. The command starts to execute, and then her window appears to freeze.

    ```
    UPDATE Products
    SET UnitPrice = 7.00
    WHERE ID = 501;
    ```

    The database server must guarantee repeatable reads at isolation level 2. Because the Accountant is using isolation level 2, the database server places a read lock on each row of the Products table that the Accountant reads. When the Sales Manager tries to change the price back, her transaction must acquire a write lock on the plastic visor row of the Products table. Since write locks are exclusive, her transaction must wait until the Accountant's transaction releases its read lock.

12. The Accountant is finished looking at the prices. He doesn't want to risk accidentally changing the database, so he completes his transaction with a ROLLBACK statement.

    ```
    ROLLBACK;
    ```

    When the database server executes this statement, the Sales Manager's transaction completes.

ID	Name	UnitPrice
500	Visor	7.00
501	Visor	7.00

13. The Sales Manager can finish now. She wants to commit her change to restore the original price.

```
COMMIT;
```

Types of locks and different isolation levels

When you upgraded the Accountant's isolation from level 1 to level 2, the database server used read locks where none had previously been acquired. In general, each isolation level is characterized by the types of locks needed and by how locks held by other transactions are treated.

At isolation level 0, the database server needs only write locks. It makes use of these locks to ensure that no two transactions make modifications that conflict. For example, a level 0 transaction acquires a write lock on a row before it updates or deletes it, and inserts any new rows with a write lock already in place.

Level 0 transactions perform no checks on the rows they are reading. For example, when a level 0 transaction reads a row, it does not check what locks may or may not have been acquired on that row by other transactions. Since no checks are needed, level 0 transactions are fast. This speed comes at the expense of consistency. Whenever they read a row which is write locked by another transaction, they risk returning dirty data.

At level 1, transactions check for write locks before they read a row. Although one more operation is required, these transactions are assured that all the data they read is committed. Try repeating the first tutorial with the isolation level set to 1 instead of 0. You will find that the Accountant's computation cannot proceed while the Sales Manager's transaction, which updates the price of the tee shirts, remains incomplete.

When the Accountant raised his isolation to level 2, the database server began using read locks. From then on, it acquired a read lock for his transaction on each row that matched his selection.

Transaction blocking

In the above tutorial, the Sales Manager's window froze during the execution of her UPDATE command. The database server began to execute her command, then found that the Accountant's transaction had acquired a read lock on the row that the Sales Manager needed to change. At this point, the database server simply paused the execution of the UPDATE. Once the Accountant finished his transaction with the ROLLBACK, the database server automatically released his locks. Finding no further obstructions, it then proceeded to complete execution of the Sales Manager's UPDATE.

In general, a locking conflict occurs when one transaction attempts to acquire an exclusive lock on a row on which another transaction holds a lock, or attempts to acquire a shared lock on a row on which another transaction holds an exclusive lock. One transaction must wait for another transaction to complete. The transaction that must wait is said to be **blocked** by another transaction.

When the database server identifies a locking conflict which prohibits a transaction from proceeding immediately, it can either pause execution of the transaction, or it can terminate the transaction, roll back

any changes, and return an error. You control the route by setting the blocking option. When the blocking is set to On the second transaction waits, as in the above tutorial.

For more information about the blocking option, see "The blocking option" on page 108.

Using snapshot isolation to avoid non-repeatable reads

You can also use snapshot isolation to help avoid blocking. Because transactions that use snapshot isolation only see committed data, the Accountant's transaction does not block the Sales Manager's transaction.

1. Start Interactive SQL.
2. In the **Connect** window, connect to the SQL Anywhere sample database as the Sales Manager:
   a. From the **Action** dropdown list, choose **Connect to an ODBC Data Source**.
   b. In the **ODBC Data Source Name** field, choose **SQL Anywhere 12 Demo**.
   c. If necessary, click **Advanced**, to reveal the **Advanced Options** tab.
   d. Click the **Advanced Options** tab and type **Sales Manager** in the **ConnectionName** field.
   e. Click **Connect**.
3. Start a second instance of Interactive SQL.
4. In the **Connect** window, connect to the SQL Anywhere sample database as the Accountant:
   a. From the **Action** dropdown list, choose **Connect With An ODBC Data Source**.
   b. In the **ODBC Data Source Name** field, choose **SQL Anywhere 12 Demo**.
   c. If necessary, click **Advanced** button, to reveal the **Advanced Options** tab.
   d. Click the **Advanced Options** tab and type **Accountant** in the **ConnectionName** field.
   e. Click **Connect**.
5. Execute the following statements to enable snapshot isolation for the database and specify that the snapshot isolation level is used:

```
SET OPTION PUBLIC.allow_snapshot_isolation = 'On';
SET TEMPORARY OPTION isolation_level = 'snapshot';
```

6. The Accountant decides to list the prices of the visors. As the Accountant, execute the following command:

```
SELECT ID, Name, UnitPrice
FROM Products
ORDER BY ID;
```

ID	Name	UnitPrice
300	Tee Shirt	9.00
301	Tee Shirt	14.00
302	Tee Shirt	14.00
400	Baseball Cap	9.00
401	Baseball Cap	10.00
500	Visor	7.00
501	Visor	7.00
...	...	...

7. The Sales Manager decides to introduce a new sale price for the plastic visor. As the Sales Manager, execute the following command:

```
UPDATE Products
SET UnitPrice = 5.95 WHERE ID = 501;
COMMIT;
SELECT ID, Name, UnitPrice FROM Products
WHERE Name = 'Visor';
```

8. The Accountant executes his query again, and does not see the change in price because the data that was committed at the time of the first read is used for the transaction.

```
SELECT ID, Name, UnitPrice
FROM Products;
```

9. As the Sales Manager, change the plastic visor back to its original price.

```
UPDATE Products
SET UnitPrice = 7.00
WHERE ID = 501;
COMMIT;
```

   The database server does not place a read lock on the rows in the Products table that the Accountant is reading because the Accountant is viewing a snapshot of committed data that was taken before the Sales Manager made any changes to the Products table.

10. The Accountant is finished looking at the prices. He doesn't want to risk accidentally changing the database, so he completes his transaction with a ROLLBACK statement.

```
ROLLBACK;
```

Tutorial: Phantom rows

In this tutorial, you will observe the appearance of a phantom row.

Note
For this tutorial to work properly, the **Automatically Release Database Locks** option must not be selected in Interactive SQL (**Tools** » **Options** » **SQL Anywhere**).

1. Start two instances of Interactive SQL. See steps 1 through 4 of "Tutorial: Non-repeatable reads" on page 134.

2. Set the isolation level to 2 in the Sales Manager window by executing the following command.

   ```
   SET TEMPORARY OPTION isolation_level = 2;
   ```

3. Set the isolation level to 2 for the Accountant window by executing the following command.

   ```
   SET TEMPORARY OPTION isolation_level = 2;
   ```

4. In the Accountant window, enter the following command to list all the departments.

   ```
   SELECT * FROM Departments
   ORDER BY DepartmentID;
   ```

DepartmentID	DepartmentName	DepartmentHeadID
100	R & D	501
200	Sales	902
300	Finance	1293
400	Marketing	1576
500	Shipping	703

5. The Sales Manager decides to set up a new department to focus on the foreign market. Philip Chin, who has EmployeeID 129, heads the new department.

   ```
   INSERT INTO Departments
      ( DepartmentID, DepartmentName, DepartmentHeadID )
      VALUES( 600, 'Foreign Sales', 129 );

   COMMIT;
   ```

   The final command creates the new entry for the new department. It appears as a new row at the bottom of the table in the Sales Manager's window.

   In the Sales Manager window, enter the following command to list all the departments.

   ```
   SELECT *
   FROM Departments
   ORDER BY DepartmentID;
   ```

DepartmentID	DepartmentName	DepartmentHeadID
100	R & D	501
200	Sales	902
300	Finance	1293
400	Marketing	1576
500	Shipping	703
600	Foreign Sales	129

6. The Accountant, however, is not aware of the new department. At isolation level 2, the database server places locks to ensure that no row changes, but places no locks that stop other transactions from inserting new rows.

   The Accountant will only discover the new row if he executes his SELECT command again. In the Accountant's window, execute the SELECT statement again. You will see the new row appended to the table.

   ```
   SELECT *
   FROM Departments
   ORDER BY DepartmentID;
   ```

DepartmentID	DepartmentName	DepartmentHeadID
100	R & D	501
200	Sales	902
300	Finance	1293
400	Marketing	1576
500	Shipping	703
600	Foreign Sales	129

   The new row that appears is called a **phantom row** because, from the Accountant's point of view, it appears like an apparition, seemingly from nowhere. The Accountant is connected at isolation level 2. At that level, the database server acquires locks only on the rows that he is using. Other rows are left untouched, so there is nothing to prevent the Sales Manager from inserting a new row.

7. The Accountant would prefer to avoid such surprises in future, so he raises the isolation level of his current transaction to level 3. Enter the following commands for the Accountant.

   ```
   SET TEMPORARY OPTION isolation_level = 3;
   SELECT *
   ```



```
FROM Departments
ORDER BY DepartmentID;
```

8. The Sales Manager would like to add a second department to handle a sales initiative aimed at large corporate partners. Execute the following command in the Sales Manager's window.

   ```
   INSERT INTO Departments
    ( DepartmentID, DepartmentName, DepartmentHeadID )
      VALUES( 700, 'Major Account Sales', 902 );
   ```

   The Sales Manager's window pauses during execution because the Accountant's locks block the command. From the toolbar, click **Stop** (or choose **SQL** » **Stop**) to interrupt this entry.

9. To avoid changing the SQL Anywhere sample database, you should roll back the incomplete transaction that inserts the Major Account Sales department row and use a second transaction to delete the Foreign Sales department.

   a. Execute the following command in the Sales Manager's window to rollback the last, incomplete transaction:

      ```
      ROLLBACK;
      ```

   b. Also in the Sales Manager's window, execute the following two statements to delete the row that you inserted earlier and commit this operation.

      ```
      DELETE FROM Departments
      WHERE DepartmentID = 600;

      COMMIT;
      ```

Explanation

When the Accountant raised his isolation to level 3 and again selected all rows in the Departments table, the database server placed anti-insert locks on each row in the table, and added one extra phantom lock to block inserts at the end of the table. When the Sales Manager attempted to insert a new row at the end of the table, it was this final lock that blocked her command.

Notice that the Sales Manager's command was blocked even though she is still connected at isolation level 2. The database server places anti-insert locks, like read locks, as demanded by the isolation level and statements of each transactions. Once placed, these locks must be respected by all other concurrent transactions.

For more information about locking, see “How locking works” on page 111.

Using snapshot isolation to avoid phantom rows

You can use the snapshot isolation level to maintain consistency at the same level as isolation level at 3, without any sort of blocking. The Sales Manager's command is not blocked, and the Accountant does not see a phantom row.

If you have not done so, follow steps 1 through 4 of the “Tutorial: Phantom rows” on page 140 which describe how to start two instances of Interactive SQL.

1. Enable snapshot isolation for the Accountant by executing the following command.

```
SET OPTION PUBLIC. allow_snapshot_isolation = 'On';
SET TEMPORARY OPTION isolation_level = 'snapshot';
```

2. In the Accountant window, enter the following command to list all the departments.

```
SELECT * FROM Departments
ORDER BY DepartmentID;
```

DepartmentID	DepartmentName	DepartmentHeadID
100	R & D	501
200	Sales	902
300	Finance	1293
400	Marketing	1576
500	Shipping	703

3. The Sales Manager decides to set up a new department to focus on the foreign market. Philip Chin, who has EmployeeID 129, heads the new department.

```
INSERT INTO Departments
   ( DepartmentID, DepartmentName, DepartmentHeadID )
   VALUES( 600, 'Foreign Sales', 129 );
COMMIT;
```

The final command creates the new entry for the new department. It appears as a new row at the bottom of the table in the Sales Manager's window.

```
SELECT * FROM Departments
ORDER BY DepartmentID;
```

DepartmentID	DepartmentName	DepartmentHeadID
100	R & D	501
200	Sales	902
300	Finance	1293
400	Marketing	1576
500	Shipping	703
600	Foreign Sales	129

4. The Accountant can execute his query again, and does not see the new row because the transaction has not ended.

```
SELECT *
FROM Departments
ORDER BY DepartmentID;
```


DepartmentID	DepartmentName	DepartmentHeadID
100	R & D	501
200	Sales	902
300	Finance	1293
400	Marketing	1576
500	Shipping	703

5. The Sales Manager would like to add a second department to handle sales initiative aimed at large corporate partners. Execute the following command in the Sales Manager's window.

```
INSERT INTO Departments
 ( DepartmentID, DepartmentName, DepartmentHeadID )
   VALUES( 700, 'Major Account Sales', 902 );
```

   The Sales Manager's change is not blocked because the Accountant is using snapshot isolation.

6. The Accountant must end his snapshot transaction to see the changes the Sales Manager committed to the database.

```
COMMIT;
   SELECT * FROM Departments
   ORDER BY DepartmentID;
```

   Now the Accountant sees the Foreign Sales department, but not the Major Account Sales department.

DepartmentID	DepartmentName	DepartmentHeadID
100	R & D	501
200	Sales	902
300	Finance	1293
400	Marketing	1576
500	Shipping	703
600	Foreign Sales	129

7. To avoid changing the SQL Anywhere sample database, you should roll back the incomplete transaction that inserts the Major Account Sales department row and use a second transaction to delete the Foreign Sales department.

   a. Execute the following command in the Sales Manager's window to rollback the last, incomplete transaction:

```
ROLLBACK;
```

b. Also in the Sales Manager's window, execute the following two statements to delete the row that you inserted earlier and commit this operation.

```
DELETE FROM Departments
WHERE DepartmentID = 600;

COMMIT;
```

Tutorial: Practical locking implications

In this tutorial the Accountant and the Sales Manager both have tasks that involve the SalesOrder and SalesOrderItems tables. The Accountant needs to verify the amounts of the commission checks paid to the sales employees for the sales they made during the month of April 2001. The Sales Manager notices that a few orders have not been added to the database and wants to add them.

Their work demonstrates phantom locking. A **phantom lock** is a shared lock that is placed on an indexed scan position to prevent phantom rows. When a transaction at isolation level 3 selects rows that match the specified criteria, the database server places anti-insert locks to stop other transactions from inserting rows that would also match. The number of locks placed on your behalf depends both on the search criteria and on the design of your database.

Note
For this tutorial to work properly, the **Automatically Release Database Locks** option must not be selected in Interactive SQL. You can check the setting of this option by choosing **Tools** » **Options**, and then clicking **SQL Anywhere** in the left pane.

1. Start two instances of Interactive SQL. See steps 1 through 4 of "Tutorial: Non-repeatable reads" on page 134.

2. Set the isolation level to 2 in both the Sales Manager window and the Accountant window by executing the following command.

```
SET TEMPORARY OPTION isolation_level = 2;
```

3. Each month, the sales representatives are paid a commission that is calculated as a percentage of their sales for that month. The Accountant is preparing the commission checks for the month of April 2001. His first task is to calculate the total sales of each representative during this month.

   Enter the following command in the Accountant's window. Prices, sales order information, and employee data are stored in separate tables. Join these tables using the foreign key relationships to combine the necessary pieces of information.

```
SELECT EmployeeID, GivenName, Surname,
   SUM( SalesOrderItems.Quantity * UnitPrice )
      AS "April sales"
FROM Employees
   KEY JOIN SalesOrders
   KEY JOIN SalesOrderItems
   KEY JOIN Products
```



```
WHERE '2001-04-01' <= OrderDate
   AND OrderDate < '2001-05-01'
GROUP BY  EmployeeID, GivenName, Surname
ORDER BY EmployeeID;
```

EmployeeID	GivenName	Surname	April sales
129	Philip	Chin	2160.00
195	Marc	Dill	2568.00
299	Rollin	Overbey	5760.00
467	James	Klobucher	3228.00
...	...	...	...

4. The Sales Manager notices that a big order sold by Philip Chin was not entered into the database. Philip likes to be paid his commission promptly, so the Sales Manager enters the missing order, which was placed on April 25.

   In the Sales Manager's window, enter the following commands. The sales order and the items are entered in separate tables because one order can contain many items. You should create the entry for the sales order before you add items to it. To maintain referential integrity, the database server allows a transaction to add items to an order only if that order already exists.

   ```
   INSERT into SalesOrders
   VALUES ( 2653, 174, '2001-04-22', 'r1',
         'Central', 129 );
   INSERT into SalesOrderItems
   VALUES ( 2653, 1, 601, 100, '2001-04-25' );
   COMMIT;
   ```

5. The Accountant has no way of knowing that the Sales Manager has just added a new order. Had the new order been entered earlier, it would have been included in the calculation of Philip Chin's April sales.

   In the Accountant's window, calculate the April sales totals again. Use the same command, and observe that Philip Chin's April sales changes to $4560.00.

EmployeeID	GivenName	Surname	April sales
129	Philip	Chin	4560.00
195	Marc	Dill	2568.00
299	Rollin	Overbey	5760.00
467	James	Klobucher	3228.00
...	...	...	...

Imagine that the Accountant now marks all orders placed in April to indicate that commission has been paid. The order that the Sales Manager just entered might be found in the second search and marked as paid, even though it was not included in Philip's total April sales.

6. At isolation level 3, the database server places anti-insert locks to ensure that no other transactions can add a row that matches the criteria of a search or select.

   In the Sales Manager's window, execute the following statements to remove the new order.

   ```
   DELETE
   FROM SalesOrderItems
   WHERE ID = 2653;
   DELETE
   FROM SalesOrders
   WHERE ID = 2653;
   COMMIT;
   ```

7. In the Accountant's window, execute the following two statements.

   ```
   ROLLBACK;
   SET TEMPORARY OPTION isolation_level = 3;
   ```

8. In the Accountant's window, execute same query as before.

   ```
   SELECT EmployeeID, GivenName, Surname,
      SUM( SalesOrderItems.Quantity * UnitPrice )
         AS "April sales"
   FROM Employees
      KEY JOIN SalesOrders
      KEY JOIN SalesOrderItems
      KEY JOIN Products
   WHERE '2001-04-01' <= OrderDate
      AND OrderDate < '2001-05-01'
   GROUP BY  EmployeeID, GivenName, Surname;
   ```

   Because you set the isolation to level 3, the database server automatically places anti-insert locks to ensure that the Sales Manager cannot insert April order items until the Accountant finishes their transaction.

9. Return to the Sales Manager's window. Attempt to enter Philip Chin's missing order by executing the following command.

   ```
   INSERT INTO SalesOrders
   VALUES ( 2653, 174, '2001-04-22',
            'r1','Central', 129 );
   ```

   The Sales Manager's window stops responding, and the operation does not complete. On the toolbar, click **Stop** (or choose **SQL** » **Stop**) to interrupt this entry.

10. The Sales Manager cannot enter the order in April, but you might think that they could still enter it in May.

    Change the date of the command to May 05 and try again.

    ```
    INSERT INTO SalesOrders
    VALUES ( 2653, 174, '2001-05-05', 'r1',
          'Central', 129 );
    ```


The Sales Manager's window stops responding again. On the toolbar, click **Stop** (or choose **SQL** » **Stop**) to interrupt this entry. Although the database server places no more locks than necessary to prevent insertions, these locks have the potential to interfere with many transactions.

The database server places locks in table indexes. For example, it places a phantom lock in an index so a new row cannot be inserted immediately before it. However, when no suitable index is present, it must lock every row in the table.

In some situations, anti-insert locks may block some insertions into a table, yet allow others.

11. Conclude this tutorial by undoing any changes to avoid changing the SQL Anywhere sample database. Execute the following statement in the Sales Manager's window.

    ```
    ROLLBACK;
    ```

    Execute the following statement in the Accountant's window.

    ```
    ROLLBACK;
    ```

12. Shut down both instances of Interactive SQL.

Primary key generation and concurrency

You will encounter situations where the database should automatically generate a unique number. For example, if you are building a table to store sales invoices you might prefer that the database assign unique invoice numbers automatically, rather than require sales staff to pick them.

Example

For example, invoice numbers could be obtained by adding 1 to the previous invoice number. This method does not work when there is more than one person adding invoices to the database. Two employees may decide to use the same invoice number.

There is more than one solution to the problem:

- Assign a range of invoice numbers to each person who adds new invoices.

  You could implement this scheme by creating a table with the columns user name and invoice number. The table would have one row for each user that adds invoices. Each time a user adds an invoice, the number in the table would be incremented and used for the new invoice. To handle all tables in the database, the table should have three columns: table name, user name, and last key value. You should periodically verify that each person has enough numbers.

- Create a table with the columns table name and last key value.

  One row in the table contains the last invoice number used. The invoice number is automatically incremented every time a user adds an invoice, establishes a new connection, increments the invoice number, or immediately commits a change. Other users can access new invoice numbers because the row is instantly updated by a separate transaction.

- Use a column with a default value of NEWID in conjunction with the UNIQUEIDENTIFIER binary data type to generate a universally unique identifier.

  You can use UUID and GUID values to uniquely identify table rows. Because the values generated on one computer do not match the values generated on another computer, the UUID and GUID values can be used as keys in replication and synchronization environments.

  For more information about generating unique identifiers, see "The NEWID default" on page 72.

- Use a column with a default value of AUTOINCREMENT. For example:

```
CREATE TABLE Orders (
   OrderID INTEGER NOT NULL DEFAULT AUTOINCREMENT,
   OrderDate DATE,
   primary key( OrderID )
);
```

  On inserts into the table, if a value is not specified for the autoincrement column, a unique value is generated. If a value is specified, it will be used. If the value is larger than the current maximum value for the column, that value will be used as a starting point for subsequent inserts. The value of the most recently inserted row in an autoincrement column is available as the global variable @@identity.

Using a sequence to generate unique values

You can use a **sequence** to generate values that are unique across multiple tables or that are different from a set of natural numbers. A sequence is created using the CREATE SEQUENCE statement. Sequence values are returned as BIGINT values.

For each connection, the most recent use of the next value is saved as the current value.

When you create a sequence, its definition includes the number of sequence values the database server holds in memory. When this cache is exhausted, the sequence cache is repopulated. If the database server fails, then sequence values that were held in the cache may be skipped.

Obtaining values in a sequence

Use the following statement to obtain the current value in the sequence:

```
SELECT sequence-name.CURRVAL;
```

Use the following statement to obtain the next value in the sequence:

```
SELECT sequence-name.NEXTVAL;
```

You must have DBA authority, be the owner of the sequence, or have been granted permission to use the sequence to execute these statements. For more information, see "sequence-expression clause, SELECT statement" [*SQL Anywhere Server - SQL Reference*].

Choosing between sequences and autoincrement values

Autoincrement behavior	Sequence behavior
Defined for a single column in a table	Stored as a database object and can be used anywhere that an expression is allowed
Column must have an integer data type or an exact numeric data type	Values can be referred to anywhere that an expression can be used and do not have to conform to default value for a column
Values can only be used for a single column in one table	Values can be used across multiple tables
Values are part of the set of natural numbers (1, 2, 3, ...)	Can generate values other than the set of natural numbers
Values must increment	Values can increment or decrement
A unique value that is one greater than the previous maximum value in the column is generated by default The sa_reset_identity system procedure can be used to change the autoincrement value for the next row that will be inserted	Unit of increment can be specified
If the next value to be generated exceeds the maximum value that can be stored in the column, NULL is returned	Can choose to allow values to be generated after the maximum or minimum value is reached, or return an error by specifying NO CYCLE

Sequence example

Consider a sequence that is used to generate incident numbers for a customer hotline. Suppose that customers can call in with two different types of complaints: incorrect billing or missing shipments.

```
CREATE SEQUENCE incidentSequence
   MINVALUE 1000
   MAXVALUE 100000;

CREATE TABLE reportedBillingMistake(
   incidentID INT PRIMARY KEY DEFAULT (incidentSequence.nextval),
   billNumber INT,
   valueOnBill NUMERIC(10,2),
   expectedValue NUMERIC(10,2),
   comments LONG VARCHAR );

CREATE TABLE reportedMissingShipment(
   incidentID INT PRIMARY KEY DEFAULT(incidentSequence.nextval),
   orderNumber INT,
   comments LONG VARCHAR );
```

Using incidentSequence.nextval for the incidentID columns guarantees that incidentIDs are unique across the two tables. This means that when a customer calls back for further inquiries and provides an incident value, there is no possibility of confusion as to whether the incident is a billing or shipping shipment.

To insert a billing mistake, the following statements would be equivalent:

```
INSERT INTO reportedBillingMistake VALUES( DEFAULT, 12345, 100.00, 75.00,
'Bad bill' );

INSERT INTO reportedBillingMistake
   SELECT incidentSequence.nextval, 12345, 100.00, 75.00, 'Bad bill';
```

To find the incidentID that was just inserted, the connection that performed the insert (using either of the above two statements) could issue the following statement:

```
SELECT incidentSequence.currval;
```

See also

- "CREATE SEQUENCE statement" [*SQL Anywhere Server - SQL Reference*]
- "The AUTOINCREMENT default" on page 70
- "The GLOBAL AUTOINCREMENT default" on page 71

Create a sequence

To create a sequence (Sybase Central)

1. Use the SQL Anywhere 12 plug-in to connect to the database as a user with RESOURCE or DBA authority.
2. In the left pane, right-click **Sequence Generators**, and choose **New** » **Sequence Generator**.
3. Follow the instructions in the **Create Sequence Generator Wizard**.

To create a sequence (SQL)

1. Use the SQL Anywhere 12 plug-in to connect to the database as a user with RESOURCE or DBA authority.
2. Execute a CREATE SEQUENCE statement. See "CREATE SEQUENCE statement" [*SQL Anywhere Server - SQL Reference*].
3. If other users need to have access to the sequence, you must execute a GRANT USAGE ON SEQUENCE statement. See "GRANT statement" [*SQL Anywhere Server - SQL Reference*].

Alter a sequence

To alter a sequence (Sybase Central)

1. Use the SQL Anywhere 12 plug-in to connect to the database as a user with RESOURCE or DBA authority.
2. Right-click a sequence generator and then choose **Properties**.

   On the **General** tab, you can change the settings for the sequence. Clicking **Restart Now** executes an ALTER SEQUENCE ... RESTART WITH *n* statement, where *n* corresponds to the value in the **Start Value** field. The change takes effect immediately.

To alter a sequence (SQL)

1. Connect to the database as a user with RESOURCE or DBA authority.
2. Execute a CREATE SEQUENCE statement. See "ALTER SEQUENCE statement" [*SQL Anywhere Server - SQL Reference*].

Drop a sequence

To drop a sequence (Sybase Central)

1. Use the SQL Anywhere 12 plug-in to connect to the database as a user with RESOURCE or DBA authority.
2. Right-click a sequence generator and then choose **Delete**.

To drop a sequence (SQL)

1. Connect to the database as a user with RESOURCE or DBA authority.
2. Execute a DROP SEQUENCE statement. See "DROP SEQUENCE statement" [*SQL Anywhere Server - SQL Reference*].

Data definition statements and concurrency

Data definition statements that change an entire table, such as CREATE INDEX, ALTER TABLE, and TRUNCATE TABLE, are prevented whenever the table on which the statement is acting is currently being used by another connection. These data definition statements can be time consuming and the database server will not process requests referencing the same table while the command is being processed.

The CREATE TABLE statement does not cause any concurrency conflicts.

The GRANT statement, REVOKE statement, and SET OPTION statement also do not cause concurrency conflicts. These commands affect any new SQL statements sent to the database server, but do not affect existing outstanding statements.

GRANT and REVOKE for a user are not allowed if that user is connected to the database.

Data definition statements and synchronized databases
Using data definition statements in databases using synchronization requires special care. See "MobiLink - Server Administration" and "Data definition statements" [*SQL Remote*].

Summary

Transactions and locking are second only in importance to relations between tables. The integrity and performance of any database can benefit from the judicious use of locking and careful construction of transactions. Both are essential to creating databases that must execute a large number of commands concurrently.

Transactions group SQL statements into logical units of work. To complete transactions, you can either roll back all the changes you made, or commit the changes to make them permanent.

In the event of system failure, transactions are essential to data recovery. They also play a pivotal role in interweaving statements from concurrent transactions.

To improve performance, multiple transactions must be executed concurrently. Each transaction is composed of component SQL statements. When two or more transactions are executed concurrently, the database server must schedule the execution of the individual statements. Unlike sequentially executed transactions, concurrent transactions could introduce inconsistencies.

Four types of inconsistencies are used to define isolation levels:

- **Dirty read** One transaction reads data modified, but not yet committed, by another.
- **Non-repeatable read** A transaction reads the same row twice and gets different values.
- **Phantom row** A transaction selects rows, using a certain criterion, twice and finds new rows in the second result set.

- **Lost update** One transaction's changes to a row are completely lost because another transaction is allowed to save an update based on earlier data.

A schedule is called serializable whenever the effect of executing the statements according to the schedule is the same as could be achieved by executing each of the transactions sequentially. Schedules are said to be **correct** if they are serializable. A serializable schedule will cause none of the above inconsistencies.

Locking controls the amount and types of interference permitted. SQL Anywhere provides you with four levels of locking: isolation levels 0, 1, 2, and 3. At the highest isolation, level 3, SQL Anywhere guarantees that the schedule is serializable, meaning that the effect of executing all the transactions is equivalent to running them sequentially.

Unfortunately, locks acquired by one transaction may impede the progress of other transactions. Because of this problem, lower isolation levels are desirable whenever the inconsistencies they may allow are tolerable. Increased isolation to improve data consistency frequently means lowering the concurrency, the efficiency of the database at processing concurrent transactions. You must frequently balance the requirements for consistency against the need for performance to determine the best isolation level for each operation.

Conflicting locking requirements between different transactions may lead to blocking or deadlock. SQL Anywhere contains mechanisms for dealing with both these situations, and provides you with options to control them.

Transactions at higher isolation levels do not, however, *always* impact concurrency. Other transactions will be impeded only if they require access to locked rows. You can improve concurrency through careful design of your database and transactions. For example, you can shorten the time that locks are held by dividing one transaction into two shorter ones, or you might find that adding an index allows your transaction to operate at higher isolation levels with fewer locks.

Monitoring and improving database performance

This section describes how to perform database and application profiling activities, monitor and improve performance, and troubleshoot specific performance problems.

Improving database performance

To improve database performance, you must determine if the existing database is performing at optimum levels. This section provides information about using SQL Anywhere analysis tools to analyze and correct database performance.

SQL Anywhere provides several diagnostic tools for the detection of production database performance issues. Most of the tools rely on the **diagnostic tracing** infrastructure; a system of tables, files, and other components that capture and store diagnostic data. You can use diagnostic tracing data to perform diagnostic and monitoring tasks such as **application profiling**.

There are several methods for analyzing SQL Anywhere performance data including:

- **The Application Profiling Wizard** This wizard, available from Application profiling mode in Sybase Central, provides a fully-automated method of checking performance. At the end of the wizard, improvement recommendations are provided. See "Application profiling" on page 158.

- **The Database Tracing Wizard** This wizard, available from Application Profiling mode in Sybase Central, provides the ability to customize the type of performance data gathered. This allows you to monitor the performance of specific users or activities. See "Advanced application profiling using diagnostic tracing" on page 169.

- **Request trace analysis** This feature allows you to narrow diagnostic data gathering to requests (statements) issued by specific users or connections. See "Perform request trace analysis" on page 185.

- **Index Consultant** This feature analyzes the indexes in the database and provides recommendations for improvement. You can access this tool through Application Profiling mode, or as a standalone tool. See "Index Consultant" on page 164.

- **Procedure profiling** This feature allows you to determine how long it takes procedures, user-defined functions, events, system triggers, and triggers to execute. Procedure profiling is available as a feature in Sybase Central. See "Procedure profiling in Application Profiling mode" on page 160.

  You can also use system procedures to implement procedure profiling. See "Procedure profiling using system procedures" on page 189.

- **Execution plans** This feature allows you to use an execution plan to access information in the database related to a statement. You can view the execution plan in Interactive SQL or use SQL

functions. You can retrieve an execution plan in several different formats and the plan can be saved. See "Reading execution plans" on page 592.

Note
In the documentation, the terms **application profiling** and **diagnostic tracing** are used interchangeably. Diagnostic tracing is advanced application profiling.

Application profiling

Application profiling generates data that you can use to understand how applications interact with the database and to identify and eliminate performance problems. Two methods are available for generating profiling information; an automated method, using the **Application Profiling Wizard**, or using the tools and features found in Application Profiling mode of Sybase Central.

The **Application Profiling Wizard** is not supported on Windows Mobile; however, the **Database Tracing Wizard** is. You cannot automatically create a tracing database from a Windows Mobile device, and you cannot trace to the local database on a Windows Mobile device. You must trace from the Windows Mobile device to a copy of the Windows Mobile database running on a database server on a desktop computer.

- **Automated application profiling** Use the **Application Profiling Wizard** in Sybase Central to identify common performance problems. The wizard allows you to define the types of activities to profile and provides recommendations for improving database performance when it is complete. The Index Consultant has also been integrated into the **Application Profiling Wizard** and uses the data to recommend index improvements.

  An automated approach is ideal for environments with few database connections, or where sophisticated profiling is not required.

- **Advanced application profiling using diagnostic tracing** Use the **Database Tracing Wizard** to customize the data returned during a tracing session and where it is stored. You can also use the command line to return and store customized tracing data. You can control the activities profiled, and target specific issues. For example, you can target specific statements executed by the database server, query plans used, deadlocks, connections that block each other, and performance statistics.

  An advanced approach is recommended for environments in which the database has a high workload, or where sophisticated profiling is required to diagnose a problem. By customizing the tracing session, you can reduce the tracing scope to specific activities, and you can direct tracing data to a remotely located database. Both of these actions reduce the workload on the database being profiled.

  See "Advanced application profiling using diagnostic tracing" on page 169.

Application Profiling Wizard

The **Application Profiling Wizard** in Sybase Central provides an automated method of performing a diagnostic tracing session for profiling applications. The wizard gathers data on how your applications are

interacting with the database, provides you access to the data, and with indexing recommendations, if any. See "Application Profiling Wizard" on page 158.

When you use the **Application Profiling Wizard** in Sybase Central, the wizard automatically creates a tracing database with the same name you specify in the wizard for the analysis file. For more information about the database files created for application profiling and diagnostic tracing, see "Tracing session data" on page 169.

The **Application Profiling Wizard** cannot be used to create a tracing session for a database running on Windows Mobile. You must use the **Database Tracing Wizard**. See "Create a diagnostic tracing session" on page 180.

To disable the **Application Profiling Wizard** from starting automatically when switching to **Application Profiling** mode, on the first page of the wizard select **In The Future, Do Not Show This Wizard After Switching To Application Profiling Mode**. You can also suppress the first page of the wizard by selecting **In The Future, Do Not Show This Page**. To change these options at any time, choose **Tools** » **SQL Anywhere 12** » **Preferences**, select the **Utilities** tab, and then select the appropriate options.

To use the Application Profiling Wizard (Sybase Central)

1. Use the SQL Anywhere 12 plug-in to connect to the database as a user with DBA or PROFILE authority.
2. Choose **Mode** » **Application Profiling**.

   If the **Application Profiling Wizard** does not appear, choose **Application Profiling** » **Open Application Profiling Wizard**.
3. Follow the **Application Profiling Wizard** instructions. Do not click **Finish**; this ends profiling, and closes the wizard.

   The wizard:

   - creates a local database to hold diagnostic tracing information
   - starts the network server
   - starts a tracing session
   - prompts you to run the application you would like to profile
4. Return to the **Application Profiling Wizard** and click **Finish**. When the wizard finishes, it returns its findings and allows you to review the data it gathered during the tracing session.

For more information about the indexing recommendations returned from the **Application Profiling Wizard**, see "Understanding Index Consultant recommendations" on page 166.

For more information about the procedure profiling information gathered during the tracing session, see "How to read procedure profiling results" on page 163.

See also

- "PROFILE authority" [*SQL Anywhere Server - Database Administration*]

Procedure profiling in Application Profiling mode

This section explains how to use the Application Profiling mode in Sybase Central to perform procedure profiling. It is the recommended method for accessing procedure profiling results. However, you can also use SQL commands to perform procedure profiling. See "Procedure profiling using system procedures" on page 189.

Procedure profiling shows you how long it takes your procedures, user-defined functions, events, system triggers, and triggers to execute. You can also view line-by-line execution times for these objects, once they have run during profiling. Then, using the information provided in the procedure profiling results, you can determine which objects should be fine-tuned to improve performance within your database.

Procedure profiling can also help you analyze specific database procedures (including stored procedures, functions, events and triggers) found to be expensive via request logging. It can also help you discover expensive hidden procedures, for example, triggers, events, and nested stored procedure calls. As well, it can help pin-point potential problem areas within the body of a procedure.

Procedure profiling results are stored in memory by the database server. Profiling information is cumulative, and accurate to 1 ms.

Enable procedure profiling

Once procedure profiling is enabled, the database server gathers profiling information until you disable profiling or until the database server is shut down.

Note
All profiling information is deleted when the database server is shut down. To export profiling information, use the sa_procedure_profile system procedure. See "sa_procedure_profile system procedure" [*SQL Anywhere Server - SQL Reference*].

You cannot use SQL statements to query profiling information retained by the database server. Profiling information is kept in in-memory database server data structures.

To enable procedure profiling (Sybase Central)

1. Use the SQL Anywhere 12 plug-in to connect to the database as a user with DBA or PROFILE authority.
2. In the left pane, select the database.
3. Choose **Mode** » **Application Profiling**.

   If the **Application Profiling Wizard** does not appear, choose **Application Profiling** » **Open Application Profiling Wizard**.
4. Follow the **Application Profiling Wizard** instructions.

5. On the **Profiling Options** page, select **Stored Procedure, Function, Trigger, Or Event Execution Time**.

6. Click **Finish**.

   If you switch to another mode, a prompt appears asking whether you want to stop collecting procedure profiling information. Select **No** to continue working in other modes while profiling continues.

See also

- "Reset procedure profiling" on page 161
- "Disable procedure profiling" on page 161
- "Analyze procedure profiling results" on page 162
- "PROFILE authority" [*SQL Anywhere Server - Database Administration*]

Reset procedure profiling

Reset procedure profiling when you want to clear existing profiling information about procedures, functions, events, and triggers. Resetting does not stop procedure profiling if it is enabled, nor does it start procedure profiling if it is disabled.

To reset profiling (Sybase Central)

1. Use the SQL Anywhere 12 plug-in to connect to the database as a user with DBA or PROFILE authority.

2. In the left pane, select the database.

3. Choose **Mode** » **Application Profiling**. If the **Application Profiling Wizard** appears, click **Cancel**.

4. If procedure profiling is enabled: in the **Application Profiling Details** pane, click the database and then click **View Profiling Settings On Selected Databases**.

   If procedure profiling is not enabled, in the left pane, right-click the database and choose **Properties**.

5. Click the **Profiling Settings** tab.

6. Click **Reset Now**.

7. Click **OK**.

See also

- "Enable procedure profiling" on page 160
- "Disable procedure profiling" on page 161
- "Analyze procedure profiling results" on page 162

Disable procedure profiling

When you are finished capturing profiling information for procedures, triggers, and functions, you can disable procedure profiling. When you disable procedure profiling, you also have the option to delete the profiling information gathered so far. You may want to do this if you have already completed your analysis work.

If you do not choose to delete profiling data, it remains available for review in Application Profiling mode in Sybase Central, even after procedure profiling is disabled.

To disable profiling without deleting profiling information (Sybase Central)

1. Use the SQL Anywhere 12 plug-in to connect to the database as a user with DBA or PROFILE authority.
2. In the left pane, select the database.
3. Choose **Mode** » **Application Profiling**. If the **Application Profiling Wizard** appears, click **Cancel**.
4. In the **Application Profiling Details** pane, click **Stop Collecting Profiling Information On Selected Databases**.

To delete profiling information and disable procedure profiling (Sybase Central)

1. Use the SQL Anywhere 12 plug-in to connect to the database as a user with DBA or PROFILE authority.
2. In the left pane, select the database.
3. Choose **Mode** » **Application Profiling**. If the **Application Profiling Wizard** appears, click **Cancel**.
4. In the **Application Profiling Details** pane, select the database and click **View Profiling Settings On Selected Databases**.
5. Click the **Profiling Settings** tab.
6. Click **Clear Now**.
7. Click **OK**.

See also

- “Enable procedure profiling” on page 160
- “Reset procedure profiling” on page 161
- “Analyze procedure profiling results” on page 162

Analyze procedure profiling results

Even though it is called procedure profiling, you are actually able to view profiling results for stored procedures, user-defined functions, triggers, system triggers, and events in your database.

To view procedure profiling information (Sybase Central)

1. Use the SQL Anywhere 12 plug-in to connect to the database as a user with DBA authority.
2. Enable procedure profiling. See "Enable procedure profiling" on page 160.
3. In the left pane, double-click one of the following: **Triggers**, **System Triggers**, **Procedures & Functions**, or **Events**.
4. In the right pane, click the **Profiling Results** tab.

   A list appears of all the objects of the selected type that have executed since you enabled procedure profiling.

   An expected object might be missing because it has not been executed. Or, it may have executed but the results have not yet been refreshed. Press F5 to refresh the list.

   If you find more objects listed than you expected, one object can call other objects, so there may be more items listed than those that users explicitly called.
5. To view in-depth profiling results for a specific object, double-click the object on the **Profiling Results** tab.

   The right pane details are replaced with in-depth profiling information for the object.

How to read procedure profiling results

The **Profiling Results** tab provides a summary of the profiling information for all the objects, grouped by type, that have been executed within the database since you started procedure profiling. The information displayed includes:

Column	Description
Name	The name of the object.
Owner	The owner of the object.
Table or Table Name	The table a trigger belongs to (this column only appears on the database Profile tab).
Event	The type of object, for example, a procedure.
Type	The type of trigger for system triggers. This can be Update or Delete.
# Execs.	The number times each object has been called.
# msec.	The total execution time for each object.

These columns, and their content, may vary depending on the type of object.

When you double-click a specific object, such as a procedure, details specific to that object appears in the **Profiling Results** tab. The information displayed includes:

Column	Description
Execs	The number of times the line of code in the object was executed.
Milliseconds	The total amount of time that a line took to execute.
%	The percent of total time that a line took to execute.
Line	The line number within the object.
Source	The code that was executed.

Lines with long execution times compared to other lines in the code should be analyzed to see whether there is a more efficient way to achieve the same functionality. You must be connected to the database, have profiling enabled, and have DBA authority to access procedure profiling information.

Index Consultant

You must have DBA or PROFILE authority to run the Index Consultant.

The selection of a proper set of indexes can improve database performance. The SQL Anywhere Index Consultant helps you select indexes by providing recommendations on the best set of indexes for your database.

You can run the Index Consultant against a single query using Interactive SQL, or against the database using Application Profiling mode in Sybase Central. When analyzing a database, the Index Consultant uses a tracing session to gather data and make recommendations. It estimates query execution costs using those indexes to see which indexes lead to improved execution plans. The Index Consultant evaluates multiple column indexes, single-column indexes, and investigates the impact of clustered or unclustered indexes.

The Index Consultant analyzes a database or single query by generating candidate indexes and determining their effect on performance. To explore the effect of different candidate indexes, the Index Consultant repeatedly re-optimizes the queries under different sets of indexes. It does not execute the queries.

Note
You can use Sybase Central to connect to a version 9 database server. However, the layout of windows in Sybase Central reverts to the version 9 layout, which does not include Application Profiling mode. Refer to your version 9 documentation for information about locating and using the Index Consultant in Sybase Central.

See also

- "Working with indexes" on page 56
- "Indexes" on page 623
- "Application profiling" on page 158
- "PROFILE authority" [*SQL Anywhere Server - Database Administration*]
- "Application profiling" on page 158
- "Understanding Index Consultant recommendations" on page 166

Obtain Index Consultant recommendations for a query

To obtain Index Consultant recommendations for a query (Sybase Central)

1. Use the SQL Anywhere 12 plug-in to connect to the database as a user with DBA or PROFILE authority.
2. Right-click the database and choose **Open Interactive SQL**.
3. In the **SQL Statements** pane, type the query.
4. Choose **Tools** » **Index Consultant**.

Obtain Index Consultant recommendations for a database

To obtain Index Consultant recommendations for an entire database, use the Application Profiling mode in Sybase Central. The Index Consultant needs profiling data before it can made its recommendations. The following procedure is a quick way to gather data and obtain the recommendations using data gathered by the **Application Profiling Wizard**. However, if you already have application profiling data (for example, if you profiled your database already using the **Database Tracing Wizard**), you can also run the Index Consultant on the tracing database that you created.

To obtain Index Consultant recommendations for a database using Application Profiling Wizard (Sybase Central)

1. Use the SQL Anywhere 12 plug-in to connect to the database as a user with DBA or PROFILE authority.
2. Choose **Mode** » **Application Profiling**.
3. Follow the **Application Profiling Wizard** instructions.

   If the **Application Profiling Wizard** does not appear, choose **Application Profiling** » **Open Application Profiling Wizard**, and follow the wizard instructions until it completes.
4. In Sybase Central, choose **Application Profiling** » **Run Index Consultant On Tracing Database**.
5. Follow the **Index Consultant Wizard** instructions.

Understanding Index Consultant recommendations

Before analyzing a tracing session, the Index Consultant asks you for the type of recommendations you want:

- **Recommend clustered indexes** If this option is selected, the Index Consultant analyzes the effect of clustered and unclustered indexes.

  Properly selected clustered indexes can provide significant performance improvements over unclustered indexes for some workloads, but you must reorganize the table (using the REORGANIZE TABLE statement) for them to be effective. In addition, the analysis takes longer if the effects of clustered indexes are considered. See "Using clustered indexes" on page 58.

- **Keep existing secondary indexes** The Index Consultant can perform its analysis by either maintaining the existing set of secondary indexes in the database, or by ignoring the existing secondary indexes. A secondary index is an index that is not a unique constraint or a primary or foreign key. Indexes that are present to enforce referential integrity constraints are always considered when selecting access plans.

The analysis includes the following steps:

- **Generate candidate indexes** For each tracing session, the Index Consultant generates a set of candidate indexes. Creating a real index on a large table can be a time consuming operation, so the Index Consultant creates its candidates as virtual indexes. A virtual index cannot be used to actually execute queries, but the optimizer can use virtual indexes to estimate the cost of execution plans as if such an index were available. Virtual indexes allow the Index Consultant to perform "what-if" analysis without the expense of creating and managing real indexes. Virtual indexes have a limit of four columns.

- **Testing the benefits and costs of candidate indexes** The Index Consultant asks the optimizer to estimate the cost of executing the queries in the tracing database, with and without different combinations of candidate indexes.

- **Generating recommendations** The Index Consultant assembles the results of the query costs and sorts the indexes by the total benefit they provide. It provides a SQL script, which you can run to implement the recommendations or which you can save for your own review and analysis.

Understanding Index Consultant results

The Index Consultant provides a set of tabs with the results of a given analysis. You can save the results of an analysis for later review.

Summary tab

The **Summary** tab provides an overview of the analysis, including the number of queries, the number of recommended indexes, the number of pages required for the recommended indexes, and the benefit that the recommended indexes are expected to yield. The benefit number is measured in internal units of cost.

Recommended Indexes tab

The **Recommended Indexes** tab contains data about each of the recommended indexes. The information provided includes:

- **Clustered** Each table can have a single clustered index. A clustered index can sometimes provide significantly more benefit than an unclustered index. See "Using clustered indexes" on page 58.
- **Pages** The estimated number of database pages required to hold the index if you choose to create it. See "Table and page sizes" on page 622.
- **Relative Benefit** A number from one to ten, indicating the estimated overall benefit of creating the specified index. A higher number indicates a greater benefit.

  The relative benefit is computed using an internal algorithm, separately from the Total Cost Benefit column. There are several factors included in estimating the relative benefit that do not appear in the total cost benefit. For example, it can happen that the presence of one index dramatically affects the benefits associated with a second index. In this case, the relative benefit attempts to estimate the separate impact of each index.

  For more information, see "Implementing Index Consultant results" on page 168.
- **Total Benefit** The cost decrease associated with the index, summed over all operations in the tracing session, measured in internal units of cost (the cost model). See "How the optimizer works" on page 540.
- **Update Cost** Adding an index introduces cost, both in additional storage space and in extra work required when data is modified. The Update Cost column is an estimate of the additional maintenance cost associated with an index. It is measured in internal units of cost.
- **Total Cost Benefit** The total benefit minus the update cost associated with the index.

Requests tab

The **Requests** tab provides a breakdown of the impact of the recommendations for individual requests within the tracing session. The information includes the estimated cost before and after applying the recommended indexes, and the virtual indexes used by the query. A button allows you to view the best execution plan found for the request.

Updates tab

The **Updates** tab provides a breakdown of the impact of the recommendations.

Unused Indexes tab

The **Unused Indexes** tab lists indexes that already exist in the database that were not used in the execution of any requests in the tracing session. Only secondary indexes are listed: that is, neither indexes on primary keys and foreign keys nor unique constraints are listed.

Log tab

The **Log** tab lists activities that have been completed for this analysis.

See also

- "Working with indexes" on page 56
- "Indexes" on page 623
- "Application profiling" on page 158

Implementing Index Consultant results

Although the Index Consultant provides a SQL script that you can run to implement its results, you may want to assess the results before implementing them. For example, you may want to rename the proposed index names generated during the analysis.

When assessing the results, consider the following:

- **Do the proposed indexes match your expectations?** If you know the data in your database well, and you know the queries being run against the database, you may want to check the usefulness of the proposed indexes against your own knowledge. Perhaps a proposed index only affects a single query that is run rarely, or perhaps it is on a small table and makes relatively little overall impact. Perhaps an index that the Index Consultant suggests should be dropped is used for some other task that was not included in your tracing session.

- **Are there strong correlations between the effects of proposed indexes?** The index recommendations attempt to evaluate the relative benefit of each index separately. However, two indexes are of use only if both exist (a query can use both if they exist, and none if either is missing). You can study the **Requests** tab and inspect the query plans to see how the proposed indexes are being used.

- **Are you able to reorganize a table when creating a clustered index?** To take full advantage of a clustered index, you should reorganize the table on which it is created using the REORGANIZE TABLE statement. If the Index Consultant recommends many clustered indexes, you may need to unload and reload your database to get the full benefit. Unloading and reloading tables can be a time-consuming operation and can require large disk space resources. You may want to confirm that you have the time and resources you need to implement the recommendations.

- **Do the server and connection state during the analysis reflect a realistic state during product operation?** The results of the analysis depend on the state of the database server, including which data is in the cache. They also depend on the state of the connection, including some database option settings. As the analysis creates only virtual indexes, and does not execute requests, the state of the database server is essentially static during the analysis (except for changes introduced by other connections). If the state does not represent the typical operation of your database, you may want to rerun the analysis under different conditions.

See also

- "Understanding Index Consultant recommendations" on page 166
- "Using SQL command files" on page 743
- "Working with indexes" on page 56
- "Indexes" on page 623
- "REORGANIZE TABLE statement" [*SQL Anywhere Server - SQL Reference*]
- "Application profiling" on page 158

Advanced application profiling using diagnostic tracing

Diagnostic tracing is an advanced method of application profiling. The diagnostic tracing data produced by the database server can include the time stamps and connection ids of statements handled by the database server. For queries, diagnostic tracing data includes the isolation level, number of rows fetched, cursor type, and query execution plan. For INSERT, UPDATE, and DELETE statements, the number of rows affected is also included. You can also use diagnostic tracing to record information about locking and deadlocks, and to capture numerous performance statistics.

You use the data gathered during diagnostic tracing to perform in-depth application profiling activities such as identifying and troubleshooting:

- specific performance problems
- statements that are unusually slow to execute
- improper option settings
- circumstances that cause the optimizer to pick a sub-optimal plan
- contention for resources (CPUs, memory, disk I/O)
- application logic problems

Tracing data is also used by tools such as the Index Consultant to make specific recommendations on how to change your database or application to improve performance.

The tracing architecture is robust and scalable. It can record all the information that request logging records, and details to support tailored analysis. For information about request logging, see "Perform request trace analysis" on page 185.

See also

- "Application profiling" on page 158

Tracing session data

Diagnostic tracing data is gathered during a **tracing session**. Three methods are available to capture tracing session data:

- the **Database Tracing Wizard** in Sybase Central
- transparently, as part of the automated activities of the **Application Profiling Wizard**
- the ATTACH TRACING and DETACH TRACING statements

When a tracing session is in progress, SQL Anywhere generates diagnostic information for the specified database. The amount of tracing data generated depends on the tracing settings. For more information about how to configure the amount and type of tracing data generated, see "Configuring diagnostic tracing" on page 171.

The database being profiled is either referred to as the **production database**, the source database, or the database being profiled. The database into which the tracing data is stored is referred to as the **tracing database**. The production and tracing database can be the same database. However, to avoid increasing the size of the production database, it is recommended that you store tracing data in a separate database. The size of database files cannot be reduced after they have grown. Also, the production database performs better if the overhead for storing and maintaining tracing data is performed in another database, especially if the production database is large and heavily used.

The tables in the tracing database that hold the tracing data are referred to as the **diagnostic tracing tables**. These tables are owned by dbo. For more information about these tables, see "Diagnostic tracing tables" [*SQL Anywhere Server - SQL Reference*].

Note
The **Application Profiling Wizard** is not supported on Windows Mobile; however, the **Database Tracing Wizard** is. As well, you must trace from the Windows Mobile device to a copy of the Windows Mobile database running on a database server on a desktop computer. You cannot automatically create a tracing database from a Windows Mobile device, and you cannot trace to the local database on a Windows Mobile device.

Files created during a tracing session

The files created and used for a tracing session differ depending on whether you use the **Application Profiling Wizard**, or the **Database Tracing Wizard**.

When you run the **Application Profiling Wizard**, the wizard silently captures a tracing session behind the scenes, creating the tracing database to hold the diagnostic tables. This external database is created using the name and location you specify in the wizard, and it has the extension *.adb*. The wizard also creates an analysis log file in the same directory as the tracing database, using the same name but with the extension *.alg*. This analysis log file contains the results of the analysis work done by the wizard, and can be opened at any time in a text editor.

When you are finished with the data generated by the **Application Profiling Wizard**, you can delete the tracing database and analysis log file associated with the session.

When you create a tracing session using the **Database Tracing Wizard**, the wizard asks you to choose whether to save tracing data internally, in the production database, or externally, in a separate database (for example, *tracingData.db*). Creating an external tracing database is recommended. See "Creating an external tracing database" on page 186.

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.

Configuring diagnostic tracing

You cannot change the preconfigured tracing settings of the **Application Profiling Wizard** in Sybase Central. However, you can use the **Database Tracing Wizard** to configure almost all aspects of your tracing activities. Use one of the following methods to configure diagnostic tracing settings:

- use the **Database Tracing Wizard** in Sybase Central. This method is recommended because it allows you to see all the tracing settings that are in effect. See "Change the diagnostic tracing configuration settings" on page 179.
- use system procedures to change settings stored in the diagnostic tracing tables. For more information about the system procedures used to administer application profiling, see "sa_set_tracing_level system procedure" [*SQL Anywhere Server - SQL Reference*] and "sa_save_trace_data system procedure" [*SQL Anywhere Server - SQL Reference*].

Tracing settings are stored in the sa_diagnostic_tracing_level system table. See "sa_diagnostic_tracing_level table" [*SQL Anywhere Server - SQL Reference*].

The SendingTracingTo and ReceivingTracingFrom database properties identify the tracing and production databases, respectively. For more information about these properties, see "Database properties" [*SQL Anywhere Server - Database Administration*].

Choosing a diagnostic tracing level

Diagnostic tracing settings are grouped into several levels, but you can also customize the settings further within these levels. The types of information gathered at the various levels are referred to as **diagnostic tracing types**. Following are descriptions of the levels you can specify, and the diagnostic tracing types they include. For a description of the diagnostic tracing types mentioned below, see "Diagnostic tracing types" on page 174.

Customizing diagnostic tracing settings allows you to reduce the amount of unwanted tracing data in the diagnostic tracing session. For example, suppose that user AliceB has been complaining that her application has been running slowly, yet the rest of the users are not experiencing the same problem. You now want to know exactly what is going on with AliceB's queries. This means you should gather the list of all queries and other statements that AliceB runs as part of her application, and any query plans for long running queries. To do this, you could just set the diagnostic tracing level to 3 and generate tracing data for a day or two. However, since this level can significantly impact performance for other users, you should limit the tracing to just AliceB's activities. To do this, you set the diagnostic tracing level to 3, and then customize the scope of the diagnostic tracing to be USER, and specify AliceB as the user name. Allow the diagnostic tracing session to run for a couple of hours, and then examine the results.

The recommended method for customize diagnostic tracing settings is using the **Database Tracing Wizard**. See "Change the diagnostic tracing configuration settings" on page 179.

You can also use the sa_set_tracing_level system procedure; however, you cannot make as many customizations using this approach. See also "sa_set_tracing_level system procedure" [*SQL Anywhere Server - SQL Reference*].

As a good practice, you should not change diagnostic tracing settings while a tracing session is in progress because it makes interpreting the data more difficult. However, it is possible to do so. See "Change diagnostic tracing settings when a tracing session is in progress" on page 180.

Diagnostic tracing levels

The following is a list of diagnostic tracing levels specified in the **Database Tracing Wizard**. For a description of the various diagnostic tracing types, see "Diagnostic tracing types" on page 174.

Estimated impacts to performance reflect the assumption that tracing data is sent to a tracing database on another database server (recommended).

- **Level 0** This level keeps the tracing session running, but does not send any tracing data to the tracing tables.
- **Level 1** Performance counters and a sampling of executed statements (once every five seconds) are gathered. For this level, the diagnostic tracing types include:
  - volatile_statistics, with sampling every 1 second
  - nonvolatile_statistics, with sampling every 60 seconds

  This level has a negligible impact on performance.
- **Level 2** This level gathers performance counters, a sampling of executed plans (once every five seconds), and records all executed statements. For this level, the diagnostic tracing types include:
  - volatile_statistics, with sampling every 1 second
  - nonvolatile_statistics, with sampling every 60 seconds
  - statements
  - plans, sampling every 5 seconds

  This level has a medium impact on performance—up to, but not more than, a 20% overhead.
- **Level 3** This level records the same details as Level 2 but with more frequent plan samples (once every 2 seconds) and detailed blocking and deadlock information. For this level, the diagnostic tracing types include:
  - volatile_statistics, with sampling every 1 second

- nonvolatile_statistics, with sampling every 60 seconds
- statements
- blocking
- deadlock
- statements_with_variables
- plans, with sampling every 2 seconds

This level has the greatest impact on performance—greater than 20% overhead.

Diagnostic tracing scopes

Following is the list of **scopes** for diagnostic tracing. Scope values can be used to limit tracing to who (or what) is causing the activity in the database. For example, you can set the scope to trace requests coming from a specified connection. Scope values are stored in the scope column of the dbo.sa_diagnostic_tracing_level diagnostic table, and may have corresponding arguments, typically an identifier such as an object name or user name, which are stored in the identifier column. The values in the scope column reflect the settings specified in the **Database Tracing Wizard**.

Values in the scope column	Description
DATABASE	Records tracing data for any event occurring within the database, assuming the event corresponds to the specified level and condition. Used for long-term background monitoring of the database, or for short-term diagnostics, when it is necessary to determine the source of costly queries. There is no identifier to specify when you specify DATABASE.
ORIGIN	Records tracing data for the queries originating from either outside or inside the database. There are two possible identifiers you can specify when specifying the scope ORIGIN: External or Internal. External specifies to log the statement text and associated details for queries that come from outside the database server, and that correspond to the specified level and condition. Internal specifies to log the same information for queries that come from within the database server, and that correspond to the level and condition specified.

Values in the scope column	Description
USER	Records tracing data only for the queries issued by the specified user, and by connections created by the specified user. This scope is used to diagnose problematic queries originating from a particular user. The identifier for this scope is the name of the user for whom the tracing is to be performed.
CONNECTION_NAME, or CONNECTION_NUMBER	Records tracing data only for the statements executed by the current connection. These scopes are used when the user has multiple connections, one of which is executing costly statements. The identifier for this scope is the name of the connection, or the connection number, respectively.
FUNCTION, PROCEDURE, EVENT, TRIGGER, or TABLE	Records tracing data for the statements that use the specified object. If the object references other objects, all the data for those objects is recorded as well. For example, if tracing is being done for a procedure that uses a function which, in turn, triggers an event, statements for all three objects are logged, providing they correspond to the specified level and condition provided for logging. Used when use of a specific object is costly, or when the statements that reference the object take an unusually long time to finish. The TABLE scope is used for tables, materialized views, and non-materialized views. The identifier for this scope is the fully qualified name of the object.

See also

- "Diagnostic tracing types" on page 174
- "Diagnostic tracing conditions" on page 177

Diagnostic tracing types

The following table lists the tracing **types** you can choose for diagnostic tracing. Each diagnostic tracing type requires a corresponding condition, as noted below, and is stored in the trace_type column of the dbo.sa_diagnostic_tracing_level diagnostic table, and may have corresponding diagnostic tracing conditions, which are stored in the trace_condition column. For a list of all possible conditions, see "Diagnostic tracing conditions" on page 177.

The values in trace_type column reflect the settings specified in the **Database Tracing Wizard**.

Value in the trace_type column	Description
VOLATILE_STATISTICS	Collects a sample of frequently changing database and server statistics. Scopes and conditions: This diagnostic tracing type requires the DATABASE scope, and uses the SAMPLE_EVERY condition as the interval at which to collect the data. See "Diagnostic tracing scopes" on page 173, and "Diagnostic tracing conditions" on page 177.
NONVOLATILE_STATISTICS	Collects a sample of database and server statistics that do not change frequently. Non-volatile statistics cannot be collected more frequently then volatile statistics. Volatile statistics must be collected in order for non-volatile statistics to be collected, and the time difference between the sampling for non-volatile statistics should be a multiple of the time difference specified for the volatile statistics. Scopes and conditions: This diagnostic tracing type requires the DATABASE scope, and uses the SAMPLE_EVERY condition as the interval at which to collect the data. See "Diagnostic tracing scopes" on page 173, and "Diagnostic tracing conditions" on page 177.
CONNECTION_STATISTICS	Collects a sample of connection statistics. If the scope is database, statistics for all connections to the database are collected. If the scope is user, statistics for all connections for the specified user are collected. If the scope is CONNECTION_NAME or CONNECTION_NUMBER, only statistics for the specified connection are collected. Volatile statistics have to be collected in order for CONNECTION_STATISTICS to be collected, and the time interval between sampling should be a multiple of that specified for the VOLATILE_STATISTICS. Scopes and conditions: This diagnostic tracing type can be used with the DATABASE, USER, CONNECTION_NUMBER, and CONNECTION_NAME scopes, and uses the SAMPLE_EVERY condition as the interval at which to collect the data. See "Diagnostic tracing scopes" on page 173, and "Diagnostic tracing conditions" on page 177.
BLOCKING	Collects information about blocks according to the specified scope and condition. If the scope is CONNECTION_NAME or CONNECTION_NUMBER, then the block may be recorded when the connection blocks another connection, or is blocked by another connection. Scopes and conditions: This diagnostic tracing type can be used with all the scopes, and can use any one of the following conditions for collection: NONE, NULL, SAMPLE_EVERY. See "Diagnostic tracing scopes" on page 173, and "Diagnostic tracing conditions" on page 177.

Value in the trace_type column	Description
PLANS	Collects execution plans for queries, depending on the condition and scope. Scopes and conditions: This diagnostic tracing type can be used with all the scopes, and can use any one of the following conditions for collection: NONE, NULL, SAMPLE_EVERY, and ABSOLUTE_COST. See "Diagnostic tracing scopes" on page 173, and "Diagnostic tracing conditions" on page 177.
PLANS_WITH_STATISTICS	Collects plans with execution statistics. Plans are recorded at cursor close time. If the RELATIVE_COST_DIFFERENCE condition is specified, part of the statistics in the output might be best-guess statistics. Scopes and conditions: This diagnostic tracing type can be used with all the scopes, and accepts any one of the conditions for collection.
STATEMENTS	Collects SQL statements for the specified scope and condition. Internal variables are collected the first time each procedure is executed. This diagnostic tracing type is automatically included if the STATEMENTS_WITH_VARIABLES, PLANS, PLANS_WITH_STATISTICS, OPTIMIZATION_LOGGING, or OPTIMIZATION_LOGGING_WITH_PLANS diagnostic tracing type is specified. Scopes and conditions: This diagnostic tracing type can be used with all the scopes, and can use any one of the conditions for collection. See "Diagnostic tracing scopes" on page 173, and "Diagnostic tracing conditions" on page 177.
STATEMENTS_WITH_VARIABLES	Collects SQL statements and the variables attached to the statements. For each variable, either internal or host, all the values that were assigned are collected as well. Scopes and conditions: This diagnostic tracing type can be used with all the scopes, and can use any one of the conditions for collection. See "Diagnostic tracing scopes" on page 173, and "Diagnostic tracing conditions" on page 177.

Value in the trace_type column	Description
OPTIMIZATION_LOGGING	Collects data about join strategies considered by the optimizer for execution of each query. Information about cost of execution of each strategy, and the basic information necessary to reconstruct the tree for the structure, is collected. Information about rewrites applied to the query is also collected. If a scope other than DATABASE, CONNECTION_NAME, CONNECTION_NUMBER, ORIGIN, or USER is used, the first recorded statement text might be different than the initial text of the query since some rewrites can be applied before it can be determined that optimization logging should be applied to the current statement. This diagnostic tracing type is automatically added whenever the OPTIMIZATION_LOGGING_WITH_PLANS tracing type is specified. This diagnostic tracing type corresponds to all the scopes, and does not take a condition. See "Diagnostic tracing scopes" on page 173.
OPTIMIZATION_LOGGING_WITH_PLANS	Collects data about join strategies considered by the optimizer. Information about the cost of execution for each strategy, and the complete XML plan describing the join strategy tree structure, is collected. Information about rewrites applied to the query is also collected. If a scope other than DATABASE, CONNECTION_NAME, CONNECTION_NUMBER, ORIGIN, or USER is used, the first recorded statement text might be different than the initial text of the query since some rewrites can be applied before it can be determined that optimization logging should be applied to the current statement. The OPTIMIZATION_LOGGING tracing type is automatically added whenever the OPTIMIZATION_LOGGING_WITH_PLANS tracing type is specified. This diagnostic tracing type corresponds to all the scopes, and does not take a condition. See "Diagnostic tracing scopes" on page 173.

See also

- "Diagnostic tracing scopes" on page 173
- "Diagnostic tracing conditions" on page 177

Diagnostic tracing conditions

The following table lists the diagnostic tracing **conditions** you can set. Conditions control the criteria that must be met in order for a tracing entry to be made for a specific diagnostic tracing type. Most conditions require a value, as noted below. Conditions are stored in the trace_condition column of the dbo.sa_diagnostic_tracing_level diagnostic table, and may have a corresponding value, such as an amount of time in milliseconds, stored in the value column. The values in the condition column reflect the settings specified in the **Database Tracing Wizard**.

Value in the trace_condition column	Description
NONE, or NULL	Records all the tracing data that satisfies the level and scope requirements. Using expensive diagnostic tracing levels (plans, for example) with this condition for extended time periods is not recommended.
SAMPLE_EVERY	Records tracing data that satisfies the level and scope requirements if more than the specified time interval has elapsed since the last event was recorded. Values: This condition takes a positive integer, reflecting time in milliseconds.
ABSOLUTE_COST	Records the statements with cost of execution greater than, or equal to, the specified value. Values: This condition takes a cost value, specified in milliseconds.
RELATIVE_COST_DIFFERENCE	Records the statements for which the difference between the expected time for execution and the real time for execution is greater than or equal to the specified value. Values: This condition takes a cost value specified as a percentage. For example, to log statements that are at least twice as slow as estimated, specify a value of 200.

See also

- "Diagnostic tracing scopes" on page 173
- "Diagnostic tracing types" on page 174

Determine current diagnostic tracing settings

Use the **Database Tracing Wizard** in Sybase Central to view current diagnostic tracing settings. When you are done examining the settings, cancel the wizard. You can also retrieve the diagnostic tracing settings in effect by querying the sa_diagnostic_tracing_level table.

You can retrieve diagnostic tracing settings regardless of whether a tracing session is in progress.

To determine the current diagnostic tracing settings (Sybase Central)

1. Use the SQL Anywhere 12 plug-in to connect to the database as a user with DBA or PROFILE authority.
2. Choose **Mode** » **Application Profiling**. If the **Application Profiling Wizard** appears, click **Cancel**.
3. In the left pane, right-click the database and choose **Tracing**.

   If the **Database Tracing Wizard** does not appear, choose **Tracing** » **Configure**.

4. Review the settings currently specified for diagnostic tracing on the **Edit Tracing Levels** list.
5. Click **Cancel**.

To determine the current diagnostic tracing settings (Interactive SQL)

1. Connect to the database as a user with DBA authority or as a user with PROFILE authority.
2. Query the sa_diagnostic_tracing_level table for rows in which the enabled column contains a 1.

   The database server returns the diagnostic tracing settings currently in use. A 1 in the enabled column indicates that the setting is in effect.

Example

The following statement shows you how to query the sa_diagnostic_tracing_level diagnostic table to retrieve the current diagnostic tracing settings:

```
SELECT * FROM sa_diagnostic_tracing_level WHERE enabled = 1;
```

The following table is an example result set from the query:

id	scope	identifier	trace_type	trace_condition	value	enabled
1	database	(NULL)	volatile_statistics	sample_every	1,000	1
2	database	(NULL)	nonvolatile_statistics	sample_every	60.000	1
3	database	(NULL)	connection_statistics	(NULL)	60,000	1
4	database	(NULL)	blocking	(NULL)	(NULL)	1
5	database	(NULL)	deadlock	(NULL)	(NULL)	1
6	database	(NULL)	plans_with_statistics	sample_every	2,000	1

See also

- "sa_diagnostic_tracing_level table" [*SQL Anywhere Server - SQL Reference*]
- "PROFILE authority" [*SQL Anywhere Server - Database Administration*]

Change the diagnostic tracing configuration settings

Diagnostic tracing settings are specific to a production database. You use the **Database Tracing Wizard** in Sybase Central to change diagnostic tracing settings when creating a tracing session. To learn how to start the **Database Tracing Wizard**, see "Create a diagnostic tracing session" on page 180.

Diagnostic tracing settings configured in the **Database Tracing Wizard** do not affect settings or behavior for the **Application Profiling Wizard**. The settings for the **Application Profiling Wizard** are preconfigured and cannot be changed.

You can also use the sa_set_tracing_level system procedure to change the diagnostic tracing level. This does not start a tracing session, and fails if a tracing session is already in progress. Also, it does not allow you as much control over other settings such as scopes, conditions, values, and so on. For more information about this procedure, see "sa_set_tracing_level system procedure" [*SQL Anywhere Server - SQL Reference*].

To change the diagnostic tracing level (Interactive SQL)

1. Use the SQL Anywhere 12 plug-in to connect to the database as a user with DBA or PROFILE authority.
2. Use the sa_set_tracing_level system procedure to set the diagnostic tracing levels.

Example

The following statement uses the sa_set_tracing_level system procedure to set the diagnostic tracing level to 1:

```
CALL sa_set_tracing_level( 1 );
```

Existing settings are overwritten with the default settings associated with diagnostic tracing level 1. To see the default settings associated with the various diagnostic tracing levels, see "Diagnostic tracing levels" on page 172.

Change diagnostic tracing settings when a tracing session is in progress

You can change diagnostic tracing settings while a tracing session is in progress using the **Database Tracing Wizard** in Sybase Central.

To change diagnostic tracing settings during a tracing session (Sybase Central)

1. Use the SQL Anywhere 12 plug-in to connect to the database as a user with DBA or PROFILE authority.
2. In the left pane, right-click the database and choose **Tracing** » **Change Tracing Levels**.
3. Add new, or delete existing, tracing levels.
4. Click **OK**.

Create a diagnostic tracing session

When you start a diagnostic tracing session, you also configure the type of tracing you want to perform, and specify where you want the tracing data to be stored. Your tracing session continues until you explicitly request that it stops.

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*].

Note
Starting a tracing session is also referred to as attaching tracing. Likewise, stopping a tracing session is referred to as detaching tracing. The SQL statements for starting and stopping tracing are, respectively, ATTACH TRACING and DETACH TRACING.

To create a diagnostic tracing session (Sybase Central)

1. Use the SQL Anywhere 12 plug-in to connect to the database as a user with DBA or PROFILE authority.
2. Right-click the database and choose **Tracing**.
3. Click **Next**.
4. On the **Tracing Detail Level** page, select the level of tracing.
5. On the **Edit Tracing Levels** page, customize the diagnostic tracing settings.
6. On the **Create External Database** page:
   - Select **Create A New Tracing Database**.
   - Select a location to save the database.
   - Complete the **User Name** and **Password** fields.
   - Select **Start Database On The Current Server**.
   - Click **Create Database**.
7. On the **Start Tracing** page:
   - Select **Save Tracing Data In An External Database**.
   - Complete the **User Name** and **Password** fields. Specify the user name and password used to connect to the production database.
   - In the **Other Connection Parameters** field, type the database server and database name in the form of a partial connect string. For example, `Server=Server47;DBN=TracingDB`

     Note
     Only DBN, DBF, Server, DBKEY, HOST, and LINKS (CommLinks) are supported in the connection string for an external database.

   - In the **Do You Want To Limit The Volume Of Trace Data That Is Stored** list, select an option.
8. Click **Finish**.
9. When you are done gathering diagnostic tracing data, right-click the database and choose **Tracing** » **Stop Tracing With Save**.

To create a diagnostic tracing session (Interactive SQL)

1. Connect to the database as the DBA, or as a user with PROFILE authority.

2. Use the sa_set_tracing_level system procedure to set the tracing levels.
3. Start tracing by executing an ATTACH TRACING statement.
4. Stop tracing by executing a DETACH TRACING statement.

   You can view the diagnostic tracing data in Application Profiling mode in Sybase Central. See "Application profiling" on page 158.

Examples

This example shows how to start diagnostic tracing on the current database, store the tracing data in a separate database, and set a two hour limit on the amount of data to store. This example is all on one line:

```
ATTACH TRACING TO
'UID=DBA;PWD=sql;Server=server47;DBN=tracing;Host=myhost' LIMIT HISTORY 2
HOURS;
```

This example shows how to start diagnostic tracing on the current database, store the tracing data in the local database, and set a two megabyte limit on the amount of data to store:

```
ATTACH TRACING TO LOCAL DATABASE LIMIT SIZE 2 MB;
```

This example shows how to stop diagnostic tracing and save the diagnostic data that was captured during the tracing session:

```
DETACH TRACING WITH SAVE;
```

This example shows how to stop diagnostic tracing and not save the diagnostic data.

```
DETACH TRACING WITHOUT SAVE;
```

See also

- "ATTACH TRACING statement" [*SQL Anywhere Server - SQL Reference*]
- "DETACH TRACING statement" [*SQL Anywhere Server - SQL Reference*]
- "sa_set_tracing_level system procedure" [*SQL Anywhere Server - SQL Reference*]
- "PROFILE authority" [*SQL Anywhere Server - Database Administration*]

Analyzing diagnostic tracing information

Diagnostic tracing data provides a record of all activities that took place on the database server and correspond to the diagnostic tracing levels and the tracing session settings. When reviewing the data, you must consider the settings that were in place. For example, the absence of a statement that you expected to see in a tracing session might indicate that the statement never ran, but it might also indicate that the statement was not expensive enough to fulfill a condition that only expensive statements be traced.

There are many reasons you may want to examine in detail what activities the database server is performing. These include troubleshooting performance problems, estimating resource usage to plan for future workloads, and debugging application logic.

See also

- "Application profiling tutorials" on page 234

Troubleshooting performance problems

Use the application profiling feature to determine whether performance problems are caused by:

- long application processing times
- poor query plans
- contention for shared hardware resources such as CPU or disk I/O
- contention for database objects
- suboptimal database design

When troubleshooting poor database performance, the first task is to determine whether the application or the database server is the primary cause. To determine how much processing time a client application is consuming, use the **Details** tab in the application profiling tool and filter the results by a single connection. If there are time differences between different requests from that connection, then the primary delay is within the application client.

If the database server is affecting performance, you will need to identify the specific cause.

See also

- "Application profiling tutorials" on page 234

Detecting when hardware resources are a limiting factor

As larger and larger workloads are placed on a database, performance is typically limited by CPU cycles, memory space, or disk I/O bandwidth. An inefficient application or database server could be the cause. If you cannot detect any inefficiencies, you may need to add additional hardware resources. To view a list of common inefficiencies and recommendations for solving them, see "Troubleshooting performance problems" on page 183.

Adding resources may not resolve scalability problems or improve computer performance. For example, if a database server is fully using all of its allotted CPUs, it may indicate that you should assign more CPU resources. However, doubling the number of CPUs available to the database server may not double the amount of work the database server can perform.

Use the **Statistics** tab in the **Application Profiling Details** area to detect whether hardware resources are a limiting factor for performance.

- **Detecting whether CPU is a limiting factor** To detect whether CPU as a limiting factor, check the ProcessCPU statistic. If this statistic is not present on the graph, click the **Add Statistics** button

and select **ProcessCPU**. If the graph shows ProcessCPU increasing at a rate of nearly 1 point per second per CPU assigned to the database server, then the CPU is a limiting factor. For example, for a database server running on two CPUs, if the Process CPU counter increased from 2220 to 2237 in ten seconds, this indicates that CPU usage over that twelve second period was (2237-2220) / 10s * 100 % = 170%, meaning that each CPU is running at 170% / 2 = 85% of its capacity.

- **Detecting whether memory is a limiting factor** To detect whether memory (buffer pool size) is a limiting factor, check the CacheHits and CacheReads database statistics. If these statistics are not present on the graph, click the **Add Statistics** button and select **CacheHits And CacheReads**. If CacheHits is less than 10% of CacheReads, this indicates that the buffer pool is too small. If the ratio is in the range of 10-70%, this may indicate that the buffer pool is too small—you should try increasing the cache size for the database server. If the ratio is above 70%, the cache size is likely adequate. Note that this strategy only applies while the database server is running at a steady-state—that is, it is servicing a typical workload and has not just been started.

- **Detecting whether I/O bandwidth is a limiting factor** To detect whether I/O bandwidth is a limiting factor, check the CurrIO database statistic. If this statistic is not present on the graph, click the **Add Statistics** button and select **CurrIO**. Look for the largest sustained number for this statistic. For example, look for a high plateau on the graph; the wider it is, the more significant the impact. If the graph has sustained values equal to, or greater than 3 + the number of physical disks used by database server, it may indicate that the disk system cannot keep up with the level of database server activity.

See also

- "Performance Monitor statistics" on page 197
- "Application profiling tutorials" on page 234
- "Troubleshooting performance problems" on page 183

Debugging application logic

If you have errors in your application code or in stored procedures, triggers, functions, or events, it can be useful to examine all statements executed by the database server that relate to the incorrect code. For applications that dynamically generate SQL, you can examine the actual text seen by the database server to detect errors in how the SQL text is built by the application. Such errors may cause queries to fail to be executed, or may return different results than the query was intended to return. For example, during development, your application may occasionally report that a SQL syntax error was encountered, but your application may not be instrumented to report the SQL text of the query that failed. If you have a trace taken when the application was run, you can search for statements that returned syntax (or other) errors, and see the exact text that was generated by your application.

For internal database objects such as procedures and triggers, you can use the debugger in Sybase Central. However, there may be times when it is more effective to cause the database server to trace all statements executed by a given procedure, and then examine these statements using the application profiling tool. For example, a given stored procedure may be returning an incorrect result once out of every 1000 invocations, but you may not understand under what conditions it fails. Rather than step through the procedure code 1000 times in the debugger, you could turn on diagnostic tracing for that procedure and run your application. Then, you could examine the set of statements that the database server executed, locate the set of statements that correspond to the incorrect execution of the procedure, and determine

either why the procedure failed, or the conditions under which it behaves unexpectedly. If you know under what conditions the procedure behaves unexpectedly, you can set a breakpoint in the procedure and investigate further with the debugger. See "Debugging procedures, functions, triggers, and events" on page 840.

Perform request trace analysis

When you have a specific application or request that is problematic, you can perform a request trace analysis to determine the problem. Request trace analysis involves configuring the **Database Tracing Wizard** to narrow diagnostic data gathering to only the user, connection, or request that is experiencing the problem. Then, using the various data viewing tools in Application Profiling mode, identifying any potential conflicts or bottlenecks.

To perform a request trace analysis (Sybase Central)

1. Use the SQL Anywhere 12 plug-in to connect to the database as a user with DBA or PROFILE authority.
2. Choose **Mode** » **Application Profiling**. If the **Application Profiling Wizard** appears, click **Cancel**.
3. Right-click the database and choose **Tracing** or choose **Tracing** » **Configure And Start Tracing**.
4. Follow the instructions in the **Database Tracing Wizard**.
5. When you are done gathering tracing data, right-click the database and choose **Tracing** » **Stop Tracing With Save**.
6. In the **Application Profiling Details** pane, click **Open An Analysis File Or Connect To A Tracing Database**.
7. Choose **In A Tracing Database**, and click **Open**.
8. Complete the **User Name** and **Password** fields and click **OK**.
9. In the **Application Profiling Details** pane, select the last entry in the **Logging Session ID** list.
10. Click **Database Tracing Data** tab at the bottom of the **Application Profiling Details** pane.

    You can select from several tabs that provide you with different views of the data gathered for your analysis. For example, the **Summary** tab allows you to see all requests executed against the database during the tracing session, including how many times each request was executed, execution duration times, the user who executed the request, and so on. If the list is long and you are looking for a specific request, click the **Filtering** title bar on the **Summary** tab and enter a string in the **SQL Statements Containing** field.

    To view more details about a specific request, right-click the request and choose **Show The Detailed SQL Statements For The Selected Summary Statement**. The **Details** tab opens. Right-click the row containing the request, and additional choices for information are provided, including viewing additional SQL statement, connection, and blocking details.

Creating an external tracing database

When you create a tracing session, you have the option of storing tracing data within the database being profiled. This is suitable for development environments where you are testing applications, or if there are few connections to the database. However, if your database typically handles 10 or more connections at any given time, it is recommended that you store tracing data in an external tracing database to reduce the impact on performance.

When you start a tracing session, use the **Database Tracing Wizard** to create an external tracing database. The **Database Tracing Wizard** unloads schema and permission information from the production database. You can use the tracing database to store data for subsequent tracing sessions. For information about creating a tracing session, see "Create a diagnostic tracing session" on page 180.

Use the Unload utility (dbunload) to manually create a tracing database without a tracing session.

To create an external tracing database using the Unload utility (dbunload)

1. Connect to the database as a user with DBA authority or as a user with PROFILE authority.
2. Execute a dbunload command, similar to the following, to unload the schema from the production database into the new tracing database:

   ```
   dbunload -c "UID=DBA;PWD=sql;Server=demo;DBN=demo" -an tracing.db -n -k -kd
   ```

   This example creates a new database with the name supplied by the -an option (*tracing.db*). The -n option unloads the schema from the database being profiled (in this case, the SQL Anywhere sample database, *demo.db*) into the new tracing database. The -k option populates the tracing database with information that the application profiling tool uses to analyze the tracing data. The -kd option places all the dbspaces in a single dbspace file.
3. If you want to store the tracing database on a separate computer, copy it to the new location.

See also

- "Unload utility (dbunload)" [*SQL Anywhere Server - Database Administration*]
- "PROFILE authority" [*SQL Anywhere Server - Database Administration*]

Other diagnostic tools and techniques

In addition to application profiling and diagnostic tracing, a variety of other diagnostic tools and techniques are available to help you analyze and monitor the current performance of your SQL Anywhere database.

Request logging

Request logging logs individual requests received from, and responses sent to, an application. It is most useful for determining what the database server is being asked to do by the application.

Request logging is also a good starting point for performance analysis of a specific application when it is not obvious whether the database server or the client is at fault. You can use request logging to determine the specific request to the database server that might be responsible for problems.

Note
All the functionality and data provided by the request logging feature is also available using diagnostic tracing. Diagnostic tracing also offers additional features and data. See "Advanced application profiling using diagnostic tracing" on page 169.

Logged information includes such things as timestamps, connection IDs, and request type. For queries, it also includes the isolation level, number of rows fetched, and cursor type. For INSERT, UPDATE, and DELETE statements, it also includes the number of rows affected and number of triggers fired.

Caution
The request log can contain sensitive information because it contains the full text of SQL statements that contain passwords, such as the GRANT CONNECT, CREATE DATABASE, and CREATE EXTERNAL LOGIN statements. If you are concerned about security, you should restrict access to the request log file.

You can use the -zr server option to turn on request logging when you start the database server. You can redirect the output to a request log file for further analysis using the -zo server option. The -zn and -zs option let you specify the number of request log files that are saved and the maximum size of request log files.

For more information about these options, see:

- "-zr dbeng12/dbsrv12 server option" [*SQL Anywhere Server - Database Administration*]
- "-zo dbeng12/dbsrv12 server option" [*SQL Anywhere Server - Database Administration*]
- "-zn dbeng12/dbsrv12 server option" [*SQL Anywhere Server - Database Administration*]
- "-zs dbeng12/dbsrv12 server option" [*SQL Anywhere Server - Database Administration*]

Note
These server options do not impact diagnostic tracing in Sybase Central. File-based request logging is completely separate from the diagnostic tracing feature in Sybase Central, which makes use of dbo-owned diagnostic tables in the database to store request log information.

The sa_get_request_times system procedure reads a request log and populates a global temporary table (satmp_request_time) with statements from the log and their execution times. For INSERT/UPDATE/DELETE statements, the time recorded is the time when the statements were executed. For queries, the time recorded is the total elapsed time from PREPARE to DROP (describe/open/fetch/close). That means you need to be aware of any open cursors.

Analyze satmp_request_time for statements that could be candidates for improvements. Statements that are cheap, but frequently executed, may represent performance problems.

You can use sa_get_request_profile to call sa_get_request_times and summarize satmp_request_time into another global temporary table called satmp_request_profile. This procedure also groups statements together and provides the number of calls, execution times, and so on. See "sa_get_request_times system

procedure" [*SQL Anywhere Server - SQL Reference*], and "sa_get_request_profile system procedure" [*SQL Anywhere Server - SQL Reference*].

Caution
When client statement caching is enabled, the max_client_statements_cached option should be set to 0 to disable client statement caching while the request log is captured if the log will be analyzed using the *tracetime.pl* Perl script. See "max_client_statements_cached option" [*SQL Anywhere Server - Database Administration*].

Filtering request logs

Output to the request log can be filtered to include only requests from a specific connection or from a specific database, using the sa_server_option system procedure. This can help reduce the size of the log when monitoring a database server with many active connections or multiple databases. See "sa_server_option system procedure" [*SQL Anywhere Server - SQL Reference*].

To filter according to a connection

- Use the following syntax:

```
CALL sa_server_option( 'RequestFilterConn' , connection-id );
```

  You can obtain *connection-id* by executing `CALL sa_conn_info( )`.

To filter according to a database

- Use the following syntax:

```
CALL sa_server_option( 'RequestFilterDB' , database-id );
```

  The *database-id* can be obtained by executing `SELECT CONNECTION_PROPERTY( 'DBNumber' )` when connected to that database. Filtering remains in effect until explicitly reset, or until the database server is shut down.

To reset filtering

- Use either of the following two statements to reset filtering either by connection or by database:

```
CALL sa_server_option( 'RequestFilterConn' , -1 );
CALL sa_server_option( 'RequestFilterDB' , -1 );
```

Outputting host variables to request logs

Host variable values can be output to a request log.

To include host variable values

- To include host variable values in the request log:
  - use the -zr server option with a value of **hostvars**
  - execute the following:


```
CALL sa_server_option( 'RequestLogging' , 'hostvars' );
```

The request log analysis procedure, sa_get_request_times, recognizes host variables in the log and adds them to the global temporary table satmp_request_hostvar.

Procedure profiling using system procedures

Procedure profiling provides valuable information about the usage of stored procedures, user-defined functions, events, system triggers, and triggers by all connections. You can perform procedure profiling in either Sybase Central, or Interactive SQL using system procedure calls. Sybase Central offers much greater features and flexibility to help you perform procedure profiling. For this reason, it is recommended that you perform procedure profiling using the procedure profiling features found in the Application Profiling mode of Sybase Central. See "Procedure profiling in Application Profiling mode" on page 160.

Enable profiling using sa_server_option

To enable procedure profiling in Interactive SQL

1. Connect to the database as a user with DBA authority or as a user with PROFILE authority.
2. Call the sa_server_option system procedure, setting the ProcedureProfiling option to ON.

   For example, enter:

   ```
   CALL sa_server_option( 'ProcedureProfiling' , 'ON' );
   ```

If necessary, you can see what procedures a specific user is using, without preventing other connections from using the database. This is useful if the connection already exists, or if multiple users connect with the same user ID.

To filter procedure profiling by user in Interactive SQL

1. Connect to the database as the DBA, or as a user with PROFILE authority.
2. Call the sa_server_option system procedure as follows:

   ```
   CALL sa_server_option( 'ProfileFilterUser' , 'userid' );
   ```

   The value of *userid* is the name of the user being monitored.

See also

- "sa_server_option system procedure" [*SQL Anywhere Server - SQL Reference*]

Reset profiling using sa_server_option

When you reset profiling, the database clears the old information and immediately starts collecting new information about procedures, functions, events, and triggers. This section explains how to reset procedure profiling from Interactive SQL using the sa_server_option system procedure.

The following sections assume that you are already connected to your database as the DBA, or as a user with PROFILE authority, and that procedure profiling is enabled.

To reset profiling in Interactive SQL

- Call the sa_server_option system procedure, setting the ProcedureProfiling option to RESET.

  For example, enter:

  ```
  CALL sa_server_option( 'ProcedureProfiling' , 'RESET' );
  ```

See also

- "sa_server_option system procedure" [*SQL Anywhere Server - SQL Reference*]

Disable profiling using sa_server_option

Once you are finished with the profiling information, you can either disable profiling or you can clear profiling. If you disable profiling, the database stops collecting profiling information and the information that it has collected to that point remains on the **Profile** tab in Sybase Central. If you clear profiling, the database turns profiling off and clears all the profiling data from the **Profile** tab in Sybase Central. This section explains how to disable procedure profiling from Interactive SQL using the sa_server_option system procedure.

To disable profiling (Interactive SQL)

- Call the sa_server_option system procedure, setting the ProcedureProfiling option to OFF.

  For example, enter:

  ```
  CALL sa_server_option( 'ProcedureProfiling' , 'OFF' );
  ```

To disable profiling and clear existing data (Interactive SQL)

- Call the sa_server_option system procedure, setting the ProcedureProfiling option to CLEAR.

  For example, enter:

  ```
  CALL sa_server_option( 'ProcedureProfiling' , 'CLEAR' );
  ```

See also

- "sa_server_option system procedure" [*SQL Anywhere Server - SQL Reference*]

Retrieve profiling information using system procedures

You can use system procedures to view procedure profiling information for the following objects: stored procedures, functions, events, system triggers, and triggers. Also, procedure profiling must already be enabled. See "Enable profiling using sa_server_option" on page 189.

The sa_procedure_profile system procedure shows in-depth profiling information, including execution times for the lines within each object; each line in the result set represents an executable line of code in the object.

The sa_procedure_profile_summary system procedure shows you the overall execution time for each object, giving you a summary of all objects that ran; each line in the result set represents the execution details for one object.

When reviewing the results from these system procedures, there may be more objects listed than those specifically called. This is because one object can call another object. For example, a trigger might call a stored procedure that, in turn, calls another stored procedure.

To view summary profiling information (Interactive SQL)

1. Connect to the database as a user with DBA authority or as a user with PROFILE authority.
2. Execute the sa_procedure_profile_summary system procedure.

   For example, enter:

   ```
   CALL sa_procedure_profile_summary;
   ```

3. Choose **SQL** » **Execute**.

   A result set with information about all the procedures in your database appears on the **Results** pane.

To view in-depth profiling information (Interactive SQL)

1. Connect to the database as a user with DBA authority or as a user with PROFILE authority.
2. Execute the sa_procedure_profile system procedure.

   For example, enter:

   ```
   CALL sa_procedure_profile;
   ```

3. Choose **SQL** » **Execute**.

   A result set with profiling information appears in the **Results** pane.

See also

- "sa_procedure_profile_summary system procedure" [*SQL Anywhere Server - SQL Reference*]
- "sa_procedure_profile system procedure" [*SQL Anywhere Server - SQL Reference*]
- "sa_server_option system procedure" [*SQL Anywhere Server - SQL Reference*]

Timing utilities

Some performance testing utilities, including fetchtst, instest, and trantest, are available in *samples-dir* *SQLAnywhere*. For information about the location of *samples-dir*, see "Samples directory" [*SQL Anywhere Server - Database Administration*].

The fetchtst utility measures fetch rates for an arbitrary query. The instest utility determines the time required for rows to be inserted into a table. The trantest utility measures the load that can be handled by a given server configuration given a database design and a set of transactions.

These tools give you more accurate timings than the graphical plan with statistics, and can provide an indication of the best achievable performance (for example, throughput) for a given server and database configuration.

Complete documentation for the tools can be found in the *readme.txt* file in the same folder as the utility.

Monitoring database performance

SQL Anywhere provides a set of statistics you can use to monitor database performance. There are three ways to access these statistics:

- **SQL functions** These functions allow your application to access SQL Anywhere database statistics directly. See "Monitoring statistics using SQL functions" on page 193.
- **Sybase Central Performance Monitor** This graphical tool queries the database and graphs only those statistics you have configured the Performance Monitor to graph. See "Monitoring statistics using Sybase Central Performance Monitor" on page 194.
- **Windows Performance Monitor** This is a monitoring tool provided by your Windows operating system. See "Monitor statistics using Windows Performance Monitor" on page 195.
- **Performance Statistics utility (dbstats)** This utility provides monitoring of database server, database, and connection statistics for database servers running on Unix. See "Performance Statistics utility (dbstats) (Unix)" [*SQL Anywhere Server - Database Administration*].
- **SQL Anywhere Console utility (dbconsole)** The utility provides administration and monitoring facilities for database server connections. See "SQL Anywhere Console utility (dbconsole)" [*SQL Anywhere Server - Database Administration*].

These methods are useful for monitoring in real time. However, you can also capture statistics as part of diagnostic tracing and save them for analysis at a later time. For more information about diagnostic tracing, see "Advanced application profiling using diagnostic tracing" on page 169.

For a complete listing of the SQL Anywhere statistics available for monitoring, see "Performance Monitor statistics" on page 197.

For more information about monitoring your database, see "Monitoring your database" [*SQL Anywhere Server - Database Administration*].

Monitoring statistics using SQL functions

SQL Anywhere provides a set of system functions that can access information on a per-connection, per-database, or server-wide basis. The kind of information available ranges from static information (such as the database server name) to detailed performance-related statistics (such as disk and memory usage).

Functions that retrieve system information

The following functions retrieve system information:

- **PROPERTY function** This function provides the value of a given property on a server-wide basis. See "PROPERTY function [System]" [*SQL Anywhere Server - SQL Reference*].
- **DB_PROPERTY and DB_EXTENDED_PROPERTY functions** These functions provide the value of a given property for a given database, or by default, for the current database. See "DB_PROPERTY function [System]" [*SQL Anywhere Server - SQL Reference*], and "DB_EXTENDED_PROPERTY function [System]" [*SQL Anywhere Server - SQL Reference*].
- **CONNECTION_PROPERTY and CONNECTION_EXTENDED_PROPERTY functions** These functions provide the value of a given property for a given connection, or by default, for the current connection. See "CONNECTION_PROPERTY function [System]" [*SQL Anywhere Server - SQL Reference*], and "CONNECTION_EXTENDED_PROPERTY function [String]" [*SQL Anywhere Server - SQL Reference*].

Supply as an argument only the name of the property you want to retrieve. The functions return the value for the current server, connection, or database.

For a complete list of the properties available from the system functions, see "System functions" [*SQL Anywhere Server - SQL Reference*].

Examples

The following statement sets a variable named server_name to the name of the current server:

```
SET server_name = PROPERTY( 'name' );
```

The following query returns the user ID for the current connection:

```
SELECT CONNECTION_PROPERTY( 'UserID' );
```

The following query returns the file name for the root file of the current database:

```
SELECT DB_PROPERTY( 'file' );
```

Improving query efficiency

For better performance, a client application monitoring database activity should use the PROPERTY_NUMBER function to identify a named property, and then use the number to repeatedly retrieve the statistic.

Property names obtained in this way are available for many different database statistics, from the number of transaction log page write operations and the number of checkpoints performed, to the number of reads of index leaf pages from the memory cache.

The following set of statements illustrates the process from Interactive SQL:

```
CREATE VARIABLE propnum INT;
CREATE VARIABLE propval INT;
SET propnum = PROPERTY_NUMBER( 'CacheRead' );
SET propval = DB_PROPERTY( propnum );
```

For more information about the PROPERTY_NUMBER function, see "PROPERTY_NUMBER function [System]" [*SQL Anywhere Server - SQL Reference*].

You can view many of these statistics in graph form from the Sybase Central Performance Monitor. See "Monitoring statistics using Sybase Central Performance Monitor" on page 194.

Monitoring statistics using Sybase Central Performance Monitor

The Sybase Central Performance Monitor is useful for tracking details about database server actions, including disk and memory access. The Sybase Central Performance Monitor can graph statistics for any SQL Anywhere database server to which you can connect.

Features of the Sybase Central Performance Monitor include:

- Real-time updates (at adjustable intervals)
- A color-coded and resizable legend
- Configurable appearance properties

The Sybase Central Performance Monitor queries the database to gather its statistics. This can affect some statistics such as Cache Reads/sec. If you do not want your statistics to be affected by monitoring, you can use the Windows Performance Monitor instead. See "Monitor statistics using Windows Performance Monitor" on page 195.

If you run multiple versions of SQL Anywhere simultaneously, you can also run multiple versions of the Performance Monitor simultaneously.

For a complete listing of the SQL Anywhere statistics available for monitoring, see "Performance Monitor statistics" on page 197.

Open the Sybase Central Performance Monitor

The Sybase Central Performance Monitor appears in the right pane of Sybase Central, when the **Performance Monitor** tab is selected. The graph displays only those statistics that you configured it to display. For more information about adding and removing statistics from the Performance Monitor graph, see "Add and remove statistics" on page 195.

To open the Performance Monitor

1. Use the SQL Anywhere 12 plug-in to connect to the database.

2. In the left pane, select the server.

3. In the right pane, click the **Performance Monitor** tab.

See also

- "Monitor statistics using Windows Performance Monitor" on page 195
- "Add and remove statistics" on page 195

Add and remove statistics

To add statistics to the Sybase Central Performance Monitor

1. Use the SQL Anywhere 12 plug-in to connect to the database.

2. In the left pane, select the server.

3. In the right pane, click the **Statistics** tab.

4. Right-click a statistic that is not currently being monitored and choose **Add To Performance Monitor**.

To remove statistics from the Sybase Central Performance Monitor

1. Use the SQL Anywhere 12 plug-in to connect to the database.

2. In the left pane, select the server.

3. In the right pane, click the **Statistics** tab.

4. Right-click a statistic that is currently being monitored and choose **Remove From Performance Monitor**.

> **Tip**
> You can also add a statistic to or remove one from the Sybase Central Performance Monitor on the statistic's properties window.

For a complete listing of the SQL Anywhere statistics available for monitoring, see "Performance Monitor statistics" on page 197.

See also

- "Open the Sybase Central Performance Monitor" on page 194
- "Monitor statistics using Windows Performance Monitor" on page 195

Monitor statistics using Windows Performance Monitor

As an alternative to using the Sybase Central Performance Monitor, you can use the Windows Performance Monitor.

The Windows Performance Monitor offers more performance statistics than the Sybase Central Performance Monitor, especially network communication statistics. It also uses a shared-memory scheme instead of performing queries against the database server, so it does not affect the statistics themselves.

If you run multiple versions of SQL Anywhere simultaneously, it is also possible to run multiple versions of the Performance Monitor simultaneously.

For a complete list of performance statistics you can monitor for SQL Anywhere, see "Performance Monitor statistics" on page 197.

When starting the database server that controls the memory used by the Windows Performance Monitor, you can specify the database server options, and the maximum number of connections or database that the Performance Monitor can monitor. See:

- "-ks dbeng12/dbsrv12 server option" [*SQL Anywhere Server - Database Administration*]
- "-ksc dbeng12/dbsrv12 server option" [*SQL Anywhere Server - Database Administration*]
- "-ksd dbeng12/dbsrv12 server option" [*SQL Anywhere Server - Database Administration*]

The following instructions explain how to start Windows Performance Monitor on Windows XP. For other versions of Windows, consult your Windows operating system documentation for information about how to start Windows Performance Monitor.

To use the Windows Performance Monitor on Windows XP

1. With a SQL Anywhere server running, start the Performance Monitor:
   - From the Windows **Control Panel**, choose **Administrative Tools**.
   - Choose **Performance**.
2. On the toolbar, click the Plus sign tool (+).
3. In the **Performance Object** list, select one of the following:
   - **SQL Anywhere 12 Connection** This monitors performance for a single connection. A connection must currently exist to see this selection.
   - **SQL Anywhere 12 Database** This monitors performance for a single database.
   - **SQL Anywhere 12 Server** This monitors performance on a server-wide basis.

   The **Counters** box displays a list of the statistics you can view.

   If you selected SQL Anywhere Connection or SQL Anywhere Database, the **Instances** box displays a list of the connections or databases upon which you can view statistics.
4. In the **Counter** list, click a statistic to view. To select multiple statistics, hold the Ctrl or Shift keys while clicking.

5. If you selected SQL Anywhere 12 Connection or SQL Anywhere 12 Database, choose a database connection or database to monitor from the **Instances** box.

6. For a description of the selected counter, click **Explain**.

7. To display the counter, click **Add**.

8. When you have selected all the counters you want to display, click **Close**.

Performance Monitor statistics

SQL Anywhere provides the following statistics:

Rates are reported in 1 second intervals.

Cache statistics

These statistics describe the use of the cache.

Statistic	Scope	Description
Cache Hits/sec	Connection and database	Shows the rate at which database page lookups are satisfied by finding the page in the cache.
Cache Reads: Index Interior/sec	Connection and database	Shows the rate at which index internal-node pages are read from the cache.
Cache Reads: Index Leaf/sec	Connection and database	Shows the rate at which index leaf pages are read from the cache.
Cache Reads: Table/sec	Connection and database	Shows the rate at which table pages are read from the cache.

Statistic	Scope	Description
Cache Reads: Total Pages/sec	Connection and database	Shows the rate at which database pages are looked up in the cache.
Cache Reads: Work Table	Connection and database	Shows the rate at which work table pages are being read from the cache.
Cache Replacements: Total Pages/sec	Server	Shows the rate at which database pages are being purged from the cache to make room for another page that is needed.
Cache Size: Current	Server	Shows the current size of the database server cache, in kilobytes.
Cache Size: Maximum	Server	Shows the maximum allowed size of the database server cache, in kilobytes.
Cache Size: Minimum	Server	Shows the minimum allowed size of the database server cache, in kilobytes.
Cache Size: Peak	Server	Shows the peak size of the database server cache, in kilobytes.

Checkpoint and recovery statistics

These statistics isolate the checkpoint and recovery actions performed when the database is in an idle state.

Statistic	Scope	Description
Checkpoint Flushes/sec	Database	Shows the rate at which ranges of adjacent pages are written out during a checkpoint.
Checkpoint Urgency	Database	Shows the checkpoint urgency, expressed as a percentage.
Checkpoints/sec	Database	Shows the rate at which checkpoints are performed.
ChkptLog: Bitmap size	Database	Shows the size of the checkpoint log bitmap.
ChkptLog: Commit to disk/sec	Database	Shows the rate at which checkpoint log commit_to_disk operations are being performed.

Statistic	Scope	Description
ChkptLog: Log size	Database	Shows the size of the checkpoint log in pages.
ChkptLog: Page images saved/sec	Database	Shows the rate at which pages are being saved in the checkpoint log before modification.
ChkptLog: Pages in use	Database	Shows the number of pages in the checkpoint log which are currently in use.
ChkptLog: Relocate pages/sec	Database	Shows the rate at which pages in the checkpoint log are being relocated.
ChkptLog: Save preimage/sec	Database	Shows the rate at which new database page preimages are being added to the checkpoint log.
ChkptLog: Write pages/sec	Database	Shows the rate at which pages are being written to the checkpoint log.
ChkptLog: Writes/sec	Database	Shows the rate at which disk writes are being performed in the checkpoint log. One write can include multiple pages.
ChkptLog: Writes to bitmap/sec	Database	Shows the rate at which disk writes are being performed in the checkpoint log for bitmap pages.
Idle Actives/sec	Database	Shows the rate at which the database server's idle thread becomes active to do idle writes, idle checkpoints, and so on.
Idle Checkpoint Time	Database	Shows the total time spent doing idle checkpoints, in seconds.
Idle Checkpoints/sec	Database	Shows the rate at which checkpoints are completed by the database server's idle thread. An idle checkpoint occurs whenever the idle thread writes out the last dirty page in the cache.
Idle Writes/sec	Database	Shows the rate at which disk writes are issued by the database server's idle thread.
Recovery I/O Estimate	Database	Shows the estimated number of I/O operations required to recover the database.
Recovery Urgency	Database	Shows the recovery urgency expressed as a percentage.

Communications statistics

These statistics describe client/server communication activity.

Statistic	Scope	Description
Comm: Bytes Received/sec	Connection and server	Shows the rate at which network data (in bytes) are received.
Comm: Bytes Received Uncompressed/sec	Connection and server	Shows the rate at which bytes would have been received if compression was disabled.
Comm: Bytes Sent/sec	Connection and server	Shows the rate at which bytes are transmitted over the network.
Comm: Bytes Sent Uncompressed/sec	Connection and server	Shows the rate at which bytes would have been sent if compression was disabled.
Comm: Free Buffers	Server	Shows the number of free network buffers.
Comm: Multi-packets Received/sec	Server	Shows the rate at which multi-packet deliveries are received.
Comm: Multi-packets Sent/sec	Server	Shows the rate at which multi-packet deliveries are transmitted.
Comm: Packets Received/sec	Connection and server	Shows the rate at which network packets are received.
Comm: Packets Received Uncompressed/sec	Connection and server	Shows the rate at which network packets would have been received if compression was disabled.
Comm: Packets Sent/sec	Connection and server	Shows the rate at which network packets are transmitted.
Comm: Packets Sent Uncompressed/sec	Connection and server	Shows the rate at which network packets would have been transmitted if compression was disabled.
Comm: Remoteput Waits/sec	Server	Shows the rate at which the communication link must wait because it does not have buffers available to send information. This statistic is collected for TCP/IP only.

Statistic	Scope	Description
Comm: Requests Received	Connection and server	Shows the number of client/server communication requests or round-trips. It is different from the Comm: Packets Received statistic in that multi-packet requests count as one request, and liveness packets are not included.
Comm: Send Fails/sec	Server	Shows the rate at which the underlying protocol(s) failed to send a packet.
Comm: Total Buffers	Server	Shows the total number of network buffers.
Comm: Unique Client Addresses	Server	Shows the number of unique client network addresses connected to the database server. This is usually the number of client machines connected, and may be less than the total number of connections.

Disk I/O statistics

These statistics combine disk reads and disk writes to give overall information about the amount of activity devoted to disk I/O.

Statistic	Scope	Description
Disk: Active I/Os	Database	Shows the current number of file I/Os issued by the database server which have not yet completed.
Disk: Maximum Active I/Os	Database	Shows the maximum value "Disk: Active I/Os" has reached.

Disk read statistics

These statistics describe the amount and type of activity devoted to reading information from disk.

Statistic	Scope	Description
Disk Reads: Total Pages/sec	Connection and database	Shows the rate at which pages are read from a file.

Statistic	Scope	Description
Disk Reads: Active	Database	Shows the current number of file reads issued by the database server which haven't yet completed.
Disk Reads: Index interior/sec	Connection and database	Shows the rate at which index internal-node pages are being read from disk.
Disk Reads: Index leaf/sec	Connection and database	Shows the rate at which index leaf pages are being read from disk.
Disk Reads: Table/sec	Connection and database	Shows the rate at which table pages are being read from disk.
Disk Reads: Maximum Active	Database	Shows the maximum value "Disk Reads: Active" has reached.
Disk Reads: Work Table	Connection and database	Shows the rate at which work table pages are being read from disk.

Disk write statistics

These statistics describe the amount and type of activity devoted to writing information to disk.

Statistic	Scope	Description
Disk Writes: Active	Database	Shows the current number of file writes issued by the database server that aren't yet completed.
Disk Writes: Maximum Active	Database	Shows the maximum value "Disk Writes: Active" has reached.
Disk Writes: Commit Files/sec	Database	Shows the rate at which the database server forces a flush of the disk cache. Windows platforms use unbuffered (direct) I/O, so the disk cache doesn't need to be flushed.
Disk Writes: Database Extends/sec	Database	Shows the rate at which the database file is extended, in pages/sec.
Disk Writes: Temp Extends/sec	Database	Shows the rate at which temporary files are extended, in pages/sec.

Statistic	Scope	Description
Disk Writes: Pages/sec	Connection and database	Shows the rate at which modified pages are being written to disk.
Disk Writes: Transaction Log/sec	Connection and database	Shows the rate at which pages are written to the transaction log.
Translog Group Commits/sec	Connection and database	Shows the rate at which a commit of the transaction log was requested but the log had already been written (so the commit was done for free).

Index statistics

These statistics describe the use of the index.

Statistic	Scope	Description
Index: Adds/sec	Connection and database	Shows the rate at which entries are added to indexes.
Index: Lookups/sec	Connection and database	Shows the rate at which entries are looked up in indexes.
Index: Full Compares/sec	Connection and database	Shows the rate at which comparisons beyond the hash value in an index must be performed.

Memory diagnostic statistics

These statistics describe how the database server is using memory.

Statistic	Scope	Description
Cache: Multi-Page Allocations	Server	Shows the number of multi-page allocations.
Cache: Panics	Server	Shows the number of times the cache manager has failed to find a page to allocate.
Cache: Scavenge Visited	Server	Shows the number of pages visited while scavenging for a page to allocate.

Statistic	Scope	Description
Cache: Scavenges	Server	Shows the number of times the cache manager has scavenged for a page to allocate.
Cache Pages: Allocated Structures	Server	Shows the number of cache pages that have been allocated for database server data structures.
Cache Pages: File	Server	Shows the number of cache pages used to hold data from database files.
Cache Pages: File Dirty	Server	Shows the number of cache pages that are dirty (needing a write).
Cache Pages: Free	Server	Shows the number of cache pages not being used.
Cache Pages: Pinned	Server	Shows the number of pages currently unavailable for reuse.
Cache Replacements: Total Pages/sec	Server	Shows the rate at which database pages are being purged from the cache to make room for another page that is needed.
Heaps: Carver	Connection and server	Shows the number of heaps used for short-term purposes such as query optimization..
Heaps: Query Processing	Connection and server	Shows the number of heaps used for query processing (hash and sort operations).
Heaps: Relocatable	Connection and server	Shows the number of relocatable heaps.
Heaps: Relocatable Locked	Connection and server	Shows the number of relocatable heaps currently locked in the cache.
Map physical memory/sec	Server	Shows the rate at which database page address space windows are being mapped to physical memory in the cache using Address Windowing Extensions.

Statistic	Scope	Description
Mem Pages: Carver	Connection and server	Shows the number of heap pages used for short-term purposes such as query optimization.
Mem Pages: Pinned Cursor	Server	Shows the number of pages used to keep cursor heaps pinned in memory.
Mem Pages: Query Processing	Connection and server	Shows the number of cache pages used for query processing (hash and sort operations).
Query Memory: Current Active	Connection and server	Shows the current number of requests actively using query memory.
Query Memory: Estimated Active	Server	Shows the database server's estimate of the steady state average of the number of requests actively using query memory.
Query Memory: Extra Available	Server	Shows the amount of memory available to grant beyond the base memory-intensive grant.
Query Memory: Number of Grant Fails	Connection and server	Shows the total number of times any request waited for query memory and failed to get it.
Query Memory: Number of Grant Requests	Connection and server	Shows the total number of times any request attempted to acquire query memory.
Query Memory: Number of Grant Waits	Connection and server	Shows the total number of times any request waited for memory.
Query Memory: Pages Granted	Connection and server	Shows the number of pages currently granted to requests.
Query Memory: Requests Waiting	Connection and server	Shows the current number of requests waiting for query memory.

Memory pages statistics

These statistics describe the amount and purpose of memory used by the database server.

Statistic	Scope	Description
Mem Pages: Lock Table	Database	Shows the number of pages used to store lock information.
Mem Pages: Locked Heap	Server	Shows the number of heap pages locked in the cache.
Mem Pages: Main Heap	Server	Shows the number of pages used for global database server data structures.
Mem Pages: Map Pages	Database	Shows the number of map pages used for accessing the lock table, frequency table, and table layout.
Mem Pages: Procedure Definitions	Database	Shows the number of relocatable heap pages used for procedures.
Mem Pages: Relocatable	Database	Shows the number of pages used for relocatable heaps (cursors, statements, procedures, triggers, views, and so on).
Mem Pages: Relocations/sec	Database	Shows the rate at which relocatable heap pages are read from the temporary file.
Mem Pages: Rollback Log	Connection and database	Shows the number of pages in the rollback log.
Mem Pages: Trigger Definitions	Database	Shows the number of relocatable heap pages used for triggers.
Mem Pages: View Definitions	Database	Shows the number of relocatable heap pages used for views.

Request statistics

These statistics describe the database server activity devoted to responding to requests from client applications.

Statistic	Scope	Description
Cursors	Connection	Shows the number of declared cursors currently maintained by the database server.

Statistic	Scope	Description
Cursors Open	Connection	Shows the number of open cursors currently maintained by the database server.
Lock Count	Connection and database	Shows the number of locks.
Requests/sec	Server	Shows the rate at which the database server is entered to allow it to handle a new request or continue processing an existing request.
Requests: Active	Server	Shows the number of database server threads that are currently handling a request.
Tasks: Exchange	Server	Shows the number of database server threads that are currently being used for parallel execution of a query.
Requests: Unscheduled	Server	Shows the number of requests that are currently queued up waiting for an available database server thread.
Snapshot Count	Connection and database	Shows the number of active snapshots.
Statement Cache Hits	Connection and server	Shows the rate at which statement prepares cached by the client are being re-used by the database server.
Statement Cache Misses	Connection and server	Shows the rate at which statement prepares cached by the client need to be prepared again by the database server.
Statement Prepares	Connection and database	Shows the rate at which statement prepares are being handled by the database server.
Statements	Connection	Shows the number of prepared statements currently maintained by the database server.
Transaction Commits	Connection	Shows the rate at which Commit requests are handled.
Transaction Rollbacks	Connection	Shows the rate at which Rollback requests are handled.

Miscellaneous statistics

Statistic	Scope	Description
Avail IO	Server	Shows the current number of available I/O control blocks.
Connection Count	Database	Shows the number of connections to this database.
Main Heap Bytes	Server	Shows the number of bytes used for global database server data structures.
Query: Plan cache pages	Connection and database	Shows the number of pages used to cache execution plans.
Query: Low memory strategies	Connection and database	Shows the number of times the database server changed its execution plan during execution because of low memory conditions.
Query: Rows materialized/sec	Connection and database	Shows the rate at which rows are written to work tables during query processing.
Requests: GET DATA/sec	Connection and database	Shows the rate at which a connection is issuing GET DATA requests.
Temporary Table Pages	Connection and database	Shows the number of pages in the temporary file used for temporary tables.
Version Store Pages	Database	Shows the number of pages of the temporary file currently being used for the row version store when snapshot isolation is enabled.

Performance improvement tips

Always use a transaction log

Using a transaction log can provide data protection, and can dramatically improve the performance of SQL Anywhere.

When operating without a transaction log, SQL Anywhere performs a checkpoint at the end of every transaction which consumes considerable resources.

When operating with a transaction log, SQL Anywhere only writes notes detailing the changes as they occur. It can choose to write the new database pages all at once, at the most efficient time. **Checkpoints** make sure information enters the database file, and that it is consistent and up to date.

You can further improve performance if you store the transaction log on a different physical device than the one containing the primary database file. The extra drive head does not generally have to seek to get to the end of the transaction log.

Check for concurrency issues

When the database server processes a transaction, it can lock one or more table rows. The locks maintain the reliability of information stored in the database by preventing concurrent access by other transactions. They also improve the accuracy of result queries by identifying information that is in the process of being updated.

The database server places these locks automatically and needs no explicit instruction. It holds all the locks acquired by a transaction until the transaction is completed. The transaction that has access to the row is said to hold the lock. Depending on the type of lock, other transactions may have limited access to the locked row, or none at all.

Performance can be compromised if a row or rows are frequently accessed by several users simultaneously. If you suspect locking problems, consider using the sa_locks procedure to obtain information on locks in the database. See "sa_locks system procedure" [*SQL Anywhere Server - SQL Reference*].

If lock issues are identified, information on the connection processes involved can be found using the AppInfo connection property. See "Connection properties" [*SQL Anywhere Server - Database Administration*].

Choose the optimizer goal

The optimization_goal option controls whether SQL Anywhere optimizes SQL statements for response time (First-row) or for total resource consumption (All-rows). In simpler terms, you can choose whether to optimize query processing towards returning the first row quickly, or towards minimizing the cost of returning the complete result set.

If the option is set to First-row, SQL Anywhere chooses an access plan that is intended to reduce the time to fetch the first row of the query's result, possibly at the expense of total retrieval time. In particular, the optimizer typically avoids, if possible, access plans that require the materialization of results to reduce the time to return the first row. With this setting, for example, the optimizer favors access plans that utilize an index to satisfy a query's ORDER BY clause, rather than plans that require an explicit sorting operation.

The optimization goal used by the optimizer for a particular statement is decided using these rules:

- If the main query block has a table in the FROM clause with the table hint set to FASTFIRSTROW, then the statement is optimized using the First-row optimization goal.

- If the statement has an OPTION clause containing a setting for the optimization_goal option, then the statement is optimized using this setting.
- Else, the optimizer uses the current setting of the option optimization_goal option.

Note that even if the optimization goal is First-row, the optimizer may be unable to find a plan that can quickly return the first row. For example, statements requiring materialization due to the presence of DISTINCT, GROUP BY, or ORDER BY clauses, and for which a relevant index does not exist to provide the necessary order, are optimized with the All-rows goal.

If the option is set to All-rows (the default), the SQL Anywhere query is optimized to choose an access plan with the minimal estimated total retrieval time. Setting optimization_goal to All-rows may be appropriate for applications that intend to process the entire result set, such as PowerBuilder DataWindow applications.

See also

- "optimization_goal option" [*SQL Anywhere Server - Database Administration*]
- "FROM clause" [*SQL Anywhere Server - SQL Reference*]

You can also refer to the OPTION clause of SQL statements such as the following:

- "DELETE statement" [*SQL Anywhere Server - SQL Reference*]
- "INSERT statement" [*SQL Anywhere Server - SQL Reference*]
- "SELECT statement" [*SQL Anywhere Server - SQL Reference*]
- "MERGE statement" [*SQL Anywhere Server - SQL Reference*]
- "UPDATE statement" [*SQL Anywhere Server - SQL Reference*]
- "UNION statement" [*SQL Anywhere Server - SQL Reference*]

Collect statistics on small tables

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

By default, SQL Anywhere creates statistics for all tables with five or more rows. If you need to create statistics for a table with less than five rows, you can do so using the CREATE STATISTICS statement. This statement creates statistics for all tables, regardless of how many rows are in a table. Once created, the statistics are automatically maintained by SQL Anywhere. See "CREATE STATISTICS statement" [*SQL Anywhere Server - SQL Reference*].

Declare constraints

Undeclared primary key-foreign key relationships exist between tables when there is an implied relationship between the values of columns in different tables. It is true that not declaring the relationship can save time on index maintenance, however, declaring the relationship can improve performance of

queries when joins take place because the cost model is able to do a better job of estimation. See "Using table and column constraints" on page 74.

Minimize cascading referential actions

Cascading referential actions are costly because they cause updates to multiple tables for every transaction and this affects performance. For example, if the foreign key from Employees to Departments was defined with ON UPDATE CASCADE, then updating a department ID would automatically update the Employees table. While cascading referential actions are convenient, sometimes it might be more efficient to implement them in application logic instead. See "Ensuring data integrity" on page 65.

Monitor query performance

SQL Anywhere includes several tools for testing the performance of queries. These tools are stored in subdirectories under *samples-dir\SQLAnywhere*, as noted below. Complete documentation about each tool can be found in a *readme.txt* file that is located in the same folder as the tool. For more information about the location of *samples-dir*, see "Samples directory" [*SQL Anywhere Server - Database Administration*].

For information about system procedures that measure query execution times, see "sa_get_request_profile system procedure" [*SQL Anywhere Server - SQL Reference*] and "sa_get_request_times system procedure" [*SQL Anywhere Server - SQL Reference*].

fetchtst

Function Determines the time required for a result set to be retrieved.

Location *samples-dir\SQLAnywhere\PerformanceFetch*

odbcfet

Function Determines the time required for a result set to be retrieved. This tool is similar to fetchtst, but with less functionality.

Location *samples-dir\SQLAnywhere\PerformanceFetch*

instest

Function Determines the time required for rows to be inserted into a table.

Location *samples-dir\SQLAnywhere\PerformanceInsert*

trantest

Function Measures the load that can be handled by a given database server configuration given a database design and a set of transactions.

Location *samples-dir\SQLAnywhere\PerformanceTransaction*

Normalize your table structure

One or more database tables may contain multiple copies of the same information (for example, a column that is repeated in several tables), and your table may need to be normalized.

Normalization reduces duplication in a relational database. For example, suppose your company employees work at several different offices. To normalize the database, consider placing information about the offices (such as its address and main telephone numbers) in a separate table, rather than duplicating all this information for every employee.

If the amount of duplicate information is small, you may find it better to duplicate the information and maintain its integrity using triggers or other constraints.

For more information about normalizing data, see "Creating a SQL Anywhere database" [*SQL Anywhere Server - Database Administration*].

Review the order of columns in tables

Columns in a row are accessed in a sequential manner in the order of their creation. For example, to access columns at the end of a row, SQL Anywhere skips columns that appear earlier in the row. Primary key columns are always stored at the beginning of rows. For this reason, it is important to create tables so that small and/or frequently accessed columns are placed before seldom accessed columns in the table.

See also

- "Creating a SQL Anywhere database" [*SQL Anywhere Server - Database Administration*]
- "Managing primary keys" on page 9

Place different files on different devices

Disk drives operate much more slowly than modern processors or RAM. Often, simply waiting for the disk to read or write pages is the reason that a database server is slow.

You may improve database performance by putting different physical database files on different physical devices. For example, while one disk drive is busy swapping database pages to and from the cache, another device can be writing to the log file.

Notice that to gain these benefits, the devices must be independent. A single disk, partitioned into smaller logical drives, is unlikely to yield benefits.

SQL Anywhere uses four types of files:

1. database (*.db*)

2. transaction log (*.log*)

3. transaction log mirror (*.mlg*)
4. temporary file (*.tmp*)

The **database file** holds the entire contents of your database. A single file can contain a single database, or you can add up to 12 dbspaces, which are additional files holding portions of the same database. You choose a location for the database file and dbspaces.

The **transaction log file** is required for recovery of the information in your database in the event of a failure. For extra protection, you can maintain a duplicate copy of the transaction log in a third type of file called a **transaction log mirror file**. SQL Anywhere writes the same information at the same time to each of these files.

Tip
Placing the transaction log mirror file (if you use one) on a physically separate drive helps protect against disk failure, and SQL Anywhere runs faster because it can efficiently write to the log and log mirror files. To specify the location of the transaction log and transaction log mirror files, use the Transaction Log utility (dblog), or the **Change Log File Settings Wizard** in Sybase Central. See "Transaction Log utility (dblog)" [*SQL Anywhere Server - Database Administration*], and "Change the location of a transaction log" [*SQL Anywhere Server - Database Administration*].

The **temporary file** is used when SQL Anywhere needs more space than is available to it in the cache for such operations as sorting and forming unions. When the database server needs this space, it generally uses it intensively. The overall performance of your database becomes heavily dependent on the speed of the device containing the temporary file.

Tip
If the temporary file is on a fast device, physically separate from the one holding the database file, SQL Anywhere typically runs faster. This is because many of the operations that necessitate using the temporary file also require retrieving a lot of information from the database. Placing the information on two separate disks allows the operations to take place simultaneously.

Choose the location of your temporary file carefully. The location of the temporary file can be specified when starting the database server using the -dt server option. If you do not specify a location for the temporary file when starting the database server, SQL Anywhere checks the following environment variables, in order:

1. SATMP
2. TMP
3. TMPDIR
4. TEMP

If an environment variable is not defined, SQL Anywhere places its temporary file in the current directory for Windows, and in the */tmp* directory for Unix.

If your computer has enough fast devices, you can gain even more performance by placing each of these files on a separate device. You can even divide your database into multiple dbspaces, located on separate

devices. In such a case, group tables in the separate dbspaces so that common join operations read information from different dbspaces.

When you create all tables or indexes in a location other than the system dbspace, the system dbspace is only used for the checkpoint log and system tables. This is useful if you want to put the checkpoint log on a separate disk from the rest of your database objects for performance reasons. To create base tables in another dbspace change all the CREATE TABLE statements to use the IN DBSPACE clause to specify the alternative dbspace, or change the setting of the default_dbspace option before creating any tables. Temporary tables can only be created in the TEMPORARY dbspace. See "default_dbspace option" [*SQL Anywhere Server - Database Administration*], and "CREATE TABLE statement" [*SQL Anywhere Server - SQL Reference*].

For more information about the default dbspace for base and temporary tables, see "Using additional dbspaces" [*SQL Anywhere Server - Database Administration*].

A similar strategy involves placing the temporary and database files on a RAID device or a stripe set. Although such devices act as a logical drive, they dramatically improve performance by distributing files over many physical drives and accessing the information using multiple heads.

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 dbeng12/dbsrv12 server option" [*SQL Anywhere Server - Database Administration*].

See also

- "Use work tables in query processing (use All-rows optimization goal)" on page 232
- "Backup and data recovery" [*SQL Anywhere Server - Database Administration*]
- "SATMP environment variable" [*SQL Anywhere Server - Database Administration*]

Rebuild your database

Rebuilding your database is the process of unloading and reloading your entire database. It is also called upgrading your database file format.

Rebuilding removes all the information, including data and schema, and puts it all back in a uniform fashion. Like defragmenting your disk drive, performance is improved space is filled in. It also gives you the opportunity to change certain settings. See "Rebuilding databases" on page 731.

Reduce fragmentation

Fragmentation occurs naturally as you make changes to your database. Performance can suffer if your files, tables, or indexes are excessively fragmented. Reducing fragmentation becomes more important as your database increases in size. SQL Anywhere contains stored procedures that generate information about the fragmentation of files, tables, and indexes.

If you are noticing a significant decrease in performance, consider:

- rebuilding your database to reduce table and/or index fragmentation, especially if you have performed extensive delete/update/insert activity on multiple tables
- putting the database on a disk partition by itself to reduce file fragmentation
- running one of the available Windows utilities periodically to reduce file fragmentation
- reorganizing your tables to reduce database fragmentation
- using the REORGANIZE TABLE statement to defragment rows in a table, or to compress indexes which may have become sparse due to DELETEs. Reorganizing tables can reduce the total number of pages used to store a table and its indexes, and it may reduce the number of levels in an index tree as well.

Reduce file fragmentation

A fragmented database file can affect the performance of your database server. Reducing disk fragmentation becomes more important as the size of your database increases.

The database server determines the number of file fragments in each dbspace when you start a database on Windows. The database server displays the following information in the database server messages window when the number of fragments is greater than one: `Database file "mydatabase.db" consists of nnn fragments`.

You can also obtain the number of database file fragments using the DBFileFragments database property. See "Database properties" [*SQL Anywhere Server - Database Administration*].

To eliminate file fragmentation problems, put the database on a disk partition by itself and then periodically run one of the available Windows disk defragmentation utilities.

See also

- "Performance warning: Database file %1 consists of %2 disk fragments" [*SQL Anywhere Server - Database Administration*]
- "DBFileFragments database property" [*SQL Anywhere Server - Database Administration*]

Reduce table fragmentation

Table fragmentation occurs when rows are not stored contiguously, or when rows are split between multiple pages. These rows require additional page access and this reduces the performance of the database server.

The effect that fragmentation has on performance varies. A table might be highly fragmented, but if it fits in memory, and the way it is accessed allows the pages to be cached, then the impact may be minimal. However, a fragmented table may cause much more I/O to be done and can significantly reduce performance if split rows are accessed frequently and the cost of extra I/Os is not reduced by caching.

While reorganizing tables and rebuilding a database can reduce fragmentation, doing so too frequently or not frequently enough, can also impact performance. Experiment using the tools and methods described in the following section to determine an acceptable level of fragmentation for your tables.

If you reduce fragmentation and performance is still poor, another issue may be to blame, such as inaccurate statistics. See "Improving database performance" on page 157.

Determine the degree of table fragmentation

Checking the table fragmentation just once is not helpful in determining whether to defragment to improve performance. Instead, rebuild your database and check the table fragmentation to establish baseline results. Then, continue to check the table fragmentation periodically over an extended length of time, looking for correlation between the change in fragmentation to changes in performance measures. This method helps you determine the rate at which tables become fragmented to the degree that performance is impacted, and so determine the optimal frequency at which to defragment tables.

To obtain information about the degree of fragmentation of your database tables, use one of the following methods:

- The sa_table_fragmentation system procedure. See "sa_table_fragmentation system procedure" [*SQL Anywhere Server - SQL Reference*].
- The **Fragmentation** tab in the SQL Anywhere plug-in. The **Fragmentation** tab provides a graphical representation of the results from running sa_table_fragmentation system procedure on base tables. See "Using the Fragmentation tab (SQL Anywhere plug-in)" on page 217.

To determine the degree of fragmentation (SQL)

1. Connect to the database as a user with DBA authority.
2. Run the sa_table_fragmentation system procedure.

   For example, run the following statement to get information about all tables in the database:

   ```
   CALL sa_table_fragmentation( );
   ```

   See "sa_table_fragmentation system procedure" [*SQL Anywhere Server - SQL Reference*].

See also

- "sa_table_fragmentation system procedure" [*SQL Anywhere Server - SQL Reference*]
- "REORGANIZE TABLE statement" [*SQL Anywhere Server - SQL Reference*]
- "Unload utility (dbunload)" [*SQL Anywhere Server - Database Administration*]

Methods to reduce table fragmentation

The following methods help control table fragmentation:

- **Use PCTFREE** SQL Anywhere reserves extra room on each page to allow rows to grow slightly. When an update to a row causes it to grow beyond the original space allocated for it, the row is split and the initial row location contains a pointer to another page where the entire row is stored. For

example, filling empty rows with UPDATE statements or inserting new columns into a table can lead to severe row splitting. As more rows are stored on separate pages, more time is required to access the additional pages.

You can reduce the amount of fragmentation in your tables by specifying the percentage of space in a table page that should be reserved for future updates. This PCTFREE specification can be set with CREATE TABLE, ALTER TABLE, DECLARE LOCAL TEMPORARY TABLE, or LOAD TABLE. See "PCTFREE clause" [*SQL Anywhere Server - SQL Reference*].

- **Reorganize tables** You can defragment specific tables using the REORGANIZE TABLE statement or clicking **Reorganize** on the **Fragmentation** tab in Sybase Central. See "REORGANIZE TABLE statement" [*SQL Anywhere Server - SQL Reference*] or "Using the Fragmentation tab (SQL Anywhere plug-in)" on page 217.

- **Rebuild the database** Rebuilding the database defragments all tables, including system tables, *provided the rebuild is performed as a two-step process*, that is, data is unloaded and stored to disk, and then reloaded. Rebuilding in this manner also has the benefit of rearranging the table rows so they appear in the order specified by the clustered index and primary keys. One-step rebuilds (for example, using the -ar, -an, or -ac options), do not reduce table fragmentation. See "Rebuilding databases" on page 731 and "Unload utility (dbunload)" [*SQL Anywhere Server - Database Administration*].

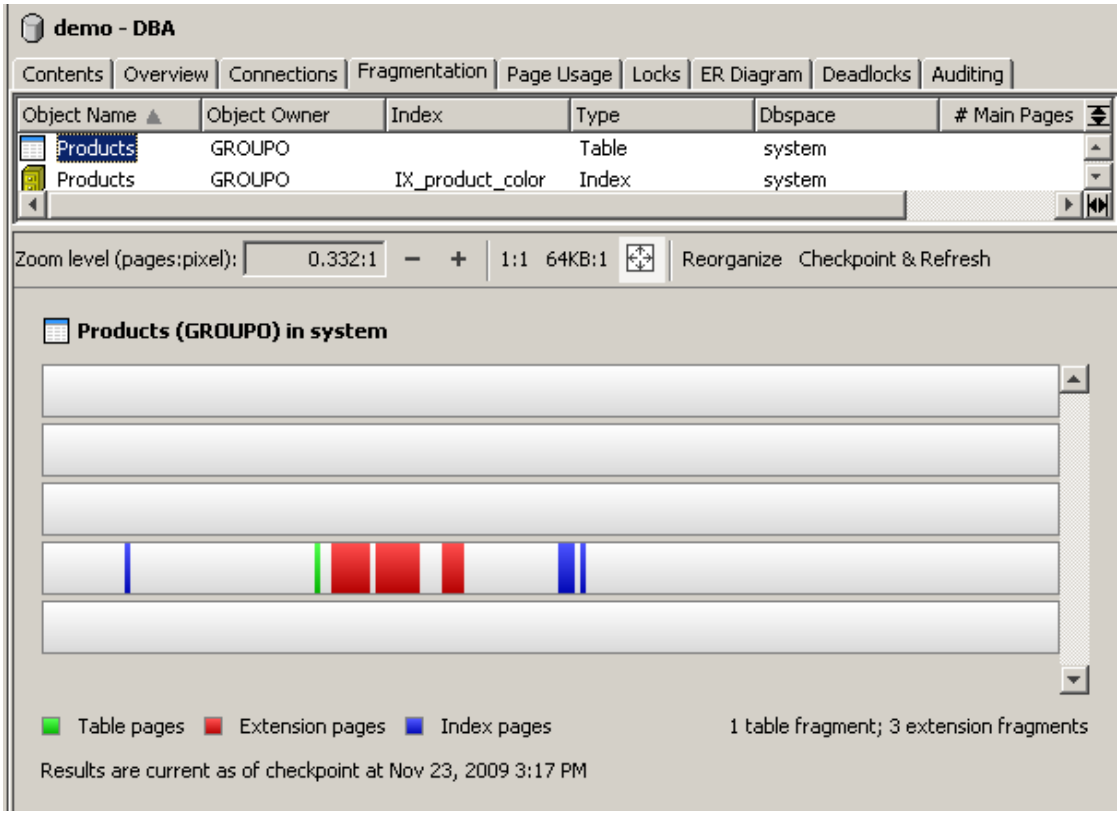
Using the Fragmentation tab (SQL Anywhere plug-in)

You can use the **Fragmentation** tab to:

- View the fragmentation of base tables and indexes on those tables.
- Reorganize tables and indexes.

To view an object's fragmentation details (Sybase Central)

1. Open the **Fragmentation** tab:
   a. Connect to the database.
   b. In the left pane, select the database.
   c. In the right pane, click the **Fragmentation** tab.
2. Select an object from the top pane. The fragmentation information appears in a dbspace map in the bottom pane:
   - When you select a base table, the table, its extension pages, and applicable index pages appear in the dbspace map in the bottom pane.
   - When you select an index, its index pages appear in the dbspace map in the bottom pane.



3. Click **Checkpoint & Refresh** to perform a checkpoint and see the most recent fragmentation information.

4. View the page indexes.
   - In the dbspace map in the bottom pane, hover your cursor over a colored-vertical bar to see the first and last page indexes at that position.
   - In the dbspace map, press and hold the Ctrl key while hovering the mouse over a colored-vertical bar to see all the page indexes at that position.

To reorganize base tables and indexes (Sybase Central)

1. Open the **Fragmentation** tab.

2. Connect to the database as a user with **DBA** authority.

3. In the left pane, select the database.

4. In the right pane, click the **Fragmentation** tab.

5. Select an object from the top pane. The fragmentation information appears in a dbspace map in the bottom pane:

6. Choose one of the following methods to reorganize the object.
   - Click **Reorganize** to execute a REORGANIZE TABLE statement on the selected object.
   - Drag an object from the top pane into an Interactive SQL **SQL Statements** pane. A REORGANIZE TABLE statement for the object appears in the **SQL Statements** pane. Execute the statement.

     This method is useful when you want to reorganize the objects at a later time or when you want to continue using Sybase Central while reorganizing the objects.
   - Select an object from the top pane, right-click, and choose **Copy** to copy a REORGANIZE TABLE statement for the object into the clipboard. Then, in Interactive SQL paste the statement into **SQL Statements** pane and execute the statement.

     This method is useful when you want to reorganize the objects at a later time or when you want to continue using Sybase Central while reorganizing the objects.

   See "REORGANIZE TABLE statement" [*SQL Anywhere Server - SQL Reference*].

Zooming within a dbspace map

By default, when a dbspace map is opened in the bottom pane of the **Fragmentation** tab, the zoom level is set to **Fit To Window**. You can zoom in by clicking the dbspace map and you can zoom out by pressing Shift while clicking. When you click or press Shift while clicking within the dbspace map, the clicked page is centered in the map after the zoom level change.

You can also use the toolbar buttons to zoom to the following levels:

Toolbar button	Definition
1 : 1	**1 page: 1 pixel**
1 64KB: 1	**1 64KB: 1 pixel** (one database read)
Fit To Window	Uses all available space in the window.

Reduce index fragmentation and skew

Indexes are designed to speed up searches on particular columns, but they can become fragmented (less dense) and skewed (unbalanced) if many delete operations are performed on the indexed table.

Index density reflects the average fullness of the index pages. Index skew reflects the typical deviation from the average density. The amount of skew is important to the optimizer when making selectivity estimates.

To determine whether your database contains indexes that contain unacceptable levels of fragmentation or skew, use the **Application Profiling Wizard**. See "Application Profiling Wizard" on page 158.

You can also use the sa_index_fragmentation system procedure to review levels of index fragmentation and skew. For example, the following statement calls the sa_index_density system procedure to examine indexes on the Customers table.

```
CALL sa_index_density( 'Customers' );
```

Table-Name	TableId	IndexName	IndexID	IndexType	LeafPag-es	Density	Skew
Customers	718	CustomersKey	0	PKEY	1	0.127686	1.000000
Customers	718	IX_custom-er_name	1	NUI	1	0.789795	1.000000

SQL Anywhere creates indexes on primary keys automatically. Note that these indexes have an IndexID of 0 in the results for the sa_index_density system procedure.

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 is 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" [*SQL Anywhere Server - SQL Reference*] and "Using the Fragmentation tab (SQL Anywhere plug-in)" on page 217.

You can also use the **Fragmentation** tab in the SQL Anywhere plug-in to review levels of index fragmentation on indexes associated with base tables. See "Using the Fragmentation tab (SQL Anywhere plug-in)" on page 217.

See also

- "sa_index_density system procedure" [*SQL Anywhere Server - SQL Reference*]
- "Working with indexes" on page 56
- "Using the Fragmentation tab (SQL Anywhere plug-in)" on page 217

Reduce primary key width

Wide primary keys are composed of two or more columns. The more columns contained in your primary key, the more demand there is on the database server. Reducing the number of columns in your primary keys can improve performance.

See also

- "Managing primary keys" on page 9
- "Use keys to improve query performance" on page 226

Reduce table widths

Tables where the combined columns (or the size of an individual row) exceeds the database page size and must be split across two or more database pages are referred to as wide table. The more pages a row takes up, the longer the database server takes to read each row. If you have wide tables, and find performance slow consider further normalizing your tables to reduce the number of columns. If that is not possible, a larger database page size may be helpful, especially if most tables are wide. See "Creating a SQL Anywhere database" [*SQL Anywhere Server - Database Administration*].

Reduce requests between client and server

If you find yourself in a situation where your network exhibits poor latency, or your application sends many cursor open and close requests, you can use the LazyClose and PrefetchOnOpen network connection parameters to reduce the number of requests between the client and server and thereby improve performance. See "LazyClose (LCLOSE) connection parameter" [*SQL Anywhere Server - Database Administration*], and "PrefetchOnOpen connection parameter" [*SQL Anywhere Server - Database Administration*].

Reduce expensive user-defined functions

Reducing expensive user-defined functions in queries where they have to be executed many times can improve performance. See "Introduction to user-defined functions" on page 799.

Replace expensive triggers

Evaluate the use of triggers to see if some of the triggers could be replaced by features available in the database server. For instance, triggers to update columns with the latest update time and user information can be replaced with the corresponding special values in the database server. As well, using the default settings on existing triggers can also improve performance. See "Introduction to triggers" on page 803.

Strategic sorting of query results

Reduce the amount of unnecessary sorting of data; unless you need the data returned in a predictable order, do not specify an ORDER BY clause in SELECT statements. Sorting requires extra time and resources to process the query.

For more information about sorting, see "The ORDER BY clause: sorting query results" on page 378, or "The GROUP BY clause: Organizing query results into groups" on page 370.

Specify the correct cursor type

Specifying the correct cursor type can improve performance. For example, if a cursor is read-only, then declaring it as read-only allows for faster optimization and execution, since there is less material to build,

such as check constraints, and so on. If the cursor is updatable, some query rewrites can be skipped. Also, if a query is updatable, then depending on the execution plan chosen by the optimizer, the database server must use a keyset driven approach. Keep in mind that keyset cursors are more expensive. See "Choosing cursor types" [*SQL Anywhere Server - Programming*].

Supply explicit selectivity estimates sparingly

Occasionally, statistics may become inaccurate. This condition is most likely to arise when only a few queries have been executed since a large amount of data was added, updated, or deleted. Inaccurate or unavailable statistics can impede performance. If SQL Anywhere is taking too long to update the statistics, try executing CREATE STATISTICS or DROP STATISTICS to refresh them.

SQL Anywhere also updates some statistics when executing LOAD TABLE statements, during query execution, and when performing update DML statements.

In unusual circumstances, however, these measures may prove ineffective. 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.

Although user defined estimates can sometimes improve performance, avoid supplying explicit user-defined 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 have used selectivity estimates that are inaccurate as a workaround to performance problems where the software-selected access plan was poor, you can set user_estimates to Off to ignore the values. See "Explicit selectivity estimates" [*SQL Anywhere Server - SQL Reference*].

Turn off autocommit mode

If your application runs in **autocommit mode**, then SQL Anywhere treats each of your statements as a separate transaction. In effect, it is equivalent to appending a COMMIT statement to the end of each of your commands.

Instead of running in autocommit mode, consider grouping your commands so each group performs one logical task. If you disable autocommit, you must execute an explicit commit after each logical group of commands. Also, be aware that if logical transactions are large, blocking and deadlock can happen.

If you are not using a transaction log file, the cost of using autocommit mode is high. Every statement forces a checkpoint—an operation that can involve writing numerous pages of information to disk.

Each application interface has its own way of setting autocommit behavior. For the Open Client, ODBC, and JDBC interfaces, Autocommit is the default behavior.

For more information about autocommit, see "Setting autocommit or manual commit mode" [*SQL Anywhere Server - Programming*].

Use an appropriate page size

The page size you choose can affect the performance of your database. There are advantages and disadvantages to both large and small page sizes.

Smaller pages hold less information and may use space less efficiently, particularly if you insert rows that are slightly more than half a page in size. However, small page sizes allow SQL Anywhere to run with fewer resources because more pages can be stored in a cache of the same size. Small pages are useful if your database runs on a small computer with limited memory. They can also help when your database is used primarily for the retrieval of small pieces of information from random locations.

A larger page size helps SQL Anywhere read databases more efficiently. Large page sizes tend to benefit large databases, and queries that perform sequential table scans. Often, the physical design of disks permits them to retrieve fewer large blocks more efficiently than many small ones. Other benefits of large page sizes include improving the fan-out of your indexes, thereby reducing the number of index levels, and allowing tables to include more columns.

Keep in mind that larger page sizes have additional memory requirements. As well, extremely large page sizes (16 KB or 32 KB) are not recommended for most applications unless you can be sure that a large database server cache is always available. Investigate the effects of increased memory and disk space on performance characteristics before using 16 KB or 32 KB page sizes.

The database server's memory usage is proportional to the number of databases loaded, and the page size of the databases. It is strongly recommended that you do performance testing (and testing in general) when choosing a page size. Then choose the smallest page size (>= 4 KB) that gives satisfactory results. It is important to pick the correct (and reasonable) page size if a large number of databases are going to be started on the same server.

You cannot change the page size of an existing database. Instead you must create a new database and use the -p option of dbinit to specify the page size. For example, the following command creates a database with 4 KB pages.

```
dbinit -p 4096 new.db
```

You can also use the CREATE DATABASE statement with a PAGE SIZE clause to create a database with the new page size. See “CREATE DATABASE statement” [*SQL Anywhere Server - SQL Reference*].

For more information about larger page sizes, see “Setting a maximum page size” [*SQL Anywhere Server - Database Administration*].

Scattered reads

If you are working with Windows, a minimum page size of 4 KB allows the database server to read a large contiguous region of database pages on disk directly into the appropriate place in cache, bypassing the 64 KB buffer entirely. This feature can significantly improve performance.

Note
Scattered reads are not used for files on remote computers, or for files specified using a UNC name such as *mycomputer**myshare**mydb.db*.

Use appropriate data types

Data types store information about specific sets of data, including ranges of values, the operations that can be performed on those values, and how the values are stored in memory. You can improve performance by using the appropriate data type for your data. For instance, avoid assigning a data type of CHAR to values that only contain numeric data. And whenever possible, choose efficient data types over the more expensive numeric and string types. See "SQL data types" [*SQL Anywhere Server - SQL Reference*].

Use AUTOINCREMENT to create primary keys

Primary key values must be unique. Although there are a variety of ways to create unique values for primary keys, the most efficient method is setting the default column value to be AUTOINCREMENT. You can use this default for any column in which you want to maintain unique values. Using the AUTOINCREMENT feature to generate primary key values is faster than other methods because the value is generated by the database server. See "CREATE TABLE statement" [*SQL Anywhere Server - SQL Reference*], and "ALTER TABLE statement" [*SQL Anywhere Server - SQL Reference*].

Use bulk operations methods

If you find yourself loading large amounts of information into your database, you can benefit from the special tools provided for these tasks.

If you are loading large files, it is more efficient to create indexes on the table after the data is loaded.

For information about improving bulk operation performance, see "Performance aspects of bulk operations" on page 699.

Use delayed commits

When the rate of committed changes to a database is high, the rate of transaction log writes can be the single largest factor in determining overall database performance. If you are trying to improve transaction log performance, you can set the delayed_commits option to On. When set to On, the database server replies to a COMMIT statement immediately instead of waiting until the transaction log entry for the COMMIT has been written to disk. When set to Off, the application must wait until the COMMIT is written to disk. Turning on the delayed_commits option results in fewer transaction log writes by avoiding multiple re-writes of partially-filled log pages, and you can set the option per connection or for all connections. When the delayed_commits option is turned on, there is a risk that committed operations may be lost if the server goes down before the transaction log pages are flushed to disk. See "delayed_commits option" [*SQL Anywhere Server - Database Administration*].

Use in-memory mode

If your application can tolerate the loss of committed transactions after the most recent checkpoint, then your application may benefit from using in-memory mode.

This mode is useful in applications where increased performance is desirable, and you are running on a system with a large amount of available memory, typically enough to hold all the database files within the cache.

You can choose between two different in-memory modes. In never-write mode, committed transactions are not written to the database file on disk. When you specify never-write mode, multiple concurrent LOAD TABLE statements can be active on the same or different tables. All changes are lost if the database is shut down or the connection is lost. In checkpoint-only mode, the database server does not use a transaction log, and you cannot recover to the most recent committed transaction. However, because the checkpoint log is enabled, the database can be recovered to the most recent checkpoint.

For more information about configuring in-memory mode and determining if it is appropriate for your application, see "-im dbeng12/dbsrv12 server option" [*SQL Anywhere Server - Database Administration*].

Separately licensed component required
In-memory mode requires a separate license. See "Separately licensed components" [*SQL Anywhere 12 - Introduction*].

Use indexes effectively

When executing a query, SQL Anywhere chooses how to access each table. Indexes greatly speed up the access. When the database server cannot find a suitable index, it resorts to scanning the table sequentially —a process that can take a long time.

For example, suppose you need to search a large database for employees, and you only know their first or last name, but not both. If no index exists, SQL Anywhere scans the entire table. If, however, you created two indexes (one that contains the last names first, and a second that contains the first names first), SQL Anywhere scans the indexes first, and can generally return the information to you faster.

Proper selection of indexes can make a large performance difference. Creating and managing indexes is described in "Working with indexes" on page 56.

Using indexes

Although indexes let SQL Anywhere locate information very efficiently, exercise some caution when adding them. Each index creates extra work every time you insert, delete, or update a row because SQL Anywhere must also update all affected indexes.

Consider adding an index when it allows SQL Anywhere to access data more efficiently. In particular, add an index when it eliminates unnecessarily accessing a large table sequentially. If, however, you need better performance when you add rows to a table, and finding information quickly is not an issue, use as few indexes as possible.

You may want to use the Index Consultant to guide you through the selection of an effective set of indexes for your database. See "Index Consultant" on page 164.

Clustered indexes

Using clustered indexes helps store rows in a table in approximately the same order as they appear in the index. See "Indexes" on page 623, and "Using clustered indexes" on page 58.

Use keys to improve query performance

Primary keys and foreign keys, while used primarily for validation purposes, can also improve database performance.

Example

The following example illustrates how primary keys can make queries execute more quickly.

```
SELECT *
FROM Employees
WHERE EmployeeID = 390;
```

The simplest way for the database server to execute this query would be to look at all 75 rows in the Employees table and check the employee ID number in each row to see if it is 390. This does not take very long since there are only 75 employees, but for tables with many thousands of entries a sequential search can take a long time.

The referential integrity constraints embodied by each primary or foreign key are enforced by SQL Anywhere through the help of an index, implicitly created with each primary or foreign key declaration. The EmployeeID column is the primary key for the Employees table. The corresponding primary key index permits the retrieval of employee number 390 quickly. This quick search takes almost the same amount of time whether there are 100 rows or 1000000 rows in the Employees table.

Separate indexes are created automatically for primary and foreign keys. This arrangement allows SQL Anywhere to perform many operations more efficiently.

For more information about how primary and foreign keys work, see "Relationships between tables" [*SQL Anywhere 12 - Introduction*].

Use the cache to improve performance

The database cache is an area of memory used by the database server to store database pages for repeated fast access. The more pages that are accessible in the cache, the fewer times the database server needs to read data from disk. As reading data from disk is a slow operation, the amount of cache available is often a key factor in determining performance.

For systems that are not dedicated database servers, restricting the cache size may improve overall system performance by leaving more memory available for other processes. In general, dynamic cache sizing tunes the cache size appropriately and automatically by monitoring the system as a whole.

You can specify the -c options to control the size of the database cache on the database server command line when the database is started.

The database server messages window displays the size of the cache at startup, and you can use the following statement to obtain the current size of the cache:

```
SELECT PROPERTY( 'CurrentCacheSize' );
```

Encrypted databases must have sufficient cache to minimize I/O operations because these operations are more expensive on encrypted databases than on unencrypted databases since encryption and/or decryption must be performed for each operation.

See also

- "-c dbeng12/dbsrv12 server option" [*SQL Anywhere Server - Database Administration*]
- "-ca dbeng12/dbsrv12 server option" [*SQL Anywhere Server - Database Administration*]
- "-ch dbeng12/dbsrv12 server option" [*SQL Anywhere Server - Database Administration*]
- "-cl dbeng12/dbsrv12 server option" [*SQL Anywhere Server - Database Administration*]

Limit cache memory use

The initial, minimum, and maximum cache sizes are all controllable from the database server command line.

- **Initial cache size** You can specify the initial cache size for the database server by using the -c database server option. If you do not specify the -c option, the database server calculates the initial cache allocation. See "-c dbeng12/dbsrv12 server option" [*SQL Anywhere Server - Database Administration*].

- **Maximum cache size** You can control the maximum cache size by specifying the database server -ch option. The default is based on a heuristic that depends on the physical memory in your computer. On Windows Mobile, the default maximum cache size is the amount of available program memory minus 4 MB. On other non-Unix computers, this is approximately the lower of the maximum non-AWE cache size and 90% of the physical memory of the computer. On Unix, the default maximum cache size is calculated as follows:

  - On 32-bit Unix platforms, it is the lesser of 90% of total physical memory or 1,834,880 KB.
  - On 64-bit Unix platforms, it is the lesser of 90% of total physical memory and 8,589,672,320 KB.

- **Minimum cache size** You can control the minimum cache size by specifying the database server -cl server option. By default, the minimum cache size is the same as the initial cache size, except on Windows Mobile. On Windows Mobile, the default minimum cache size is 600 KB.

  If you specify the -c server option without -cl, then the minimum cache size is set to the initial cache size specified by the -c server option.

  If you do not set the -c or -cl server options, the minimum cache size is set to a very low hard-coded constant value, so that the cache can shrink if necessary. On Windows this value is 2 MB, on Unix it is 8 MB, and on Windows Mobile it is 600 KB.

You can also disable dynamic cache sizing by using the -ca 0 server option.

The following database server properties return information about the database server cache:

- **CurrentCacheSize** Returns the current cache size, in kilobytes.
- **MinCacheSize** Returns the minimum allowed cache size, in kilobytes.
- **MaxCacheSize** Returns the maximum allowed cache size, in kilobytes.
- **PeakCacheSize** Returns the largest value the cache has reached in the current session, in kilobytes.

For information about obtaining server property values, see "Database server properties" [*SQL Anywhere Server - Database Administration*].

See also

- "Cache size" [*SQL Anywhere Server - Database Administration*]
- "-c dbeng12/dbsrv12 server option" [*SQL Anywhere Server - Database Administration*]
- "-ca dbeng12/dbsrv12 server option" [*SQL Anywhere Server - Database Administration*]
- "-ch dbeng12/dbsrv12 server option" [*SQL Anywhere Server - Database Administration*]
- "-cl dbeng12/dbsrv12 server option" [*SQL Anywhere Server - Database Administration*]

Dynamic cache sizing

You can use SQL Anywhere to automatically resize the database cache while the database server is running. However, the effectiveness of dynamic cache sizing is limited by the operating system on which the database server is running and by the amount of available physical memory.

With full **dynamic cache sizing**, the cache grows when the database server can usefully use more, as long as memory is available, and shrinks when cache memory is required by other applications.

Typically, the cache requirements are assessed by dynamic cache sizing once per minute. However, the assessment frequency may increase to once every five seconds for thirty seconds when a new database is started or when a file grows significantly. After the initial thirty second period, the sampling rate returns to once per minute. File growth of 1/8 since the database started or since the last growth that triggered an increase in the sampling rate is considered significant. This change improves performance further by adapting the cache size more quickly when databases are started dynamically and when a lot of data is inserted.

With dynamic cache sizing you do not need to explicitly configure the database cache.

When an Address Windowing Extensions (AWE) cache is used, dynamic cache sizing is disabled. Only the 32-bit Windows database server can use an AWE cache. See "-cw dbeng12/dbsrv12 server option (deprecated)" [*SQL Anywhere Server - Database Administration*].

Note
The use of AWE is deprecated. It is recommended that you use the 64-bit version of the SQL Anywhere database server on a 64-bit Windows operating system if you require a large cache.

See also

- "-ca dbeng12/dbsrv12 server option" [*SQL Anywhere Server - Database Administration*]
- "-ch dbeng12/dbsrv12 server option" [*SQL Anywhere Server - Database Administration*]
- "-cl dbeng12/dbsrv12 server option" [*SQL Anywhere Server - Database Administration*]
- "Cache size" [*SQL Anywhere Server - Database Administration*]

Dynamic cache sizing on Windows

On Windows and Windows Mobile, the database server evaluates cache and operating statistics once per minute and computes an optimum cache size. The database server computes a target cache size that uses all physical memory currently not in use, except for approximately 5 MB that is to be left free for system use. The target cache size is never smaller than the specified or implicit minimum cache size. The target cache size never exceeds the specified or implicit maximum cache size, or the sum of the sizes of all open database and temporary files plus the size of the main heap.

To avoid cache size oscillations, the database server increases the cache size incrementally. Rather than immediately adjusting the cache size to the target value, each adjustment modifies the cache size by 75% of the difference between the current and target cache size.

Windows can use Address Windowing Extensions (AWE) to support large cache sizes by specifying the -cw command line option when starting the database server. AWE caches do not support dynamic cache sizing. Windows Mobile does not support AWE caches. See "-cw dbeng12/dbsrv12 server option (deprecated)" [*SQL Anywhere Server - Database Administration*].

Note
The use of AWE is deprecated. It is recommended that you use the 64-bit version of the SQL Anywhere database server on a 64-bit Windows operating system if you require a large cache.

Dynamic cache sizing on Unix

On Unix, the database server uses swap space and memory to manage the cache size. The swap space is a system-wide resource on most Unix operating systems, but not on all. In this section, the sum of memory and swap space is called the **system resources**. See your operating system documentation for details.

On startup, the database allocates the specified maximum cache size from the system resources. It loads some of this into memory (the initial cache size) and keeps the remainder as swap space.

The total amount of system resources used by the database server is constant until the database server shuts down, but the proportion loaded into memory changes. Each minute, the database server evaluates cache and operating statistics. If the database server is busy and demanding of memory, it may move cache pages from swap space into memory. If the other processes in the system require memory, the database server may move cache pages out from memory to swap space.

Initial cache size

By default, the initial cache size is assigned using an heuristic based on the available system resources. The initial cache size is always less than 1.1 times the total database size.

If the initial cache size is greater than 3/4 of the available system resources, the database server exits with a `Not Enough Memory` error.

You can change the initial cache size using the -c option. See "-c dbeng12/dbsrv12 server option" [*SQL Anywhere Server - Database Administration*].

Maximum cache size

The maximum cache must be less than the available system resources on the computer. By default, the maximum cache size is assigned using an heuristic based on the available system resources and the total physical memory on the computer. The cache size never exceeds the specified or implicit maximum cache size, or the sum of the sizes of all open database and temporary files plus the size of the main heap.

If you specify a maximum cache size greater than the available system resources, the database server exits with a `Not Enough Memory` error. If you specify a maximum cache size greater than the available memory, the database server warns of performance degradation, but does not exit.

The database server allocates all the *maximum* cache size from the system resources, and does not relinquish it until the database server exits. You should be sure that you choose a maximum cache size that gives good SQL Anywhere performance while leaving space for other applications. The formula for the default maximum cache size is an heuristic that attempts to achieve this balance. You only need to tune the value if the default value is not appropriate on your system.

You can use the -ch server option to set the maximum cache size, and limit automatic cache growth. For more information, see "-ch dbeng12/dbsrv12 server option" [*SQL Anywhere Server - Database Administration*].

Minimum cache size

If the -c option is specified, the minimum cache size is the same as the initial cache size. If no -c option is specified, the minimum cache size on Unix is 8 MB.

You can use the -cl server option to adjust the minimum cache size. See "-cl dbeng12/dbsrv12 server option" [*SQL Anywhere Server - Database Administration*].

Monitoring cache size

The following statistics are included in the Windows Performance Monitor and the database's property functions.

- **CurrentCacheSize** The current cache size in kilobytes
- **MinCacheSize** The minimum allowed cache size in kilobytes
- **MaxCacheSize** The maximum allowed cache size in kilobytes

- **PeakCacheSize** The peak cache size in kilobytes

For more information about these properties, see "Database server properties" [*SQL Anywhere Server - Database Administration*].

For information about monitoring performance, see "Monitoring database performance" on page 192.

Using cache warming

Cache warming is designed to help reduce the execution times of the initial queries executed against a database. This is done by preloading the database server's cache with database pages that were referenced the last time the database was started. Warming the cache can improve performance when the same query or similar queries are executed against a database each time it is started.

You control the cache warming settings on the database server command line. There are two activities that can take place when a database is started and cache warming is turned on: collection of database pages and cache reloading (warming).

Collection of referenced database pages is controlled by the -cc database server option, and is turned on by default. When database page collection is turned on, the database server keeps track of every database page that is requested from database startup until one of the following occurs: the maximum number of pages has been collected (the value is based on cache size and database size), the collection rate falls below the minimum threshold value, or the database is shut down. Note that the database server controls the maximum number of pages and the collection threshold. Once collection completes, the referenced pages are recorded in the database so they can be used to warm the cache the next time the database is started.

Cache warming (reloading) is turned on by default, and is controlled by the -cr database server option. To warm the cache, the database server checks whether the database contains a previously recorded collection of pages. If it does, the database server loads the corresponding pages into the cache. The database can still process requests while the cache is loading pages, but warming may stop if a significant amount of I/O activity is detected in the database. Cache warming stops in this case to avoid performance degradation of queries that access pages that are not contained in the set of pages being reloaded into the cache. You can specify the -cv option if you want messages about cache warming to appear in the database server messages window.

See also

- "-cc dbeng12/dbsrv12 server option" [*SQL Anywhere Server - Database Administration*]
- "-cr dbeng12/dbsrv12 server option" [*SQL Anywhere Server - Database Administration*]
- "-cv dbeng12/dbsrv12 server option" [*SQL Anywhere Server - Database Administration*]

Use compression carefully

Compression of packets for connections

Enabling compression for one connection or all connections, and adjusting the minimum size limit at which packets are compressed can offer significant improvements to performance under some circumstances.

To determine if enabling compression is beneficial, conduct a performance analysis on your network and using your application before using communication compression in a production environment.

Enabling compression increases the quantity of information stored in data packets, thereby reducing the number of packets required to transmit a particular set of data. By reducing the number of packets, the data can be transmitted more quickly.

Specifying the compression threshold allows you to choose the minimum size of data packets that you want compressed. The optimal value for the compression threshold may be affected by a variety of factors, including the type and speed of network you are using.

Database compression

The use of file-level or disk-level compression for database and log files is not recommended since the compression layer may significantly increase the cost of IO operations.

See also

- "Adjusting communication compression settings to improve performance" [*SQL Anywhere Server - Database Administration*]
- "Compress (COMP) connection parameter" [*SQL Anywhere Server - Database Administration*]
- "CompressionThreshold (COMPTH) connection parameter" [*SQL Anywhere Server - Database Administration*]

Use the WITH EXPRESS CHECK option when validating tables

If you find that validating large databases with a small cache takes a long time, you can use one of two options to reduce the amount of time it takes. Using the WITH EXPRESS CHECK option with the VALIDATE TABLE statement, or the -fx option with the Validation utility (dbvalid) can significantly increase the speed at which your tables validate.

See also

- "Improving performance when validating databases" [*SQL Anywhere Server - Database Administration*]
- "VALIDATE statement" [*SQL Anywhere Server - SQL Reference*]
- "Validation utility (dbvalid)" [*SQL Anywhere Server - Database Administration*]

Use work tables in query processing (use All-rows optimization goal)

Work tables are materialized temporary result sets that are created during the execution of a query. Work tables are used when the cost is less than the alternative strategies. Generally, the time to fetch the first few rows is higher when a work table is used, but the cost of retrieving all rows may be substantially lower if a work table can be used. Because of this difference, SQL Anywhere chooses different strategies based on the optimization_goal setting. The default is first-row. When it is set to first-row, SQL Anywhere tries to avoid work tables. When it is set to All-rows, SQL Anywhere uses work tables when they reduce the total execution cost of a query.

For more information about the optimization_goal setting, see "optimization_goal option" [*SQL Anywhere Server - Database Administration*].

Work tables are used in the following cases:

- when a query has an ORDER BY, GROUP BY, or DISTINCT clause, and SQL Anywhere does not use an index for sorting the rows. If a suitable index exists and the optimization_goal setting is First-row, SQL Anywhere avoids using a work table. However, when optimization_goal is set to All-rows, it may be more expensive to fetch all the rows of a query using an index than it is to build a work table and sort the rows. SQL Anywhere chooses the cheaper strategy if the optimization goal is set to All-rows. For GROUP BY and DISTINCT, the hash-based algorithms use work tables, but are generally more efficient when fetching all the rows out of a query.
- when a hash join algorithm is chosen. In this case, work tables are used to store interim results (if the input doesn't fit into memory) and a work table is used to store the results of the join.
- when a cursor is opened with sensitive values. In this case, a work table is created to hold the row identifiers and primary keys of the base tables. This work table is filled in as rows are fetched from the query in the forward direction. However, if you fetch the last row from the cursor, the entire table is filled in.
- when a cursor is opened with insensitive semantics. In this case, a work table is populated with the results of the query when the query is opened.
- when a multiple-row UPDATE is being performed and the column being updated appears in the WHERE clause of the update or in an index being used for the update
- when a multiple-row UPDATE or DELETE has a subquery in the WHERE clause that references the table being modified
- when performing an INSERT from a SELECT statement and the SELECT statement references the insert table
- when performing a multiple row INSERT, UPDATE, or DELETE, and a corresponding trigger is defined on the table that may fire during the operation

In these cases, the records affected by the operation go into the work table. In certain circumstances, such as keyset-driven cursors, a temporary index is built on the work table. The operation of extracting the required records into a work table can take a significant amount of time before the query results appear. Creating indexes that can be used to do the sorting in the first case, above, improves the time to retrieve the first few rows. However, the total time to fetch all rows may be lower if work tables are used, since these permit query algorithms based on hashing and merge sort. These algorithms use sequential I/O, which is faster than the random I/O used with an index scan.

The optimizer analyzes each query to determine whether a work table would give the best performance. No user action is required to take advantage of these optimizations.

Notes

The INSERT, UPDATE, and DELETE cases above are usually not a performance problem since they are usually one-time operations. However, if problems occur, you may be able to rephrase the command to avoid the conflict and avoid building a work table. This is not always possible.

Application profiling tutorials

Use the application profiling tutorials to learn how to use the **Application Profiling Wizard** and the **Database Tracing Wizard** to analyze common performance problems including deadlocks, slow statements, index fragmentation, table fragmentation, and slow procedures.

Caution
The application profiling tutorials use the test database *app_profiling.db* which you create, not the sample database (*demo.db*). Do not use the sample database to complete the tutorials.

Create a test database for the application profiling tutorials

Use the following procedure to create the test database *app_profiling.db*. The test database is used in all of the application profiling tutorials.

To create the test database

1. Create the directory *c:\AppProfilingTutorial*.
2. Run the following command to create the test database *app_profiling.db* that contains data from the sample database:

```
newdemo c:\AppProfilingTutorial\app_profiling.db
```

See also

- "Application Profiling Wizard" on page 158
- "Advanced application profiling using diagnostic tracing" on page 169
- "Unload utility (dbunload)" [*SQL Anywhere Server - Database Administration*]

Tutorial: Diagnosing deadlocks

Use the lessons in this tutorial to learn how to use the **Database Tracing Wizard** to view deadlocks that might occur in your database. You can use the **Database Tracing Wizard** to examine the conditions under which the deadlocks are occurring, and the connections that are causing them.

Lesson 1: Set up steps for creating a deadlock

Deadlocks occur when two or more transactions block one another. For example, Transaction A requires access to Table B, but Table B is locked by Transaction B. Transaction B requires access to Table A, but Table A is locked by Transaction A. A cyclical blocking conflict occurs.

A good indication that deadlocks are occurring is when SQLCODE -306 and -307 are returned. To resolve a deadlock, SQL Anywhere automatically rolls back the last statement that created the deadlock. Performance problems occur if statements are constantly rolled back.

To create a deadlock

1. This tutorial assumes you have created the test database, *app_profiling.db*. If you have not, see "Create a test database for the application profiling tutorials" on page 234.

2. Connect to *app_profiling.db* as follows:

   a. In Sybase Central, in the SQL Anywhere 12 plug-in, choose **Connections** » **Connect With SQL Anywhere 12**.

   b. In the **Connect** window, complete the following fields to connect to the test database, *app_profiling.db*, and then click **Connect**:

      - **Authentication** **Database**
      - **User ID** **DBA**
      - **Password** **sql**
      - **Action** **Start A Database On This Computer**
      - **Database File** **C:\AppProfilingTutorial\app_profiling.db**
      - **Start Line** **dbeng12 -x tcpip**

3. In the left pane, click **app_profiling - DBA**, and then choose **File** » **Open Interactive SQL**.

   Interactive SQL starts and connects to the *app_profiling.db* database.

4. In Interactive SQL:

   a. Execute the following SQL statements to create two tables you will use later to create the deadlock:

```
CREATE TABLE "DBA"."deadlock1" (
   "id" UNSIGNED BIGINT NOT NULL DEFAULT AUTOINCREMENT,
   "val" CHAR(1) );
CREATE TABLE "DBA"."deadlock2" (
   "id" UNSIGNED BIGINT NOT NULL DEFAULT AUTOINCREMENT,
   "val" CHAR(1) );
```

   b. Execute the following SQL statements to insert values into each table:

```
INSERT INTO "deadlock1"("val") VALUES('x');
INSERT INTO "deadlock2"("val") VALUES('x');
```

   c. Execute the following SQL statements to create two procedures that you will use later to cause the deadlock:

```
CREATE PROCEDURE "DBA"."proc_deadlock1"(  )
   BEGIN
```

```
        LOCK TABLE "DBA"."deadlock1" IN EXCLUSIVE MODE;
        WAITFOR DELAY '00:00:20:000';
        UPDATE deadlock2 SET val='y';
    END;
CREATE PROCEDURE "DBA"."proc_deadlock2"(  )
    BEGIN
        LOCK TABLE "DBA"."deadlock2" IN EXCLUSIVE MODE;
        WAITFOR DELAY '00:00:20:000';
        UPDATE deadlock1 SET val='y';
    END;
```

d. Execute the following SQL statements to commit the changes you made to the database:

```
COMMIT;
```

5. Close Interactive SQL.

6. Click this link to continue the tutorial: "Lesson 2: Create a deadlock and capture information about it" on page 236.

See also

- "Connect to a local database" [*SQL Anywhere Server - Database Administration*]

Lesson 2: Create a deadlock and capture information about it

The **Database Tracing Wizard** can be used to create a diagnostic tracing session. The tracing session captures deadlock data.

> **Tip**
> In the application profiling tutorials, tracing information is stored in the test database (*app_profiling.db*), which is the same database you are running the tutorials on. However, if you profile a database that experiences heavy loads, you should consider storing tracing data in a separate database than your production database to avoid impacting performance.

To capture deadlock data

1. In Sybase Central, choose **Mode** » **Application Profiling**.

   If the **Application Profiling Wizard** appears, click **Cancel**.

2. Start the **Database Tracing Wizard** as follows:
   a. In the left pane, click **app_profiling - DBA**, and choose **File** » **Tracing**.
   b. On the **Welcome** page, click **Next**.
   c. On the **Tracing Detail Level** page, select **High Detail (Recommended For Short-Term, Intensive Monitoring)**, and then click **Next**.
   d. On the **Edit Tracing Levels** page, click **Next**.
   e. On the **Create External Database** page, select **Do Not Create A New Database. I Will Use An Existing Tracing Database**, and then click **Next**.

f. On the **Start Tracing** page, select **Save Tracing Data In This Database**.

g. To place no limits on the amount of stored tracing data, select **No Limit**, and then click **Finish**.

3. Create the deadlock as follows:

a. In the left pane, click **app_profiling - DBA**, and then choose **File** » **Open Interactive SQL**.

Interactive SQL starts and connects to the *app_profiling.db* database.

b. Repeat the previous step to open a second Interactive SQL window.

c. In the first Interactive SQL window, run the following SQL statement:

```
CALL "DBA"."proc_deadlock1"();
```

d. In the second Interactive SQL window, run the following SQL statement within 20 seconds of running the SQL statement in the other Interactive SQL window:

```
CALL "DBA"."proc_deadlock2"();
```

After a few moments, an **ISQL Error** window appears indicating that a deadlock has been detected.

e. The deadlock occurred because proc_deadlock1 requires access to the deadlock2 table, which is locked by proc_deadlock2. At the same time, proc_deadlock2 requires access to the deadlock1 table, which is locked by proc_deadlock1.

f. Click **OK**.

4. SQL Anywhere stopped the deadlocked operations, so you can close the Interactive SQL windows.

5. In Sybase Central, stop the tracing session by clicking the **app_profiling - DBA** connection in the left pane and choosing **File** » **Tracing** » **Stop Tracing With Save**.

6. Click this link to continue the tutorial: "Lesson 3: Review blocked connection data" on page 237.

See also

- "Transaction blocking and deadlock" on page 107
- "Choosing a diagnostic tracing level" on page 171
- "Deadlock" on page 108
- "Advanced application profiling using diagnostic tracing" on page 169

Lesson 3: Review blocked connection data

The **Application Profiling** mode provides a graphical representation of the connections participating in the deadlock. It also provides a **Connection Blocks** tab that provides additional information on the blocked connections.

To review blocked connection data

1. Open the analysis file created during the tracing session as follows:

a. In Sybase Central, choose **Application Profiling** » **Open Analysis File Or Connect To A Tracing Database**.

b. Select **In A Tracing Database**.

c. Click **Open**.

d. In the **User ID** field, type **DBA**.

e. In the **Password** field, type **sql**.

f. From the **Action** dropdown list, choose **Connect To A Running Database On This Computer**.

g. In the **Database Name** field, type **app_profiling**.

h. Click **Connect**.

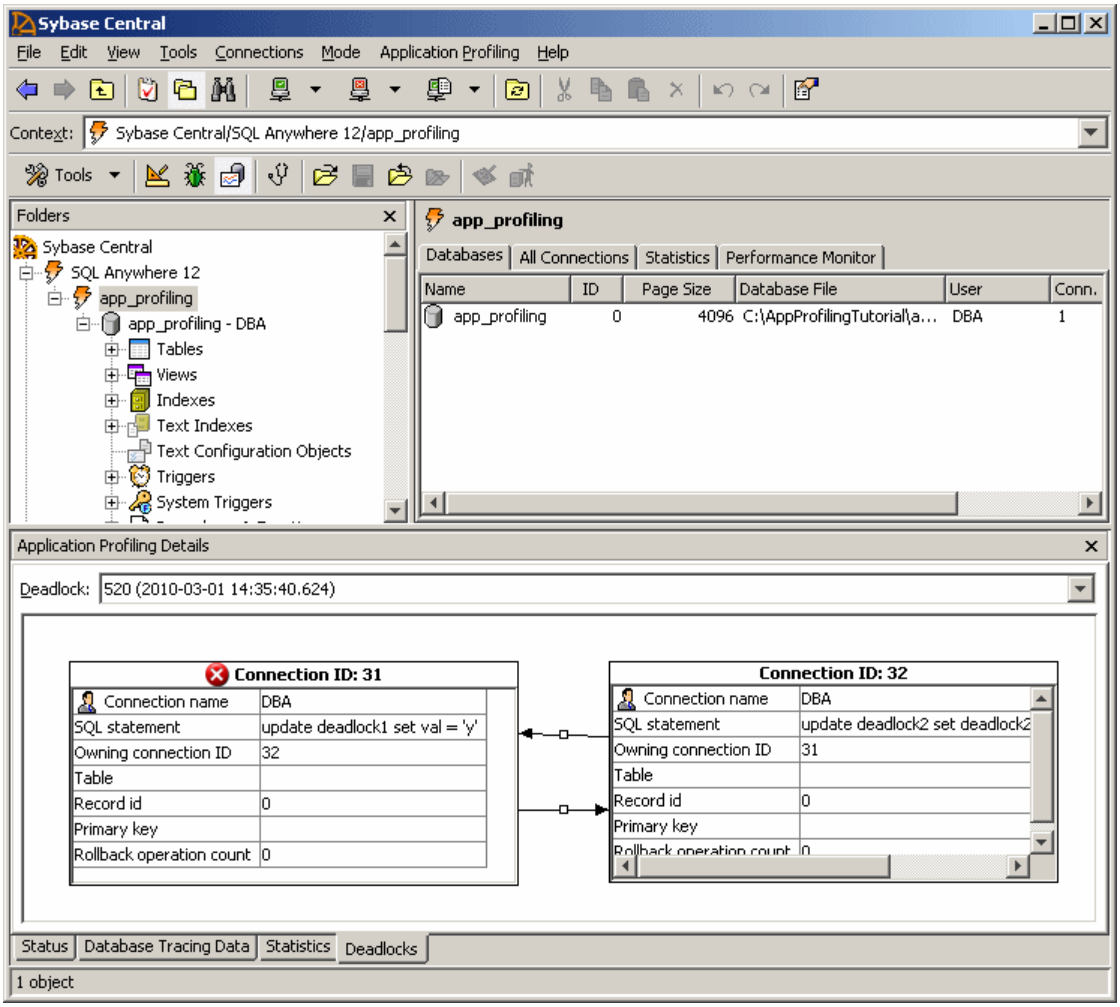
2. View the graphical representation of the deadlock as follows:

   a. In the **Application Profiling Details** pane click the **Status** tab and choose the most recent ID from the **Logging Session ID** list.

      If the **Application Profiling Details** pane does not appear, choose **View** » **Application Profiling Details**.

   b. At the bottom of the **Application Profiling Details** pane, click the **Deadlocks** tab. The most recent deadlock appears. Click the **Deadlock** list to view additional deadlocks.

   c. The following image shows how the UPDATE statements created a deadlock condition.



Each connection involved in the deadlock is represented by a table with the following fields:

- **Connection Name** This field shows the user ID that opened the connection.
- **SQL Statement** This field shows the actual statement involved in the deadlock. In this case, the deadlock was caused by the UPDATE statements found in the procedures you executed from each instance of Interactive SQL.
- **Owning Connection ID** This field shows the ID of the connection that blocked the current connection.
- **Record ID** This field shows the ID of the row that the current connection is blocked on.
- **Rollback Operation Count** This field shows the number of operations that must be rolled back as a result of the deadlock. In this case, the procedures contained only the UPDATE statements, so the count is 0.

3. To view additional deadlock information, such as how often they occur and how long they last, use the **Connection Blocks** tab, as follows:

a. In the **Application Profiling Details** pane, click the **Database Tracing Data** tab.

b. Click the **Connection Blocks** tab, just above the **Database Tracing Data** tab.

c. The **Connection Blocks** tab appears, displaying the block time, unblock time, and duration of each blocked connection.

4. You have completed the application profiling tutorial on deadlocks. Choose **Connections** » **Disconnect**, and then close Sybase Central.

See also

- "Transaction blocking and deadlock" on page 107
- "Choosing a diagnostic tracing level" on page 171
- "Deadlock" on page 108
- "Advanced application profiling using diagnostic tracing" on page 169

Tutorial: Diagnosing slow statements

Use the lessons in this tutorial to learn how to use the **Database Tracing Wizard** to view execution times for statements, and how to identify statements that appear to run slowly.

A slow statement occurs when the database server takes a long time to process the statement. Long processing times can be the result of several issues, such as an improperly designed database, poor use of indexes, index and table fragmentation, or a small cache size. A statement may also run slowly because it is not well formed, or does not use more efficient shortcuts to achieve results.

This tutorial does not show you how to rewrite slow statements since each statement can have special requirements. However, the tutorial does show you where to look for execution times, and how to compare execution times when rewriting queries using alternate syntax.

See also

- "Querying data" on page 255
- "Joins: Retrieving data from several tables" on page 386
- "Using subqueries" on page 491
- "Choosing a diagnostic tracing level" on page 171
- "Advanced application profiling using diagnostic tracing" on page 169

Lesson 1: Create a diagnostic tracing session

The **Database Tracing Wizard** is used to create a diagnostic tracing session. The tracing session captures processing statement data which includes duration times.

To create a diagnostic tracing session

1. This tutorial assumes you have created the test database, *app_profiling.db*, required for the application profiling tutorials. If you have not, see "Create a test database for the application profiling tutorials" on page 234.

2. Connect to *app_profiling.db* as follows:
   a. In Sybase Central, in the SQL Anywhere 12 plug-in, choose **Connections** » **Connect With SQL Anywhere 12**.
   b. In the **Connect** window, complete the following fields to connect to the test database, *app_profiling.db*, and then click **Connect**:
      - **Authentication** **Database**
      - **User ID** **DBA**
      - **Password** **sql**
      - **Action** **Start A Database On This Computer**
      - **Database File** **C:\AppProfilingTutorial\app_profiling.db**
      - **Start Line** **dbeng12 -x tcpip**
3. Start the **Database Tracing Wizard** as follows:
   a. In Sybase Central, choose **Mode** » **Application Profiling**. If the **Application Profiling Wizard** appears, click **Cancel**.
   b. Choose **File** » **Tracing** » **Configure And Start Tracing**.
   c. On the **Welcome** page, click **Next**.
   d. On the **Tracing Detail Level** page, select **High Detail (Recommended For Short-Term, Intensive Monitoring)**, and then click **Next**.
   e. On the **Edit Tracing Levels** page, click **Next**.
   f. On the **Create External Database** page, select **Do Not Create A New Database**, and then click **Next**.
   g. On the **Start Tracing** page, select **Save Tracing Data In This Database**.
   h. To allow no limits on the amount of stored tracing data, select **No Limit**, and then click **Finish**.
4. In the left pane, click **app_profiling - DBA**, and then choose **File** » **Open Interactive SQL**.

   Interactive SQL starts and connects to the *app_profiling.db* database.
5. In Interactive SQL, run the following SQL statement.

```
SELECT SalesOrderItems.ID, LineID, ProductID, SalesOrderItems.Quantity,
ShipDate
FROM SalesOrderItems, SalesOrders
WHERE SalesOrders.CustomerID = 105 AND
   SalesOrderItems.ID=SalesOrders.ID;
```

6. Exit Interactive SQL.
7. To stop the tracing session, in Sybase Central click **app_profiling - DBA** and choose **File** » **Tracing** » **Stop Tracing With Save**.
8. Click this link to continue the tutorial: "Lesson 2: Review statements processed by the database server" on page 242.

See also

- "Choosing a diagnostic tracing level" on page 171
- "Advanced application profiling using diagnostic tracing" on page 169

Lesson 2: Review statements processed by the database server

You can identify which statements the database server spends the most time processing by using the **Summary** and **Detail** tabs, located on the **Application Profiling** pane in Sybase Central.

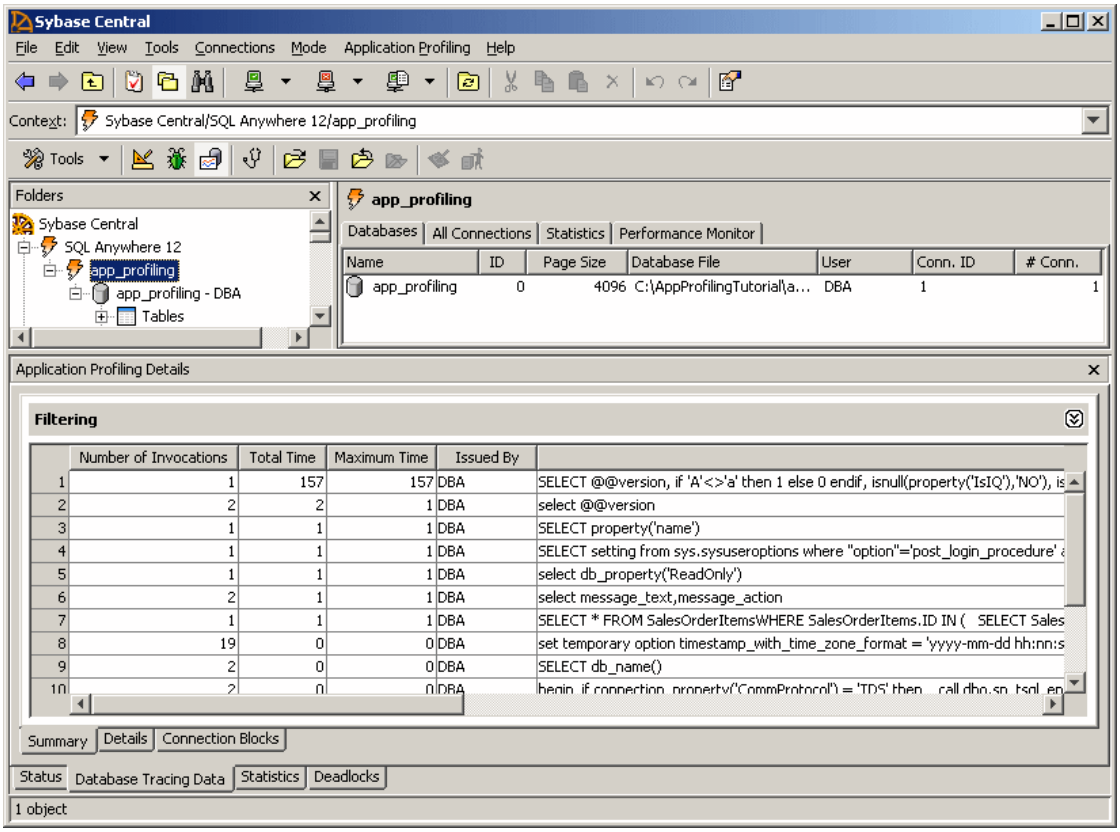
To review statements processed by the database server

1. Open the analysis file as follows:
   a. In Sybase Central, choose **Mode** » **Application Profiling**. If the **Application Profiling Wizard** appears, click **Cancel**.
   b. Choose **Application Profiling** » **Open Analysis File Or Connect To A Tracing Database**.
   c. Select **In A Tracing Database**, and then click **Open**.
   d. In the **User ID** field, type **DBA**.
   e. In the **Password** field, type **sql**.
   f. From the **Action** dropdown list, choose **Connect To A Running Database On This Computer**.
   g. In the **Database Name** field, type **app_profiling**.
   h. Click **Connect**.
   i. If the **Application Profiling Details** pane does not appear at the bottom of the window, choose **View** » **Application Profiling Details**.
2. Examine the statement execution times of statements that were processed during the tracing session, as follows:
   a. On the **Status** tab in the **Application Profiling Details** pane, select the most recent ID (highest number) from the **Logging Session ID** field, and then click the **Database Tracing Data** tab.

      On the **Summary** tab, the SQL statements you executed during the session appear. You may see more additional statements as well. This is because statements you executed automatically caused other statements to be executed (for example, a trigger).

      The **Summary** tab groups similar statements together and summarizes the total number of invocations and the total time spent processing them. SELECT, INSERT, UPDATE, and DELETE statements are grouped together by what tables, columns, and expressions they reference. Other statements are grouped together as a whole (for example, all CREATE TABLE statements appear as a single entry in the **Summary** tab). A statement may appear expensive in the **Summary** tab because it is an expensive statement, or because it is frequently executed.

      Find statements that are running slowly on your system by examining the **Total Time** and **Maximum Time** columns. These provide information about the execution times for each statement processed by the database server.



3. To view information about the connection that executed a statement, go to the **Details** tab, right-click the statement and choose **View Connection Details For The Selected Statement**.

4. To view the execution plan used for a SQL statement, go to the **Details** tab, right-click the statement and choose **View More SQL Statement Details For The Selected Statement**.

   The **SQL Statement Details** window appears, displaying the full text of the statement along with details about the context in which it was used. The text displayed for the statement may not match the original SQL statement you executed. Instead, the **SQL Statement Details** window displays the statement in its rewritten form, as it was processed by the database server. For example, queries over views may appear very different, since the view definitions are often rewritten by the optimizer when executing the query.

   Click the **Query Information** tab at the bottom of the **SQL Statement Details** window to see the execution plan.

5. You have completed the tutorial on diagnosing slow statements.

See also

- "Using subqueries" on page 491
- "Perform request trace analysis" on page 185
- "Querying data" on page 255
- "Joins: Retrieving data from several tables" on page 386
- "Using subqueries" on page 491
- "Choosing a diagnostic tracing level" on page 171
- "Advanced application profiling using diagnostic tracing" on page 169

Tutorial: Diagnosing index fragmentation

Use the lessons in this tutorial to learn how to use the **Application Profiling Wizard** to determine if your database has unacceptable levels of index fragmentation.

When an index is created, table data is read and values for the index are recorded on index pages following a logical order. As data changes in the table, new index values can be inserted between existing values. To maintain the logical order of index values, the database server may need to create new index pages to accommodate existing values that are moved. The new pages are not usually adjacent to the pages on which the values were originally stored. This cumulative degradation in the order of index pages is called index fragmentation.

Commonly executed queries taking longer to perform on tables where large blocks of rows are continuously being inserted, updated, and deleted is a symptom of index fragmentation.

Lesson 1: Find and fix index fragmentation using the Application Profiling Wizard

Use this procedure to find and fix index fragmentation. It is recommended that you periodically check for fragmentation on your production database.

To find and fix index fragmentation

1. This tutorial assumes you have created the test database, *app_profiling.db*, required to use the application profiling tutorials. If you have not, see "Create a test database for the application profiling tutorials" on page 234.

2. Connect to *app_profiling.db* as follows:
   a. In Sybase Central, in the SQL Anywhere 12 plug-in, choose **Connections** » **Connect With SQL Anywhere 12**.
   b. In the **Connect** window, complete the following fields to connect to the test database, *app_profiling.db*, and then click **Connect**:
      - **User ID** **DBA**
      - **Password** **sql**

- **Action** **Start A Database On This Computer**
- **Database File** **C:\AppProfilingTutorial\app_profiling.db**
- **Start Line** **dbeng12 -x tcpip**

3. Choose **Mode** » **Application Profiling**.

   If the **Application Profiling Wizard** does not appear, choose **Application Profiling** » **Open Application Profiling Wizard**.

4. On the **Welcome** page, click **Next**.

5. On the **Profiling Options** page, select **Overall Database Performance Based On The Database Schema**, and then click **Next**.

6. On the **Analysis File** page, in the **Save The Analysis To The Following File** field, type *C:\AppProfilingTutorial\analysis*.

7. Click **Finish**.

   A list of recommendations appears in the **Application Profiling Details** pane.

8. If you see **Fragmented Indexes**, double-click it. A **Recommendation** window appears containing a SQL statement you can run to resolve the index fragmentation.

9. You have completed the tutorial on using application profiling to identify and fix index fragmentation.

> **Note**
> You can also identify and fix index fragmentation using SQL. See "Identifying and fixing index fragmentation using SQL" on page 245.

See also

- "ALTER INDEX statement" [*SQL Anywhere Server - SQL Reference*]
- "sa_index_density system procedure" [*SQL Anywhere Server - SQL Reference*]
- "REORGANIZE TABLE statement" [*SQL Anywhere Server - SQL Reference*]
- "Rebuild indexes" on page 62
- "Reduce index fragmentation and skew" on page 219
- "Application profiling" on page 158
- "Choosing a diagnostic tracing level" on page 171
- "Advanced application profiling using diagnostic tracing" on page 169

Identifying and fixing index fragmentation using SQL

You can also identify and fix index fragmentation using SQL.

To check the index density of a table

1. In the left pane, click **app_profiling - DBA**, and then choose **File** » **Open Interactive SQL**.

Interactive SQL starts and connects to the *app_profiling.db* database.

2. In Interactive SQL, run the following SQL statements to test the index density on the Employees table:

```
CALL sa_index_density( 'Employees' );
```

Density values range between 0 and 1. Values closer to 1 indicate little index fragmentation. Values less than 0.5 indicate a level of index fragmentation that may impact performance.

Note
The values for the indexes on the Employees will appear to show fragmentation issues because the values in the Density column of the results are well under 0.5. However, these numbers are artificially low due to the fact that the table is very small.

3. In Interactive SQL, run an ALTER INDEX ... REBUILD statement similar to the following to improve the density of an index:

```
ALTER INDEX PRIMARY KEY ON Employees REBUILD;
```

See also

- "ALTER INDEX statement" [*SQL Anywhere Server - SQL Reference*]
- "sa_index_density system procedure" [*SQL Anywhere Server - SQL Reference*]
- "REORGANIZE TABLE statement" [*SQL Anywhere Server - SQL Reference*]
- "Rebuild indexes" on page 62
- "Reduce index fragmentation and skew" on page 219
- "Application profiling" on page 158
- "Choosing a diagnostic tracing level" on page 171
- "Advanced application profiling using diagnostic tracing" on page 169

Tutorial: Diagnosing table fragmentation

Use the lessons in this tutorial to learn how to use the **Application Profiling Wizard** to determine if your database has unacceptable levels of table fragmentation.

Table data is stored on database pages. When Data Modification Language (DML) statements such as INSERT, UPDATE, and DELETE are executed against a table, rows might not be stored contiguously, or might be split between multiple pages. Even though CPU activity is high, table fragmentation can negatively impact the performance of queries that require a scan of the table.

Lesson 1: Check for table fragmentation using the Application Profiling Wizard

Use this procedure to find and fix table fragmentation. It is recommended that you periodically check for fragmentation on your production database.

To find and fix table fragmentation

1. This tutorial assumes you have created the test database, *app_profiling.db*, required to use the application profiling tutorials. If you have not, see "Create a test database for the application profiling tutorials" on page 234.

2. Connect to *app_profiling.db* as follows:

   a. In Sybase Central, in the SQL Anywhere 12 plug-in, choose **Connections** » **Connect With SQL Anywhere 12**.

   b. In the **Connect** window, complete the following fields to connect to the test database, *app_profiling.db*, and then click **Connect**:

      - **User ID** **DBA**
      - **Password** **sql**
      - **Action** **Start A Database On This Computer**
      - **Database File** **C:\AppProfilingTutorial\app_profiling.db**
      - **Start Line** **dbeng12 -x tcpip**

3. In Sybase Central, choose **Mode** » **Application Profiling**.

   If the **Application Profiling Wizard** does not appear, choose **Application Profiling** » **Open Application Profiling Wizard**.

4. On the **Profiling Options** page, select **Overall Database Performance Based On The Database Schema**.

5. On the **Analysis File** page, save the analysis file in the appropriate directory. For example, *C:\AppProfilingTutorial*. If you are prompted to replace the file because it already exists, click **Yes**.

6. Click **Finish**.

   A list of recommendations appear in the **Application Profiling Details** pane.

7. If you see **Fragmented Tables**, double-click it. A **Recommendation** window appears containing a SQL statement you can run to resolve the table fragmentation.

8. You have completed the tutorial on using application profiling to identify and fix table fragmentation.

Note
You can also identify and fix table fragmentation using SQL. See "Identifying and fixing table fragmentation using SQL" on page 248.

See also

- "sa_table_fragmentation system procedure" [*SQL Anywhere Server - SQL Reference*]
- "REORGANIZE TABLE statement" [*SQL Anywhere Server - SQL Reference*]
- "Reduce table fragmentation" on page 215
- "Application profiling" on page 158
- "Choosing a diagnostic tracing level" on page 171
- "Advanced application profiling using diagnostic tracing" on page 169

Identifying and fixing table fragmentation using SQL

You can also identify and fix table fragmentation using SQL.

To check for table fragmentation

1. In the left pane of Sybase Central, click **app_profiling - DBA**, and then choose **File** » **Open Interactive SQL**.

   Interactive SQL starts and connects to the *app_profiling.db* database.

2. In Interactive SQL, run the following SQL statements to test for table fragmentation on the Employees table:

   ```
   CALL sa_table_fragmentation( 'Employees' );
   ```

   If the value in the segs_per_row (the number of segments per row) column is greater than 1.1, then table fragmentation is present. Higher degrees of fragmentation may negatively impact performance.

3. In Interactive SQL, run a REORGANIZE TABLE statement similar to the following to reduce table fragmentation:

   ```
   REORGANIZE TABLE Employees;
   ```

4. You have completed the tutorial on diagnosing table fragmentation.

See also

- "sa_table_fragmentation system procedure" [*SQL Anywhere Server - SQL Reference*]
- "REORGANIZE TABLE statement" [*SQL Anywhere Server - SQL Reference*]
- "Reduce table fragmentation" on page 215
- "Application profiling" on page 158
- "Choosing a diagnostic tracing level" on page 171
- "Advanced application profiling using diagnostic tracing" on page 169

Tutorial: Baselining with procedure profiling

Use the lessons in this tutorial to learn how to use the **Application Profiling Wizard** to create a baseline that you can use for comparison purposes when improving performance.

Procedure profiling provides execution time measurements for procedures, user-defined functions, events, system triggers, and triggers. You can set your saved results as a baseline and make incremental changes to a procedure and run it after each change you make. This allows you to compare the new results to the baseline.

Lesson 1: Create a baseline procedure

To create a baseline procedure

1. This tutorial assumes you have created the test database required for the application profiling tutorials. If you have not, see "Create a test database for the application profiling tutorials" on page 234.

2. Connect to *app_profiling.db* as follows:

   a. In Sybase Central, in the SQL Anywhere 12 plug-in, **choose Connections** » **Connect With the SQL Anywhere 12**.

   b. In the **Connect** window, complete the following fields to connect to the test database, *app_profiling.db*, and then click **Connect**:

   - **Authentication** **Database**
   - **User ID** **DBA**
   - **Password** **sql**
   - **Action** **Start A Database On This Computer**
   - **Database File** **C:\AppProfilingTutorial\app_profiling.db**
   - **Start Line** **dbeng12 -x tcpip**

3. In the left pane, click **app_profiling - DBA**, and then choose **File** » **Open Interactive SQL**.

   Interactive SQL starts and connects to the *app_profiling.db* database.

4. In Interactive SQL, run the following SQL statements:

   a. Create a table:

```
CREATE TABLE table1 (
Count INT );
```

   b. Create a baseline procedure:

```
CREATE PROCEDURE baseline( )
   BEGIN
      INSERT table1
         SELECT COUNT (*)
            FROM rowgenerator r1, rowgenerator r2,
                 rowgenerator r3
         WHERE r3.row_num < 5;
   END;
```

   c. Commit the changes you made to the database:

```
COMMIT;
```

5. Close Interactive SQL.

6. Click this link to continue the tutorial: "Lesson 2: Run an updated procedure against the baseline procedure" on page 250.

See also

- "Application profiling" on page 158
- "Analyze procedure profiling results" on page 162
- "Procedure profiling in Application Profiling mode" on page 160
- "Choosing a diagnostic tracing level" on page 171
- "Advanced application profiling using diagnostic tracing" on page 169

Lesson 2: Run an updated procedure against the baseline procedure

To run an updated procedure against the baseline procedure

1. In Sybase Central, choose **Mode** » **Application Profiling**.

   If the **Application Profiling Wizard** does not appear, choose **Application Profiling** » **Open Application Profiling Wizard**.

2. On the **Welcome** page, click **Next**.

3. On the **Profiling Options** page, select **Stored Procedure, Function, Trigger, Or Event Execution Time**.

4. Click **Finish**.

   The database server begins procedure profiling.

5. In the left pane of Sybase Central, double-click **Procedures & Functions**.

6. Right-click the baseline procedure and choose **Execute From Interactive SQL**. Procedure profiling is enabled, so execution details for the procedure are captured.

7. Close Interactive SQL.

8. View the profiling results.

   a. In the left pane of Sybase Central, select the baseline procedure.

   b. Click the **Profiling Results** tab in the right pane. If no results appear, choose **View** » **Refresh Folder**.

      The execution times appear for each line in the baseline procedure.

9. Save the profiling results.

   a. Right-click the database and choose **Properties**.

   b. Click the **Profiling Settings** tab.

c. Select **Save The Profiling Information Currently In The Database To The Following Profiling Log File**, and then specify a location and file name for the profiling log file.

d. Click **Apply**. Do not close the properties window.

The procedure profiling information that was just gathered is saved to the specified profiling log file (*.plg*).

10. Enable baselining against the profiling log file.

a. On the **Profiling Settings** tab of the **App_Profiling - DBA Database Properties** window, select **Use The Profiling Information In The Following Profiling Log File As A Baseline For Comparison**.

b. Browse to and select the profiling log file you created.

c. Click **Apply**.

d. Click **OK** to close the **App_Profiling - DBA Database Properties** window.

11. Make changes to the baseline procedure.

a. In Sybase Central, choose **Mode** » **Design**.

b. In the left pane, browse to and select the baseline procedure in the **Procedures & Functions**.

c. On the **SQL** tab in the right pane, delete the existing INSERT statement.

d. Copy and paste the following SQL statement into the procedure:

```
INSERT table1
   SELECT COUNT ( * ) FROM rowgenerator r1, rowgenerator r2,
rowgenerator r3
   WHERE r3.row_num < 250;
```

e. Choose **File** » **Save**.

12. In **Procedures & Functions**, right-click the baseline procedure and choose **Execute From Interactive SQL**.

13. Exit Interactive SQL when the procedure completes.

14. Click this link to continue the tutorial: "Lesson 3: Compare the procedure profiling results" on page 252.

See also

- "Application profiling" on page 158
- "Analyze procedure profiling results" on page 162
- "Procedure profiling in Application Profiling mode" on page 160
- "Choosing a diagnostic tracing level" on page 171
- "Advanced application profiling using diagnostic tracing" on page 169

Lesson 3: Compare the procedure profiling results

To compare the procedure profiling results

1. In Sybase Central, choose **Mode** » **Application Profiling**.

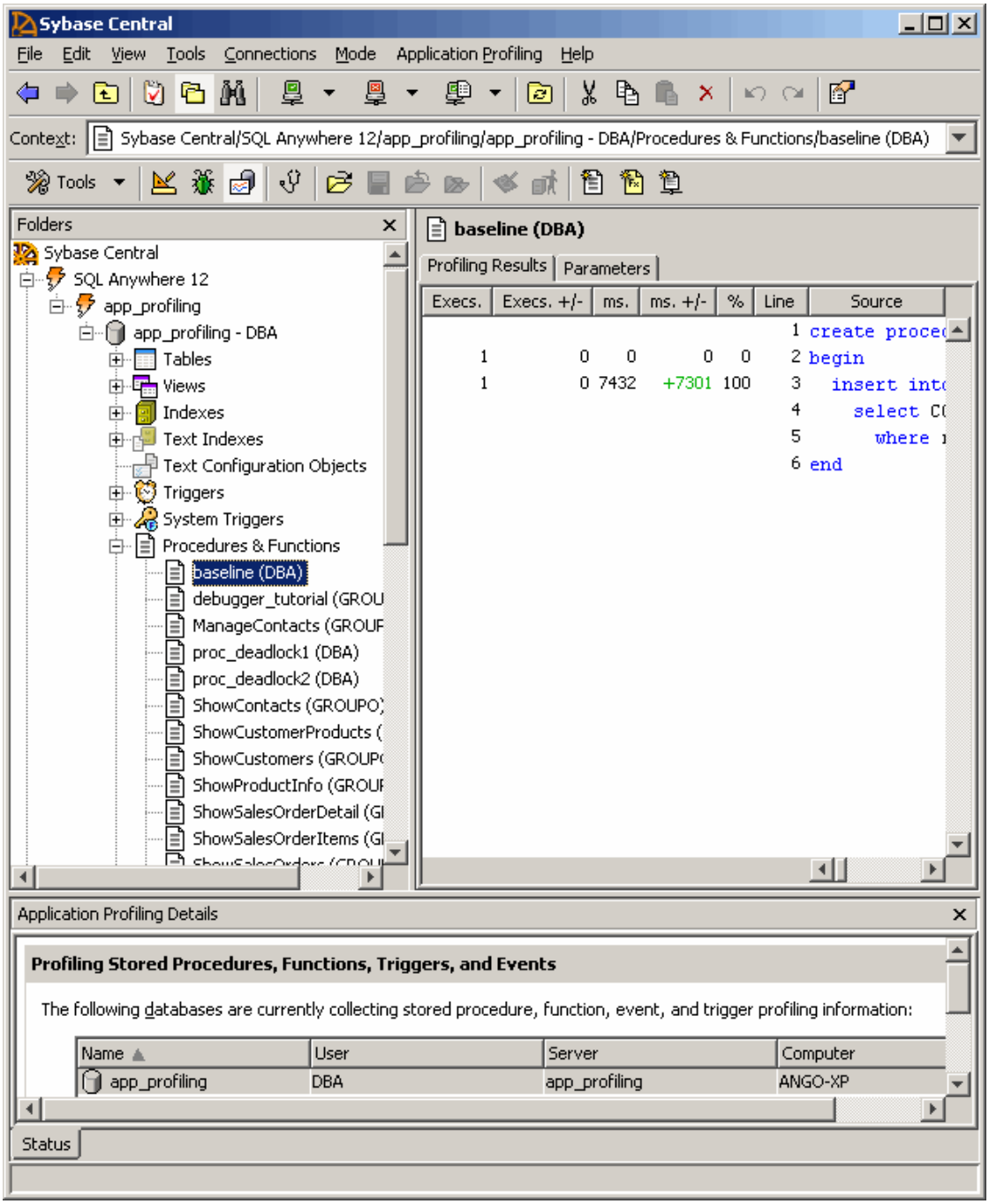
   If the **Application Profiling Wizard** appears, click **Cancel**.

2. In the left pane of Sybase Central, in **Procedures & Functions**, click the baseline procedure.

3. In the right pane, click the **Profiling Results** tab.

4. Choose **View** » **Refresh Folder**.

   Two new columns, **Execs. +/-** and **ms. +/-**, appear.



The **Execs.** +/- and **ms.** +/- columns result from comparing statistics in the profiling log file to the statistics captured during the most recent execution of the procedure. Specifically, they compare number of executions and duration of execution, respectively, for each line of code in the procedure.

Typically, you are interested in the **ms. +/-** column, which indicates whether you improved the execution time for lines of code in the procedure. Faster times are indicated by a minus sign and red font. Slower times are indicated by no sign, and green font. In this tutorial, the value in the **ms. +/-** column should be a + sign along with an execution time in green font. The INSERT statement in the updated procedure has a slower time than the INSERT statement in the baseline procedure.

5. You have completed the tutorial on baselining with procedure profiling.

See also

- "Application profiling" on page 158
- "Analyze procedure profiling results" on page 162
- "Procedure profiling in Application Profiling mode" on page 160
- "Choosing a diagnostic tracing level" on page 171
- "Advanced application profiling using diagnostic tracing" on page 169

Querying and modifying data

This section describes how to query and modify data, including how to use joins. It includes several chapters on queries, from simple to complex, and information about inserting, deleting, and updating data. This chapter also includes an in-depth look at how to create analytical queries that return multidimensional results.

Querying data

A query requests data from the database and receives the results. This process is also known as data retrieval. All SQL queries are expressed using the SELECT statement. You use the SELECT statement to retrieve all, or a subset of, the rows in one or more tables, and to retrieve all, or a subset of, the columns in one or more tables.

Querying and the SELECT statement

The SELECT statement retrieves information from a database for use by the client application. SELECT statements are also called **queries**. The information is delivered to the client application in the form of a result set. The client can then process the result set. For example, Interactive SQL displays the result set in the Results pane. Result sets consist of a set of rows, just like tables in the database.

SELECT statements contain **clauses**, which are commands that define the scope of the results to return. In the following SELECT syntax, each new line is a separate clause. Only the more common clauses are listed here.

SELECT *select-list*
[**FROM** *table-expression*]
[**WHERE** *search-condition*]
[**GROUP BY** *column-name*]
[**HAVING** *search-condition*]
[**ORDER BY** { *expression* | *integer* }]

The clauses in the SELECT statement are as follows:

- The SELECT clause specifies the columns you want to retrieve. It is the only required clause in the SELECT statement.

- The FROM clause specifies the tables from which columns are pulled. It is required in all queries that retrieve data from tables. SELECT statements without FROM clauses have a different meaning, and this section does not discuss them.

  Although most queries operate on tables, queries may also retrieve data from other objects that have columns and rows, including views, other queries (derived tables) and stored procedure result sets. See "FROM clause" [*SQL Anywhere Server - SQL Reference*].

- The WHERE clause specifies the rows in the tables you want to see.
- The GROUP BY clause allows you to aggregate data.
- The HAVING clause specifies rows on which aggregate data is to be collected.
- The ORDER BY clause sorts the rows in the result set. (By default, rows are returned from relational databases in an order that has no meaning.)

  For information about GROUP BY, HAVING, and ORDER BY clauses, see "Summarizing, grouping, and sorting query results" on page 366.

Most of the clauses are optional, but if they are included then they must appear in the correct order.

See "SELECT statement" [*SQL Anywhere Server - SQL Reference*].

SQL queries

Throughout the documentation, SELECT statements and other SQL statements appear with each clause on a separate row, and with the SQL keywords in uppercase. This is done to make the statements easier to read but is not a requirement. You can enter SQL keywords in any case, and you can have line breaks anywhere in the statement.

Keywords and line breaks

For example, the following SELECT statement finds the first and last names of contacts living in California from the Contacts table.

```
SELECT GivenName, Surname
FROM Contacts
WHERE State = 'CA';
```

It is equally valid, though not as readable, to enter the statement as follows:

```
SELECT GivenName,
Surname from Contacts
WHERE State
 = 'CA';
```

Case sensitivity of strings and identifiers

Identifiers such as table names, column names, and so on, are case insensitive in SQL Anywhere databases.

Strings are case insensitive by default, so that 'CA', 'ca', 'cA', and 'Ca' are equivalent, but if you create a database as case sensitive then the case of strings is significant. The SQL Anywhere sample database is case insensitive.

See also

- "Creating a SQL Anywhere database" [*SQL Anywhere Server - Database Administration*]
- "Initialization utility (dbinit)" [*SQL Anywhere Server - Database Administration*]
- "Case sensitivity" on page 648

Qualifying identifiers

You can qualify the names of database identifiers if there is ambiguity about which object is being referred to. For example, the SQL Anywhere sample database contains several tables with a column called City, so you may have to qualify references to City with the name of the table. In a larger database you may also have to use the name of the owner of the table to identify the table.

```
SELECT Contacts.City
FROM Contacts
WHERE State = 'CA';
```

Since the examples in this section involve single-table queries, column names in syntax models and examples are usually not qualified with the names of the tables or owners to which they belong.

These elements are left out for readability; it is never wrong to include qualifiers.

Row order in the result set

Row order in the result set is insignificant. There is no guarantee of the order in which rows are returned from the database, and no meaning to the order. If you want to retrieve rows in a particular order, you must specify the order in the query.

The select list: Specifying columns

The select list comprises one or more objects from which to query data. The select list commonly consists of a series of column names separated by commas, or an asterisk as shorthand to represent all columns. More generally, the select list can include one or more expressions, separated by commas. There is no comma after the last column in the list, or if there is only one column in the list.

The general syntax for the select list looks like this:

SELECT *expression* [, *expression*]...

If any table or column name in the list does not conform to the rules for valid identifiers, you must enclose the identifier in double quotes.

The select list expressions can include * (all columns), a list of column names, character strings, column headings, and expressions including arithmetic operators. You can also include aggregate functions, which are discussed in “Summarizing, grouping, and sorting query results” on page 366.

For more information about expressions, see “Expressions” [*SQL Anywhere Server - SQL Reference*].

Selecting all columns from a table

The asterisk (*) has a special meaning in SELECT statements. It represents all the column names in all the tables specified in the FROM clause. You can use it to save entering time and errors when you want to see all the columns in a table.

When you use SELECT *, the columns are returned in the order in which they were defined when the table was created.

The syntax for selecting all the columns in a table is:

```
SELECT *
FROM table-expression;
```

SELECT * finds all the columns currently in a table, so that changes in the structure of a table such as adding, removing, or renaming columns automatically modify the results of SELECT *. Listing the columns individually gives you more precise control over the results.

Example

The following statement retrieves all columns in the Departments table. No WHERE clause is included; therefore, this statement retrieves every row in the table:

```
SELECT *
FROM Departments;
```

The results look like this:

DepartmentID	DepartmentName	DepartmentHeadID
100	R & D	501
200	Sales	902
300	Finance	1293
400	Marketing	1576
...	...	...

You get exactly the same results by listing all the column names in the table in order after the SELECT keyword:

```
SELECT DepartmentID, DepartmentName, DepartmentHeadID
FROM Departments;
```

Like a column name, "*" can be qualified with a table name, as in the following query:

```
SELECT Departments.*
FROM Departments;
```

Selecting specific columns from a table

You can limit the columns that a SELECT statement retrieves by listing the column(s) immediately after the SELECT keyword. This SELECT statement has the following syntax:

SELECT *column-name* [, *column-name*]...
FROM *table-name*

In the syntax, *column-name* and *table-name* should be replaced with the names of the columns and table you are querying.

For example:

```
SELECT Surname, GivenName
FROM Employees;
```

Projections and restrictions

A **projection** is a subset of the columns in a table. A **restriction** (also called **selection**) is a subset of the rows in a table, based on some conditions.

For example, the following SELECT statement retrieves the names and prices of all products in the SQL Anywhere sample database that cost more than $15:

```
SELECT Name, UnitPrice
FROM Products
WHERE UnitPrice > 15;
```

This query uses both a projection (`SELECT Name, UnitPrice`) and a restriction (`WHERE UnitPrice > 15`).

Rearranging the order of columns

The order in which you list column names determines the order in which the columns are displayed. The two following examples show how to specify column order in a display. Both of them find and display the department names and identification numbers from all five of the rows in the Departments table, but in a different order.

```
SELECT DepartmentID, DepartmentName
FROM Departments;
```

DepartmentID	DepartmentName
100	R & D
200	Sales
300	Finance
400	Marketing
...	...

```
SELECT DepartmentName, DepartmentID
FROM Departments;
```

DepartmentName	DepartmentID
R & D	100

DepartmentName	DepartmentID
Sales	200
Finance	300
Marketing	400
...	...

Joins

A join links the rows in two or more tables by comparing the values in columns of each table. For example, you might want to select the order item identification numbers and product names for all order items that shipped more than a dozen pieces of merchandise:

```
SELECT SalesOrderItems.ID, Products.Name
FROM Products JOIN SalesOrderItems
WHERE SalesOrderItems.Quantity > 12;
```

The Products table and the SalesOrderItems table are joined together based on the foreign key relationship between them.

See “Joins: Retrieving data from several tables” on page 386.

Renaming columns in query results

By default, the heading for each column of a result set is the name of the expression supplied in the select list. For expressions that are column values, the heading will be the column name. In embedded SQL, one can use the DESCRIBE statement to determine the name of each expression returned by a cursor. Other application interfaces also support querying the names of each result set column through interface-specific mechanisms. The sa_describe_query system procedure offers an interface-independent means to determine the names of the result set columns for an arbitrary SQL query.

You can override the name of any expression in a query's SELECT list by using an **alias**, as follows:

SELECT *column-name* [**AS**] *alias*

Providing an alias can produce more readable results. For example, you can change DepartmentName to Department in a listing of departments as follows:

```
SELECT DepartmentName AS Department,
   DepartmentID AS "Identifying Number"
FROM Departments;
```

Department	Identifying Number
R & D	100

Department	Identifying Number
Sales	200
Finance	300
Marketing	400
...	...

Usage

- **Using spaces and keywords in an alias** In the example above, the "Identifying Number" alias for DepartmentID is enclosed in double quotes because it contains a blank. You also use double quotes if you want to use keywords or special characters in aliases. For example, the following query is invalid without the quotation marks:

  ```
  SELECT DepartmentName AS Department,
     DepartmentID AS "integer"
  FROM Departments;
  ```

- **Name space occlusion** Aliases can be used anywhere in the SELECT block in which they are defined, including other SELECT list expressions that in turn define additional aliases. Cyclic alias references are not permitted. If the alias specified for an expression is identical to the name of a column or variable in the name space of the SELECT block, the alias definition occludes the column or variable. For example:

  ```
  SELECT DepartmentID AS DepartmentName
  FROM Departments
  WHERE DepartmentName = 'Marketing'
  ```

  will return an error, "cannot convert 'Marketing' to a numeric". This is because the equality predicate in the query's WHERE clause is attempting to compare the string literal "Marketing" to the integer column DepartmentID, and the data types are incompatible.

  Note
  When referencing column names you can explicitly qualify the column name by its table name, for example Departments.DepartmentID, to disambiguate a naming conflict with an alias.

- **Transact-SQL compatibility** Adaptive Server Enterprise supports *both* the SQL/2008 AS keyword, and the use of an equals sign, to identify an alias for a SELECT list item.

  See also "Writing compatible queries" on page 653.

See also

- "sa_describe_query system procedure" [*SQL Anywhere Server - SQL Reference*]
- "DESCRIBE statement [ESQL]" [*SQL Anywhere Server - SQL Reference*]

Character strings in query results

Most SELECT statements produce results that consist solely of data from the tables in the FROM clause. However, strings of characters can also be displayed in query results by enclosing them in single quotation marks and separating them from other elements in the select list with commas. To enclose a quotation mark in a string, you precede it with another quotation mark. For example:

```
SELECT 'The department''s name is' AS "Prefix",
   DepartmentName AS Department
FROM Departments;
```

Prefix	Department
The department's name is	R & D
The department's name is	Sales
The department's name is	Finance
The department's name is	Marketing
The department's name is	Shipping

Computing values in the SELECT list

The expressions in the select list can be more complicated than just column names or strings. For example, you can perform computations with data from numeric columns in a select list.

Arithmetic operations

To illustrate the numeric operations you can perform in the select list, you start with a listing of the names, quantity in stock, and unit price of products in the SQL Anywhere sample database.

```
SELECT Name, Quantity, UnitPrice
FROM Products;
```

Name	Quantity	UnitPrice
Tee Shirt	28	9
Tee Shirt	54	14
Tee Shirt	75	14
Baseball Cap	112	9
...	...	...

Suppose the practice is to replenish the stock of a product when there are ten items left in stock. The following query lists the number of each product that must be sold before re-ordering:

```
SELECT Name, Quantity - 10
   AS "Sell before reorder"
FROM Products;
```

Name	Sell before reorder
Tee Shirt	18
Tee Shirt	44
Tee Shirt	65
Baseball Cap	102
...	...

You can also combine the values in columns. The following query lists the total value of each product in stock:

```
SELECT Name, Quantity * UnitPrice AS "Inventory value"
FROM Products;
```

Name	Inventory value
Tee Shirt	252.00
Tee Shirt	756.00
Tee Shirt	1050.00
Baseball Cap	1008.00
...	...

When there is more than one arithmetic operator in an expression, multiplication, division, and modulo are calculated first, followed by subtraction and addition. When all arithmetic operators in an expression have the same level of precedence, the order of execution is left to right. Expressions within parentheses take precedence over all other operations.

For example, the following SELECT statement calculates the total value of each product in inventory, and then subtracts five dollars from that value.

```
SELECT Name, Quantity * UnitPrice - 5
FROM Products;
```

To ensure correct results, use parentheses where possible. The following query has the same meaning and gives the same results as the previous one, but the syntax is more precise:

```
SELECT Name, ( Quantity * UnitPrice ) - 5
FROM Products;
```

Arithmetic operations may overflow because the result of the operation can not be represented in the data type. When an overflow occurs, an error is returned instead of a value.

See also "Operator precedence" [*SQL Anywhere Server - SQL Reference*].

String operations

You can concatenate strings using a string concatenation operator. You can use either || (defined by SQL/2008) or + (supported by Adaptive Server Enterprise) as the concatenation operator. For example, the following statement retrieves and concatenates GivenName and Surname values in the results:

```
SELECT EmployeeID, GivenName || ' ' || Surname AS Name
FROM Employees;
```

EmployeeID	Name
102	Fran Whitney
105	Matthew Cobb
129	Philip Chin
148	Julie Jordan
...	...

Date and time operations

Although you can use operators on date and time columns, this typically involves the use of functions. See "SQL functions" [*SQL Anywhere Server - SQL Reference*].

Additional notes on calculated columns

- **Columns can be given an alias** By default the column name is the expression listed in the select list, but for calculated columns the expression is cumbersome and not very informative. See "Renaming columns in query results" on page 260.

- **Other operators are available** The multiplication operator can be used to combine columns. You can use other operators, including the standard arithmetic operators, and logical operators and string operators.

  For example, the following query lists the full names of all customers:

  ```
  SELECT ID, (GivenName || ' ' || Surname ) AS "Full name"
  FROM Customers;
  ```

  The || operator concatenates strings. In this query, the alias for the column has spaces, and so must be surrounded by double quotes. This rule applies not only to column aliases, but to table names and other identifiers in the database. See "Operators" [*SQL Anywhere Server - SQL Reference*].

- **Functions can be used** In addition to combining columns, you can use a wide range of built-in functions to produce the results you want.

  For example, the following query lists the product names in uppercase:

```
SELECT ID, UCASE( Name )
FROM Products;
```

ID	UCASE(Products.name)
300	TEE SHIRT
301	TEE SHIRT
302	TEE SHIRT
400	BASEBALL CAP
...	...

  See "SQL functions" [*SQL Anywhere Server - SQL Reference*].

Eliminating duplicate query results

The optional DISTINCT keyword eliminates duplicate rows from the results of a SELECT statement. If you do not specify DISTINCT, you get all rows, including duplicates. Optionally, you can specify ALL before the select list to get all rows. For compatibility with other implementations of SQL, SQL Anywhere syntax allows the use of ALL to explicitly ask for all rows. ALL is the default.

For example, if you search for all the cities in the Contacts table without DISTINCT, you get 60 rows:

```
SELECT City
FROM Contacts;
```

You can eliminate the duplicate entries using DISTINCT. The following query returns only 16 rows:

```
SELECT DISTINCT City
FROM Contacts;
```

NULL values are not distinct

The DISTINCT keyword treats NULL values as duplicates of each other. In other words, when DISTINCT is included in a SELECT statement, only one NULL is returned in the results, no matter how many NULL values are encountered. See "Elimination of unnecessary DISTINCT conditions" on page 529.

The FROM clause: Specifying tables

The FROM clause is required in every SELECT statement involving data from tables, views, or stored procedures.

The FROM clause can include JOIN conditions linking two or more tables, and can include joins to other queries (derived tables). For information about these features, see "Joins: Retrieving data from several tables" on page 386.

Qualifying table names

In the FROM clause, the full naming syntax for tables and views is always permitted, such as:

```
SELECT select-list
FROM owner.table-name;
```

Qualifying table, view, and procedure names is necessary only when the object is owned by a user ID that is different from the user ID of the current connection, or if the user ID of the owner is not the name of a group to which the user ID of the current connection belongs.

Using correlation names

You can give a table name a correlation name to improve readability, and to save entering the full table name each place it is referenced. You assign the correlation name in the FROM clause by entering it after the table name, like this:

```
SELECT d.DepartmentID, d.DepartmentName
FROM Departments d;
```

When a correlation name is used, all other references to the table, for example in a WHERE clause, *must* use the correlation name, rather than the table name. Correlation names must conform to the rules for valid identifiers.

See "FROM clause" [*SQL Anywhere Server - SQL Reference*].

Querying derived tables

A derived table is a table derived directly, or indirectly, from one or more tables by the evaluation of a query expression. Derived tables are defined in the FROM clause of a SELECT statement.

Querying a derived table works the same as querying a view. That is, the values of a derived table are determined at the time the derived table definition is evaluated. Derived tables differ from views, however, in that the definition for a derived table is not stored in the database. Derived tables differ from base and temporary tables in that they are not materialized and they cannot be referred to from outside the query in which they are defined.

The following query uses a derived table (my_derived_table) to hold the maximum salary in each department. The data in the derived table is then joined to the Employees table to get the surnames of the employee earning the salaries.

```
SELECT Surname,
   my_derived_table.maximum_salary AS Salary,
   my_derived_table.DepartmentID
FROM Employees e,
  ( SELECT MAX( Salary ) AS maximum_salary, DepartmentID
      FROM Employees
```



```
      GROUP BY DepartmentID ) my_derived_table
  WHERE e.Salary = my_derived_table.maximum_salary
  AND e.DepartmentID = my_derived_table.DepartmentID
ORDER BY Salary DESC;
```

Surname	Salary	DepartmentID
Shea	138948.00	300
Scott	96300.00	100
Kelly	87500.00	200
Evans	68940.00	400
Martinez	55500.80	500

The following example creates a derived table (MyDerivedTable) that ranks the items in the Products table, and then queries the derived table to return the three least expensive items:

```
SELECT TOP 3 *
       FROM ( SELECT Description,
                     Quantity,
                     UnitPrice,
                     RANK() OVER ( ORDER BY UnitPrice ASC )
                  AS Rank
                  FROM Products ) AS MyDerivedTable
ORDER BY Rank;
```

See also: "FROM clause" [*SQL Anywhere Server - SQL Reference*].

Querying objects other than tables

The most common elements in a FROM clause are table names. However, it is also possible to query rows from other database objects that have a table-like structure—that is, a well-defined set of rows and columns. For example, you can query views, or query stored procedures that return result sets.

For example, the following statement queries the result set of a stored procedure called ShowCustomerProducts.

```
SELECT *
FROM ShowCustomerProducts( 149 );
```

Executing a SELECT over a DML statement

SQL Anywhere supports the use of a DML statement (INSERT, UPDATE, DELETE, or MERGE) as a table expression in a query's FROM clause.

When you include a *dml-derived-table* in a statement, it is ignored during the DESCRIBE. At OPEN time, the UPDATE statement is executed first, and the results are stored in a temporary table. The temporary table uses the column names of the table that is being modified by the statement. You can refer to the modified values by using the correlation name from the REFERENCING clause. By specifying OLD or

FINAL, you do not need a set of unique column names for the updated table that is referenced in the query. The *dml-derived-table* statement can only reference one updatable table; updates over multiple tables return an error.

For example, the following query uses a SELECT over an UPDATE statement to perform the operations listed below:

- Updates all products in the sample database with a 7% price increase
- Lists the affected products and their orders that were shipped between April 10, 2000 and May 21, 2000 whose order quantity was greater than 36

```
SELECT old_products.ID, old_products.name, old_products.UnitPrice AS
OldPrice,
       final_products.UnitPrice AS NewPrice, SOI.ID AS OrderID, SOI.Quantity
FROM
( UPDATE Products SET UnitPrice = UnitPrice * 1.07 )
     REFERENCING ( OLD AS old_products FINAL AS final_products )
  JOIN SalesOrderItems AS SOI ON SOI.ProductID = old_products.ID
WHERE SOI.ShipDate BETWEEN '2000-04-10' AND '2000-05-21'
      AND SOI.QUANTITY > 36
ORDER BY old_prodcuts.ID;
```

The following query uses both a MERGE statement and an UPDATE statement. The modified_employees table represents a collection of employees whose state has been altered, while the MERGE statement merges employee identifiers and names for those employees whose salary has been increased by 3% with employees who are included in the modified_employees table. In this query, the option settings that are specified in the OPTION clause apply to both the UPDATE and MERGE statements.

```
CREATE TABLE modified_employees
( EmployeeID INTEGER PRIMARY KEY, Surname VARCHAR(40), GivenName
VARCHAR(40) );


MERGE INTO modified_employees AS me
USING (SELECT modified_employees.EmployeeID,
              modified_employees.Surname,
              modified_employees.GivenName
       FROM (
          UPDATE Employees
          SET Salary = Salary * 1.03
          WHERE ManagerID = 501)
            REFERENCING (FINAL AS modified_employees) ) AS dt_e
       ON dt_e.EmployeeID = me.EmployeeID
WHEN MATCHED THEN SKIP
WHEN NOT MATCHED THEN INSERT
OPTION( optimization_level=1, isolation_level=2 );
```

Using multiple tables within a query

When you use multiple *dml-derived-table* arguments within a query, the order of execution of the UPDATE statement is not guaranteed. The following statement updates both the Products and SalesOrderItems tables in the sample database, and then produces a result based on a join that includes these modifications:

```
SELECT old_products.ID, old_products.name, old_products.UnitPrice AS
OldPrice,
       final_products.UnitPrice AS NewPrice,
       SalesOrders.ID AS OrderID, SalesOrders.CustomerID,
```



```
       old_order_items.Quantity,
       old_order_items.ShipDate AS OldShipDate,
       final_order_items.ShipDate AS RevisedShipDate
FROM
( UPDATE Products SET UnitPrice = UnitPrice * 1.07 )
      REFERENCING ( OLD AS old_products FINAL AS final_products )
  JOIN
( UPDATE SalesOrderItems
  SET ShipDate = DATEADD( DAY, 6, ShipDate )
  WHERE ShipDate BETWEEN  '2000-04-10' AND '2000-05-21' )
      REFERENCING ( OLD AS old_order_items FINAL AS final_order_items )
        ON (old_order_items.ProductID = old_products.ID)
  JOIN SalesOrders ON ( SalesOrders.ID = old_order_items.ID )
WHERE old_order_items.Quantity > 36
ORDER BY old_products.ID;
```

Using tables without materializing results

You can also embed an UPDATE statement without materializing its result by using the REFERENCING (NONE) clause. Because the result of the UPDATE statement is empty in this case, you must write your query to ensure that the query returns the intended result. You can ensure that a non-empty result is returned by placing the *dml-derived-table* in the null-supplying side of an outer join. For example:

```
SELECT 'completed' AS finished, ( SELECT COUNT( * ) FROM Products ) AS
product_total
FROM SYS.DUMMY LEFT OUTER JOIN
    ( UPDATE Products SET UnitPrice = UnitPrice * 1.07 )
      REFERENCING ( NONE ) ON 1=1;
```

You can also ensure that a non-empty result is returned by using the *dml-derived-table* as part of a query expression using one of the set operators (UNION, EXCEPT, or INTERSECT). For example:

```
SELECT 'completed' AS finished, ( SELECT COUNT( * ) FROM Products ) AS
product_total
FROM SYS.DUMMY
UNION ALL
SELECT 'dummy', 1 /* This query specification returns the empty set */
FROM ( UPDATE Products SET UnitPrice = UnitPrice * 1.07 )
      REFERENCING ( NONE );
```

See also

- "FROM clause" [*SQL Anywhere Server - SQL Reference*]
- "Data modification statements" on page 513

The WHERE clause: Specifying rows

The WHERE clause in a SELECT statement specifies the search conditions for exactly which rows are retrieved. Search conditions are also referred to as **predicates**. The general format is:

SELECT *select-list*
FROM *table-list*
WHERE *search-condition*

Search conditions in the WHERE clause include the following:

- **Comparison operators** (=, <, >, and so on) For example, you can list all employees earning more than $50,000:

```
SELECT Surname
   FROM Employees
   WHERE Salary > 50000;
```

- **Ranges** (BETWEEN and NOT BETWEEN) For example, you can list all employees earning between $40,000 and $60,000:

```
SELECT Surname
   FROM Employees
   WHERE Salary BETWEEN 40000 AND 60000;
```

- **Lists** (IN, NOT IN) For example, you can list all customers in Ontario, Quebec, or Manitoba:

```
SELECT CompanyName, State
   FROM Customers
   WHERE State IN( 'ON', 'PQ', 'MB');
```

- **Character matches** (LIKE and NOT LIKE) For example, you can list all customers whose phone numbers start with 415. (The phone number is stored as a string in the database):

```
SELECT CompanyName, Phone
   FROM Customers
   WHERE Phone LIKE '415%';
```

- **Unknown values** (IS NULL and IS NOT NULL) For example, you can list all departments with managers:

```
SELECT DepartmentName
   FROM Departments
   WHERE DepartmentHeadID IS NOT NULL;
```

- **Combinations** (AND, OR) For example, you can list all employees earning over $50,000 whose first name begins with the letter A.

```
SELECT GivenName, Surname
   FROM Employees
   WHERE Salary > 50000
   AND GivenName like 'A%';
```

For the full syntax of search conditions, see "Search conditions" [*SQL Anywhere Server - SQL Reference*].

Using comparison operators in the WHERE clause

You can use comparison operators in the WHERE clause. The operators follow the syntax:

WHERE *expression comparison-operator expression*

See "Comparison operators" [*SQL Anywhere Server - SQL Reference*], and "Expressions" [*SQL Anywhere Server - SQL Reference*].

Notes on comparisons

- **Sort orders** In comparing character data, < means earlier in the sort order and > means later in the sort order. The sort order is determined by the collation chosen when the database is created. You can find out the collation by running the dbinfo utility against the database:

  ```
  dbinfo -c "uid=DBA;pwd=sql"
  ```

  You can also find the collation from Sybase Central by going to the Extended Information tab of the **Database Properties** window.

- **Trailing blanks** When you create a database, you indicate whether trailing blanks are ignored for comparison purposes.

  By default, databases are created with trailing blanks not ignored. For example, 'Dirk' is not the same as 'Dirk '. You can create databases with blank padding, so that trailing blanks are ignored.

- **Comparing dates** In comparing dates, < means earlier and > means later.

- **Case sensitivity** When you create a database, you indicate whether string comparisons are case sensitive or not.

  By default, databases are created case insensitive. For example, 'Dirk' is the same as 'DIRK'. You can create databases to be case sensitive.

Here are some SELECT statements using comparison operators:

```
SELECT *
   FROM Products
   WHERE Quantity < 20;
SELECT E.Surname, E.GivenName
   FROM Employees E
   WHERE Surname > 'McBadden';
SELECT ID, Phone
   FROM Contacts
   WHERE State  != 'CA';
```

The NOT operator

The NOT operator negates an expression. Either of the following two queries find all Tee shirts and baseball caps that cost $10 or less. However, note the difference in position between the negative logical operator (NOT) and the negative comparison operator (!>).

```
SELECT ID, Name, Quantity
   FROM Products
   WHERE (name = 'Tee Shirt' OR name = 'BaseBall Cap')
   AND NOT UnitPrice > 10;
SELECT ID, Name, Quantity
   FROM Products
   WHERE (name = 'Tee Shirt' OR name = 'BaseBall Cap')
   AND UnitPrice !> 10;
```

Using ranges in the WHERE clause

The BETWEEN keyword specifies an inclusive range, in which the lower value and the upper value and the values they bracket are searched for.

To list all the products with prices between $10 and $15, inclusive

- Enter the following query:

```
SELECT Name, UnitPrice
   FROM Products
   WHERE UnitPrice BETWEEN 10 AND 15;
```

Name	UnitPrice
Tee Shirt	14
Tee Shirt	14
Baseball Cap	10
Shorts	15

You can use NOT BETWEEN to find all the rows that are not inside the range.

To list all the products cheaper than $10 or more expensive than $15

- Execute the following query:

```
SELECT Name, UnitPrice
   FROM Products
   WHERE UnitPrice NOT BETWEEN 10 AND 15;
```

Name	UnitPrice
Tee Shirt	9
Baseball Cap	9
Visor	7
Visor	7
...	...

Using lists in the WHERE clause

The IN keyword allows you to select values that match any one of a list of values. The expression can be a constant or a column name, and the list can be a set of constants or, more commonly, a subquery.

For example, without IN, if you want a list of the names and states of all the customers who live in Ontario, Manitoba, or Quebec, you can enter this query:


```
SELECT CompanyName, State
   FROM Customers
   WHERE State = 'ON' OR State = 'MB' OR State = 'PQ';
```

However, you get the same results if you use IN. The items following the IN keyword must be separated by commas and enclosed in parentheses. Put single quotes around character, date, or time values. For example:

```
SELECT CompanyName, State
   FROM Customers
   WHERE State IN( 'ON', 'MB', 'PQ');
```

Perhaps the most important use for the IN keyword is in nested queries, also called subqueries.

Matching character strings in the WHERE clause

Pattern matching is a versatile way of identifying character data. In SQL, the LIKE keyword is used to search for patterns. Pattern matching employs wildcard characters to match different combinations of characters.

The LIKE keyword indicates that the following character string is a matching pattern. LIKE is used with character data.

The syntax for LIKE is:

expression [**NOT**] **LIKE** *match-expression*

The expression to be matched is compared to a match-expression that can include these special symbols:

Symbols	Meaning
%	Matches any string of 0 or more characters
_	Matches any one character
[specifier]	The specifier in the brackets may take the following forms: • **Range** A range is of the form *rangespec1-rangespec2*, where *rangespec1* indicates the start of a range of characters, the hyphen indicates a range, and *rangespec2* indicates the end of a range of characters. • **Set** A set can include any discrete set of values, in any order. For example, [a2bR]. Note that the range [a-f], and the sets [abcdef] and [fcbdae] return the same set of values.
[^specifier]	The caret symbol (^) preceding a specifier indicates non-inclusion. [^a-f] means not in the range a-f; [^a2bR] means not a, 2, b, or R.

You can match the column data to constants, variables, or other columns that contain the wildcard characters displayed in the table. When using constants, you should enclose the match strings and character strings in single quotes.

Examples

All the following examples use LIKE with the Surname column in the Contacts table. Queries are of the form:

```
SELECT Surname
   FROM Contacts
   WHERE Surname LIKE match-expression;
```

The first example would be entered as

```
SELECT Surname
   FROM Contacts
   WHERE Surname LIKE 'Mc%';
```

Match expression	Description	Returns
'Mc%'	Search for every name that begins with the letters Mc	McEvoy
'%er'	Search for every name that ends with er	Brier, Miller, Weaver, Rayner
'%en%'	Search for every name containing the letters en.	Pettengill, Lencki, Cohen
'_ish'	Search for every four-letter name ending in ish.	Fish
'Br[iy][ae]r'	Search for Brier, Bryer, Briar, or Bryar.	Brier
'[M-Z]owell'	Search for all names ending with owell that begin with a single letter in the range M to Z.	Powell
'M[^c]%'	Search for all names beginning with M' that do not have c as the second letter	Moore, Mulley, Miller, Masalsky

Wildcards require LIKE

Wildcard characters used without LIKE are interpreted as **string literals** rather than as a pattern: they represent exactly their own values. The following query attempts to find any phone numbers that consist of the four characters 415% only. It does not find phone numbers that start with 415.

```
SELECT Phone
   FROM Contacts
   WHERE Phone = '415%';
```


See also "String literals" [*SQL Anywhere Server - SQL Reference*].

Using LIKE with date and time values

You can use LIKE on DATE, TIME, TIMESTAMP, and TIMESTAMP WITH TIME ZONE fields. However, the LIKE predicate only works on character data. When you use LIKE with date and time values, the values are implicitly CAST to CHAR or VARCHAR using the corresponding option setting for DATE, TIME, TIMESTAMP, and TIMESTAMP WITH TIME ZONE data types to format the value:

Date/time type	CAST to VARCHAR using
DATE	date_format
TIME	time_format
TIMESTAMP	timestamp_format
TIMESTAMP WITH TIME ZONE	timestamp_with_time_zone_format

A consequence of using LIKE when searching for DATE, TIME or TIMESTAMP values is that, since date and time values may contain a variety of date parts, and may be formatted in different ways based on the above option settings, the LIKE pattern has to be written carefully to succeed.

For example, if you insert the value 9:20 and the current date into a TIMESTAMP column named arrival_time, the following clause will evaluate to TRUE if the timestamp_format option formats the time portion of the value using colons to separate hours and minutes:

```
WHERE arrival_time LIKE '%09:20%'
```

In contrast to LIKE, search conditions that contain a simple comparison between a string literal and a DATE, TIME, TIMESTAMP, or TIMESTAMP WITH TIME ZONE value use the date/time data type as the comparison domain. In this case, SQL Anywhere first converts the string literal to a TIMESTAMP value and then uses the necessary portion(s) of that value to perform the comparison. SQL Anywhere follows the ISO 8601 standard for converting TIME, DATE, and TIMESTAMP values, with additional extensions.

For example, the clause below will evaluate to TRUE because the constant string value 9:20 is converted to a TIMESTAMP using 9:20 as the time portion and the current date for the date portion:

```
WHERE arrival_time = '9:20'
```

Using NOT LIKE

With NOT LIKE, you can use the same wildcard characters that you can use with LIKE. To find all the phone numbers in the Contacts table that do not have 415 as the area code, you can use either of these queries:

```
SELECT Phone
   FROM Contacts
   WHERE Phone NOT LIKE '415%';

SELECT Phone
   FROM Contacts
   WHERE NOT Phone LIKE '415%';
```

Using underscores

Another special character that can be used with LIKE is the _ (underscore) character, which matches exactly one character. For example, the pattern 'BR_U%' matches all names starting with BR and having U as the fourth letter. In Braun the _ character matches the letter A and the % matches N. See "LIKE search condition" [*SQL Anywhere Server - SQL Reference*].

Character strings and quotation marks

When you enter or search for character and date data, you must enclose it in single quotes, as in the following example.

```
SELECT GivenName, Surname
   FROM Contacts
   WHERE GivenName = 'John';
```

If the quoted_identifier database option is set to Off (it is On by default), you can also use double quotes around character or date data.

To set the quoted_identifier option off for the current user ID

- Enter the following command:

  ```
  SET OPTION quoted_identifier = 'Off';
  ```

The quoted_identifier option is provided for compatibility with Adaptive Server Enterprise. By default, the Adaptive Server Enterprise option is quoted_identifier Off and the SQL Anywhere option is quoted_identifier On. See "quoted_identifier option" [*SQL Anywhere Server - Database Administration*].

Quotation marks in strings

There are two ways to specify literal quotations within a character entry. The first method is to use two consecutive quotation marks. For example, if you have begun a character entry with a single quotation mark and want to include a single quotation mark as part of the entry, use two single quotation marks:

```
'I don''t understand.'
```

With double quotation marks (quoted_identifier Off), specify:

```
"He said, ""It is not really confusing."""
```

The second method, applicable only with quoted_identifier Off, is to enclose a quotation in the other kind of quotation mark. In other words, surround an entry containing double quotation marks with single quotation marks, or vice versa. Here are some examples:

```
'George said, "There must be a better way."'
"Isn't there a better way?"
'George asked, "Isn''t there a better way?"'
```

Unknown Values: NULL

A NULL in a column means that the user or application has made no entry in that column. That is, a data value for the column is unknown or not available.

NULL does not mean the same as zero (numerical values) or blank (character values). Rather, NULL values allow you to distinguish between a deliberate entry of zero for numeric columns or blank for character columns and a non-entry, which is NULL for both numeric and character columns.

Entering NULL

NULL can be entered only where NULL values are permitted for the column. Whether a column can accept NULL values is determined when the table is created. Assuming a column can accept NULL values, NULL is inserted:

- **Default** If no data is entered, and the column has no other default setting.
- **Explicit entry** You can explicitly insert the word NULL without quotation marks. If the word NULL is typed in a character column with quotation marks, it is treated as data, not as the NULL value.

For example, the DepartmentHeadID column of the Departments table allows NULL values. You can enter two rows for departments with no manager as follows:

```
INSERT INTO Departments (DepartmentID, DepartmentName)
   VALUES (201, 'Eastern Sales')
INSERT INTO Departments
   VALUES (202, 'Western Sales', null);
```

Returning NULL values

NULL values are returned to the client application for display, just as with other values. For example, the following example illustrates how NULL values are displayed in Interactive SQL:

```
SELECT *
FROM Departments;
```

DepartmentID	DepartmentName	DepartmentHeadID
100	R & D	501
200	Sales	904
300	Finance	1293
400	Marketing	1576
500	Shipping	703
201	Eastern Sales	(null)
202	Western Sales	(null)

Testing a column for NULL

You can use the IS NULL search conditions to compare column values to NULL, and to select them or perform a particular action based on the results of the comparison. Only columns that return a value of TRUE are selected or result in the specified action; those that return FALSE or UNKNOWN do not.

The following example selects only rows for which UnitPrice is less than $15 or is NULL:

```
SELECT Quantity, UnitPrice
   FROM Products
   WHERE UnitPrice < 15
   OR UnitPrice IS NULL;
```

The result of comparing any value to NULL is UNKNOWN, since it is not possible to determine whether NULL is equal (or not equal) to a given value or to another NULL.

There are some conditions that never return true, so that queries using these conditions do not return result sets. For example, the following comparison can never be determined to be true, since NULL means having an unknown value:

```
WHERE column1 > NULL
```

This logic also applies when you use two column names in a WHERE clause, that is, when you join two tables. A clause containing the condition `WHERE column1 = column2` does not return rows where the columns contain NULL.

You can also find NULL or non-NULL with these patterns:

```
WHERE column_name IS NULL

WHERE column_name IS NOT NULL
```

For example:

```
WHERE advance < $5000
OR advance IS NULL
```

See “NULL value” [*SQL Anywhere Server - SQL Reference*].

Properties of NULL

The following list expands on the properties of a NULL value.

- **The difference between FALSE and UNKNOWN** Although neither FALSE nor UNKNOWN returns values, there is an important logical difference between FALSE and UNKNOWN; the opposite of false ("not false") is true, whereas the opposite of UNKNOWN does not mean something is known. For example, `1 = 2` evaluates to false, and `1 != 2` (1 does not equal 2) evaluates to true.

  But if a NULL is included in a comparison, you cannot negate the expression to get the opposite set of rows or the opposite truth value. An UNKNOWN value remains UNKNOWN.

- **Substituting a value for NULL values** You can use the ISNULL built-in function to substitute a particular value for NULL values. The substitution is made only for display purposes; actual column values are not affected. The syntax is:

  ISNULL(*expression*, *value*)

  For example, use the following statement to select all the rows from Departments, and display all the NULL values in column DepartmentHeadID with the value -1.

  ```
  SELECT DepartmentID,
         DepartmentName,
         ISNULL( DepartmentHeadID, -1 ) AS DepartmentHead
     FROM Departments;
  ```

- **Expressions that evaluate to NULL** An expression with an arithmetic or bitwise operator evaluates to NULL if any of the operands are the NULL value. For example, `1 + column1` evaluates to NULL if column1 is NULL. See "Arithmetic operators" [*SQL Anywhere Server - SQL Reference*], and "Bitwise operators" [*SQL Anywhere Server - SQL Reference*].

- **Concatenating strings and NULL** If you concatenate a string and NULL, the expression evaluates to the string. For example, the following statement returns the string abcdef:

  ```
  SELECT 'abc' || NULL || 'def';
  ```

Connecting conditions with logical operators

The logical operators AND, OR, and NOT are used to connect search conditions in WHERE clauses. When more than one logical operator is used in a statement, AND operators are normally evaluated before OR operators. You can change the order of execution with parentheses.

Using AND

The AND operator joins two or more conditions and returns results only when all the conditions are true. For example, the following query finds only the rows in which the contact's last name is Purcell and the contact's first name is Beth.

```
SELECT *
   FROM Contacts
   WHERE GivenName = 'Beth'
      AND Surname = 'Purcell';
```

Using OR

The OR operator connects two or more conditions and returns results when *any* of the conditions is true. The following query searches for rows containing variants of Elizabeth in the GivenName column.

```
SELECT *
   FROM Contacts
   WHERE GivenName = 'Beth'
      OR GivenName = 'Liz';
```

Using NOT

The NOT operator negates the expression that follows it. The following query lists all the contacts who do not live in California:

```
SELECT *
   FROM Contacts
   WHERE NOT State = 'CA';
```

Comparing dates in search conditions

You can use operators other than equals to select a set of rows that satisfy the search condition. The inequality operators (< and >) can be used to compare numbers, dates, and even character strings.

List all employees born before March 13, 1964

- In Interactive SQL, execute the following query:

```
SELECT Surname, BirthDate
   FROM Employees
   WHERE BirthDate < 'March 13, 1964'
   ORDER BY BirthDate DESC;
```

Surname	BirthDate
Ahmed	1963-12-12
Dill	1963-07-19
Rebeiro	1963-04-12
Garcia	1963-01-23
Pastor	1962-07-14
...	...

Notes

- **Automatic conversion to dates** The SQL Anywhere database server knows that the BirthDate column contains dates, and automatically converts the string `'March 13, 1964'` to a date.

- **Ways of specifying dates** There are many ways of specifying dates. For example:

```
'March 13, 1964'
'1964/03/13'
'1964-03-13'
```

  You can configure the interpretation of dates in queries by setting the date_order option database option. See “date_order option” [*SQL Anywhere Server - Database Administration*].

Dates in the format *yyyy/mm/dd* or *yyyy-mm-dd* are always recognized unambiguously as dates, regardless of the date_order setting.

- **Other comparison operators** SQL Anywhere supports several comparison operators. See "Comparison operators" [*SQL Anywhere Server - SQL Reference*].

Matching rows by sound

With the SOUNDEX function, you can match rows by sound. For example, suppose a phone message was left for a name that sounded like "Ms. Brown". You could execute the following query to search for employees that have names that sound like Brown.

List employees with a last name that sound like Brown

- In Interactive SQL, execute the following query:

```
SELECT Surname, GivenName
   FROM Employees
   WHERE SOUNDEX( Surname ) = SOUNDEX( 'Brown' );
```

Surname	GivenName
Braun	Jane

The algorithm used by SOUNDEX makes it useful mainly for English-language databases. See "SOUNDEX function [String]" [*SQL Anywhere Server - SQL Reference*].

The ORDER BY clause: Ordering results

Unless otherwise requested, the database server returns the rows of a table in an order that has no meaning. Often it is useful to look at the rows in a table in a more meaningful sequence. For example, you might like to see products in alphabetical order.

You order the rows in a result set by adding an ORDER BY clause to the end of the SELECT statement. This SELECT statement has the following syntax:

SELECT *column-name-1*, *column-name-2*,...
FROM *table-name*
ORDER BY *order-by-column-name*

You must replace *column-name-1*, *column-name-2*, and *table-name* with the names of the columns and table you are querying, and *order-by-column-name* with a column in the table. As before, you can use the asterisk as a short form for all the columns in the table.

List the products in alphabetical order

- In Interactive SQL, execute the following query:

```
SELECT ID, Name, Description
   FROM Products
   ORDER BY Name;
```

ID	Name	Description
400	Baseball Cap	Cotton Cap
401	Baseball Cap	Wool cap
700	Shorts	Cotton Shorts
600	Sweatshirt	Hooded Sweatshirt
...	...	...

Notes

- **The order of clauses is important** The ORDER BY clause must follow the FROM clause and the SELECT clause.
- **You can specify either ascending or descending order** The default order is ascending. You can specify a descending order by adding the keyword DESC to the end of the clause, as in the following query:

```
SELECT ID, Quantity
   FROM Products
   ORDER BY Quantity DESC;
```

ID	Quantity
400	112
700	80
302	75
301	54
600	39
...	...

- **You can order by several columns** The following query sorts first by size (alphabetically), and then by name:

```
SELECT ID, Name, Size
   FROM Products
   ORDER BY Size, Name;
```


ID	Name	Size
600	Sweatshirt	Large
601	Sweatshirt	Large
700	Shorts	Medium
301	Tee Shirt	Medium
...	...	...

- **The ORDER BY column does not need to be in the select list** The following query sorts products by unit price, even though the price is not included in the result set:

```
SELECT ID, Name, Size
   FROM Products
   ORDER BY UnitPrice;
```

ID	Name	Size
500	Visor	One size fits all
501	Visor	One size fits all
300	Tee Shirt	Small
400	Baseball Cap	One size fits all
...	...	...

- **If you do not use an ORDER BY clause, and you execute a query more than once, you may appear to get different results** This is because SQL Anywhere may return the same result set in a different order. In the absence of an ORDER BY clause, SQL Anywhere returns rows in whatever order is most efficient. This means the appearance of result sets may vary depending on when you last accessed the row and other factors. The only way to ensure that rows are returned in a particular order is to use ORDER BY.

Using indexes to improve ORDER BY performance

Sometimes there is more than one possible way for the SQL Anywhere database server to execute a query with an ORDER BY clause. You can use indexes to enable the database server to search the tables more efficiently.

Queries with WHERE and ORDER BY clauses

An example of a query that can be executed in more than one possible way is one that has both a WHERE clause and an ORDER BY clause.

```
SELECT *
   FROM Customers
   WHERE ID > 300
   ORDER BY CompanyName;
```

In this example, SQL Anywhere must decide between two strategies:

1. Go through the entire Customers table in order by company name, checking each row to see if the customer ID is greater than 300.

2. Use the key on the ID column to read only the companies with ID greater than 300. The results would then need to be sorted by company name.

If there are very few ID values greater than 300, the second strategy is better because only a few rows are scanned and quickly sorted. If most of the ID values are greater than 300, the first strategy is much better because no sorting is necessary.

Solving the problem

Creating a two-column index on ID and CompanyName could solve the example above. SQL Anywhere can use this index to select rows from the table in the correct order. However, keep in mind that indexes take up space in the database file and involve some overhead to keep up to date. Do not create indexes indiscriminately. See "Using indexes" on page 225.

Aggregate functions

Some queries examine aspects of the data in your table that reflect properties of groups of rows rather than of individual rows. For example, you may want to find the average amount of money that a customer pays for an order, or to see how many employees work for each department. For these types of tasks, you use **aggregate** functions and the GROUP BY clause.

List the number of employees in the company

- In Interactive SQL, execute the following query:

```
SELECT COUNT( * )
   FROM Employees;
```

COUNT(*)
75

The result set consists of only one column, with title COUNT(*), and one row, which contains the total number of employees.

List the number of employees in the company and the birth dates of the oldest and youngest employee

- In Interactive SQL, execute the following query:


```
SELECT COUNT( * ), MIN( BirthDate ), MAX( BirthDate )
   FROM Employees;
```

COUNT(*)	MIN(Employees.BirthDate)	MAX(Employees.BirthDate)
75	1936-01-02	1973-01-18

The functions COUNT, MIN, and MAX are called **aggregate functions**. Aggregate functions summarize information. Other aggregate functions include statistical functions such as AVG, STDDEV, and VARIANCE. All but COUNT require a parameter. See "Aggregate functions" [*SQL Anywhere Server - SQL Reference*].

Aggregate functions return a single value for a set of rows. If there is no GROUP BY clause, the aggregate function is called a **scalar aggregate** and it returns a single value for all the rows that satisfy other aspects of the query. If there is a GROUP BY clause, the aggregate is termed a **vector aggregate** and it returns a value for each group.

SQL Anywhere supports additional aggregate functions for analytics, sometimes referred to as OLAP functions. Several of these functions can be used as window functions: they include RANK, PERCENT_RANK, CUME_DIST, ROW_NUMBER, and functions to support linear regression analysis. See "OLAP support" on page 445.

Applying aggregate functions to grouped data

In addition to providing information about an entire table, aggregate functions can be used on groups of rows. The GROUP BY clause arranges rows into groups, and aggregate functions return a single value for each group of rows.

Example

List the sales representatives and the number of orders each has taken

- In Interactive SQL, execute the following query:

```
SELECT SalesRepresentative, COUNT( * )
   FROM SalesOrders
   GROUP BY SalesRepresentative
   ORDER BY SalesRepresentative;
```

SalesRepresentative	count(*)
129	57
195	50
299	114
467	56

SalesRepresentative	count(*)
...	...

A GROUP BY clause tells SQL Anywhere to partition the set of all the rows that would otherwise be returned. All rows in each partition, or group, have the same values in the named column or columns. There is only one group for each unique value or set of values. In this case, all the rows in each group have the same SalesRepresentative value.

Aggregate functions such as COUNT are applied to the rows in each group. So, this result set displays the total number of rows in each group. The results of the query consist of one row for each sales rep ID number. Each row contains the sales rep ID, and the total number of sales orders for that sales representative.

Whenever GROUP BY is used, the resulting table has one row for each column or set of columns named in the GROUP BY clause. See "The GROUP BY clause: Organizing query results into groups" on page 370.

Semantic differences with the empty set

The SQL language treats the empty set differently when using aggregate functions. Without a GROUP BY clause, a query containing an aggregate function over zero input rows will return a single row as the result. In the case of COUNT, its result will be the value zero, and with all other aggregate functions the result will be NULL. However, if the query contains a GROUP BY clause, and the input to the query is empty, then the query's result is empty and no rows are returned.

For example, the following query returns a single row with the value 0; there are no employees in department 103.

```
SELECT COUNT() FROM Employees WHERE DepartmentID = 103;
```

However, this modified query returns no rows, due to the presence of the GROUP BY clause.

```
SELECT COUNT() FROM Employees WHERE DepartmentID = 103 GROUP BY State;
```

A common error with GROUP BY

A common error with GROUP BY is to try to get information that cannot properly be put in a group. For example, the following query gives an error.

```
SELECT SalesRepresentative, Surname, COUNT( * )
   FROM SalesOrders KEY JOIN Employees
   GROUP BY SalesRepresentative;
```

The error message indicates that a reference to the Surname column must also appear in the GROUP BY clause. This error occurs because SQL Anywhere cannot verify that each of the result rows for an employee with a given ID have the same last name.

To fix this error, add the column to the GROUP BY clause.

```
SELECT SalesRepresentative, Surname, COUNT( * )
   FROM SalesOrders KEY JOIN Employees
   GROUP BY SalesRepresentative, Surname
   ORDER BY SalesRepresentative;
```


If this is not appropriate, you can instead use an aggregate function to select only one value:

```
SELECT SalesRepresentative, MAX( Surname ), COUNT( * )
   FROM SalesOrders KEY JOIN Employees
   GROUP BY SalesRepresentative
   ORDER BY SalesRepresentative;
```

The MAX function chooses the maximum (last alphabetically) Surname from the detail rows for each group. This statement is valid because there can be only one distinct maximum value. In this case, the same Surname appears on every detail row within a group.

Restricting groups

You have already seen how to restrict rows in a result set using the WHERE clause. You restrict the rows in groups using the HAVING clause.

List all sales representatives with more than 55 orders

- In Interactive SQL, execute the following query:

```
SELECT SalesRepresentative, COUNT( * ) AS orders
FROM SalesOrders KEY JOIN Employees
GROUP BY SalesRepresentative
HAVING count( * ) > 55
ORDER BY orders DESC;
```

SalesRepresentative	orders
299	114
129	57
1142	57
467	56

See also "The HAVING clause: selecting groups of data" on page 376.

Combining WHERE and HAVING clauses

Sometimes you can specify the same set of rows using either a WHERE clause or a HAVING clause. In such cases, one method is not more or less efficient than the other. The optimizer always automatically analyzes each statement you enter and selects an efficient means of executing it. It is best to use the syntax that most clearly describes the intended result. In general, that means eliminating undesired rows in earlier clauses.

Example

To list all sales reps with more than 55 orders and an ID of more than 1000, enter the following statement.

```
SELECT SalesRepresentative, COUNT( * )
FROM SalesOrders
WHERE SalesRepresentative > 1000
GROUP BY SalesRepresentative
HAVING count( * ) > 55
ORDER BY SalesRepresentative;
```

The following statement produces the same results.

```
SELECT SalesRepresentative, COUNT( * )
FROM SalesOrders
GROUP BY SalesRepresentative
HAVING count( * ) > 55 AND SalesRepresentative > 1000
ORDER BY SalesRepresentative;
```

SQL Anywhere detects that both statements describe the same result set, and so executes each efficiently.

Full text search

What is full text search?

Full text search is a more advanced way to search a database. Full text search quickly finds all instances of a term (word) in a table without having to scan rows and without having to know which column a term is stored in. Full text search works by using **text indexes**. A text index stores positional information for all terms found in the columns you create the text index on. Using a text index can be faster than using a regular index to find rows containing a given value. See "How to manage text indexes" on page 339.

Full text search capability in SQL Anywhere differs from searching using predicates such as LIKE, REGEXP, and SIMILAR TO, because the matching is term-based, not pattern-based.

String comparisons in full text search use all the normal collation settings for the database. For example, if the database is configured to be case insensitive, then full text searches will be case insensitive. See "Understanding collations" [*SQL Anywhere Server - Database Administration*].

Except where noted, full text search leverages all the international features supported by SQL Anywhere. See "International languages and character sets" [*SQL Anywhere Server - Database Administration*].

To perform a full text search on a database containing Chinese, Japanese, and Korean (CJK) data, see the whitepaper "Performing Full Text Searches on Chinese, Japanese, and Korean Data in SQL Anywhere 11" at http://www.sybase.com/detail?id=1061814.

Considerations when using full text search

There are some considerations to make when deciding whether to use full text indexes over regular indexes:

- You cannot use aliases in a CONTAINS clause or a CONTAINS search condition.
- When using duplicate correlation names in a query, a CONTAINS (FROM CONTAINS()) is only supported on the first instance of the correlation name. For example, the following syntax returns an error because of the second CONTAINS predicate involving A:


```
SELECT *
FROM CONTAINS(A contains-query-string) JOIN B ON A.x = B.x,
     CONTAINS(A contains-query-string) JOIN C ON A.y = C.y;
```

When using external term breaker and prefilter libraries, there are several additional considerations:

- **Querying and updating** The external library must remain available for any operations that require updating, querying, or altering the text indexes built using the libraries.
- **Unloading and reloading** The external library must be available during unloading and reloading of data of data associated with the full text index.
- **Database recovery** The external library must be available to recover the database. This is because the database can not recover if there are operations in the transaction log that involved the external library since the last checkpoint.

See also

- "CONTAINS search condition" [*SQL Anywhere Server - SQL Reference*]
- "External term breaker and prefilter libraries" on page 346

Terms dropped from the text index and CONTAINS queries

Text indexes are built according to the settings defined for the text configuration object used to create the text index. A term does not appear in a text index if one or more of the following conditions are true:

- the term is included in the stoplist
- the term is shorter than the minimum term length (GENERIC only)
- the term is longer than the maximum term length

The same rules apply to query strings. The dropped term can match zero or more terms at the end or beginning of the phrase. For example, suppose the term `'the'` is in the stoplist:

- If the term appears on either side of an AND, OR, or NEAR, then both the operator and the term are removed. For example, searching for `'the AND apple'`, `'the OR apple'`, or `'the NEAR apple'` are equivalent to searching for `'apple'`.
- If the term appears on the right side of an AND NOT, both the AND NOT and the term are dropped. For example, searching for `'apple AND NOT the'` is equivalent to searching for `'apple'`.

  If the term appears on the left side of an AND NOT, the entire expression is dropped and no rows are returned. For example, `'orange and the AND NOT apple' = 'orange'`
- If the term appears in a phrase, the phrase is allowed to match with any term at the dropped term's position. For example, searching for `'feed the dog'` matches `'feed the dog'`, `'feed my dog'`, `'feed any dog'`, and so on.

If none of the terms you are searching for are in the text index, no rows are returned. For example, suppose both `'the'` and `'a'` are in the stoplist. Searching for `'a OR the'` returns no rows.

See also

- "How to manage text configuration objects" on page 323

Scoring full text search results

When you include a CONTAINS clause in the FROM clause of a query, each match has a score associated with it. The score indicates how close the match is, and you can use score information to sort the data.

Scoring is based on two main criteria:

- **Number of times a term appears in the indexed row** The more times a term appears in an indexed row, the higher its score.
- **Number of times a term appears in the text index** The more times a term appears in a text index, the lower its score. In Sybase Central, you can view how many times a term appears in the text index by viewing the **Vocabulary** tab for the text index. Choose the **term** column to sort the terms alphabetically. The **freq** column tells you how many times the term appears in the text index.

Then, depending on the type of full text search, other criteria impact scoring. For example, in proximity searches, the proximity of search terms impacts scoring.

How to use scores

By default, the result set of a CONTAINS clause has the correlation name **contains** that has a single column in it called **score**. You can refer to `"contains".score` in the SELECT list, ORDER BY clause, or other parts of the query. However, because contains is a SQL reserved word, you must remember to put it in double quotes. Alternatively, you can specify another correlation name such (for example, `CONTAINS ( expression ) AS ct`). In the documentation examples for full text search, the score column is referred to as `ct.score`.

The following statement searches MarketingInformation.Description for terms starting with **stretch** or terms starting with **comfort**:

```
SELECT ID, ct.score, Description
    FROM MarketingInformation CONTAINS ( MarketingInformation.Description,
'stretch* | comfort*' ) AS ct
    ORDER BY ct.score DESC;
```


ID	score	Description
910	5.570408968026068	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Shorts</title></head><body lang=EN-US> These quick-drying cotton shorts provide all day **comfort** on or off the trails. Now with a more **comfort**able and **stretch**y fabric and an adjustable drawstring waist. </body></html>
907	3.658418186470189	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Visor</title></head><body lang=EN-US> A polycarbonate visor with an abrasion-resistant coating on the outside. Great for jogging in the spring, summer, and early fall. The elastic headband has plenty of **stretch** to give you a snug yet **comfort**able fit every time you wear it. </body></html>

ID	score	Description
905	1.6750365447462499	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Baseball Cap</title></head><body lang=EN-US> A lightweight wool cap with mesh side vents for breathable **comfort** during aerobic activities. Moisture-absorbing headband liner. </body></html>

Item 910 has the highest score because it contains two instances of the prefix term **comfort**, whereas the others only have one instance. As well, item 910 has an instance of the prefix term **stretch.**

Example 2: Searching multiple columns

The following example shows you how to perform a full text search across multiple columns and score the results:

1. Create an immediate text index on the Products table as follows:

```
CREATE TEXT INDEX scoringExampleMult
   ON Products ( Description, Name );
```

2. Perform a full text search on the Description and Name columns for the terms **cap** or **visor**, as follows. The result of the CONTAINS clause is assigned the correlation name ct, and is referenced in the SELECT list so that it is included in the results. Also, the ct.score column is referenced in the ORDER BY clause to sort the results in descending order by score.

```
SELECT Products.Description, Products.Name, ct.score
    FROM Products CONTAINS ( Products.Description, Products.Name, 'cap OR
visor' ) ct
    ORDER BY ct.score DESC;
```

Description	Name	score
Cloth Visor	Visor	3.5635154905713042
Plastic Visor	Visor	3.4507856451176244
Wool cap	Baseball Cap	3.2340501745357333
Cotton Cap	Baseball Cap	3.090467108972918

The scores for a multi-column search are calculated as if the column values were concatenated together and indexed as a single value. Note, however, that phrases and NEAR operators never match across column boundaries, and that a search term that appears in more than one column increases the score more than it would in a single concatenated value.

3. For other examples in the documentation to work properly, you must delete the text index you created on the Products table. To do so, execute the following statement:

```
DROP TEXT INDEX scoringExampleMult ON Products;
```

How to perform a full text search

You can perform a full text query either by using a CONTAINS clause in the FROM clause of a SELECT statement, or by using a CONTAINS search condition (predicate) in a WHERE clause. Both return the same rows; however, use a CONTAINS clause in a FROM clause also returns scores for the matching rows.

The following examples show how the CONTAINS clause and search condition are used in a query. However, if you run these examples in Interactive SQL they will fail because you have not built a text index on MarketingInformation.Description yet.

```
SELECT *
   FROM MarketingInformation CONTAINS ( Description, 'cotton' );

SELECT *
   FROM MarketingInformation
   WHERE CONTAINS ( Description, 'cotton' );
```

Types of full text searches

Using full text search, you can search for **terms**, **phrases** (sequences of terms), or **prefixes**. You can also combine multiple terms, phrases, or prefixes into boolean expressions, or require that expressions appear near to each other with proximity searches.

You perform a full text search using a CONTAINS clause in either a WHERE clause or a FROM clause of a SELECT statement. You can also perform a full text search as part of the IF search condition (for example, `SELECT IF CONTAINS...`).

The following sections show you how to perform the different types of full text search available in SQL Anywhere.

Term and phrase searching

When performing a full text search for a list of terms, the order of terms is not important unless they are within a phrase. If you put the terms within a phrase, the database server looks for those terms in exactly the same order, and same relative positions, in which you specified them.

When performing a term or phrase search, if terms are dropped from the query because they exceed term length settings or because they are in the stoplist, you can get back a different number of rows than you expect. This is because removing the terms from the query is equivalent to changing your search criteria.

For example, if you search for the phrase `'"grown cotton"'` and grown is in the stoplist, you get every indexed row containing cotton.

You can search for the terms that are considered keywords of the CONTAINS clause grammar, as long as they are within phrases.

Term searching

In the sample database, a text index called MarketingTextIndex has been built on the Description column of the MarketingInformation table. The following statement queries the MarketingInformation.Description column and returns the rows where the value in the Description column contains the term **cotton**.

```
SELECT ID, Description
   FROM MarketingInformation
   WHERE CONTAINS ( Description, 'cotton' );
```

ID	Description
906	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Visor</title></head><body lang=EN-US> Light-weight 100% organically grown **cotton** construction. Shields against sun and precipitation.cotton Metallic ions in the fibers inhibit bacterial growth, and help neutralize odor. </body></html>
908	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Sweatshirt</title></head><body lang=EN-US> Lightweight 100% organically grown **cotton** hooded sweatshirt with taped neck seams. Comes pre-washed for softness and to lessen shrinkage. </body></html>
909	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Sweatshirt</title></head><body lang=EN-US> Top-notch construction includes durable topstitched seams for strength with low-bulk, resilient rib-knit collar, cuffs and bottom. An 80% **cotton**/20% polyester blend makes it easy to keep them clean. </body></html>
910	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Shorts</title></head><body lang=EN-US> These quick-drying **cotton** shorts provide all day comfort on or off the trails. Now with a more comfortable and stretchy fabric and an adjustable drawstring waist. </body></html>

The following example queries the MarketingInformation table and returns a single value for each row indicating whether the value in the Description column contains the term **cotton**.

```
SELECT ID, IF CONTAINS ( Description, 'cotton' )
      THEN 1
      ELSE 0
      ENDIF AS Results
   FROM MarketingInformation;
```

ID	Results
901	0
902	0
903	0
904	0
905	0
906	1
907	0
908	1
909	1
910	1

The next example queries the MarketingInformation table for items that have the term **cotton** the Description column, and shows the score for each match.

```
SELECT ID, ct.score, Description
   FROM MarketingInformation CONTAINS ( MarketingInformation.Description,
'cotton' ) as ct
   ORDER BY ct.score DESC;
```

ID	score	Description
908	0.9461597363521859	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Sweatshirt</title></head><body lang=EN-US> Lightweight 100% organically grown **cotton** hooded sweatshirt with taped neck seams. Comes pre-washed for softness and to lessen shrinkage. </body></html>

ID	score	Description
910	0.9244136988525732	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Shorts</title></head><body lang=EN-US> These quick-drying **cotton** shorts provide all day comfort on or off the trails. Now with a more comfortable and stretchy fabric and an adjustable drawstring waist. </body></html>
906	0.9134171046194403	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Visor</title></head><body lang=EN-US> Lightweight 100% organically grown **cotton** construction. Shields against sun and precipitation. Metallic ions in the fibers inhibit bacterial growth, and help neutralize odor. </body></html>
909	0.8856420222728282	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Sweatshirt</title></head><body lang=EN-US> Top-notch construction includes durable topstitched seams for strength with low-bulk, resilient rib-knit collar, cuffs and bottom. An 80% **cotton**/20% polyester blend makes it easy to keep them clean. </body></html>

For more information about scoring results when CONTAINS is used in the FROM clause of a query, see "Scoring full text search results" on page 290.

Phrase searching

When performing a full text search for a phrase, you enclose the phrase in double quotes. A column matches if it contains the terms in the specified order and relative positions.

You cannot specify CONTAINS keywords, such as AND or FUZZY, as terms to search for unless you place them inside a phrase (single term phrases are allowed). For example, the statement below is acceptable even though NOT is a CONTAINS keyword. For a list of CONTAINS keywords and special characters, see "CONTAINS search condition" [*SQL Anywhere Server - SQL Reference*].

```
SELECT * FROM table-name CONTAINS ( Remarks, '"NOT"' );
```

With the exception of asterisk, special characters are not interpreted as special characters when they are in a phrase.

Phrases cannot be used as arguments for proximity searches.

The following statement queries MarketingInformation.Description for the phrase `"grown cotton"`, and shows the score for each match:

```
SELECT ID, ct.score, Description
   FROM MarketingInformation CONTAINS ( MarketingInformation.Description,
'"grown cotton"' ) as ct
   ORDER BY ct.score DESC;
```

ID	score	Description
908	1.6619019465461564	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Sweatshirt</title></head><body lang=EN-US> Lightweight 100% organically grown **cotton** hooded sweatshirt with taped neck seams. Comes pre-washed for softness and to lessen shrinkage. </body></html>
906	1.6043904700786786	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Visor</title></head><body lang=EN-US> Lightweight 100% organically grown **cotton** construction. Shields against sun and precipitation. Metallic ions in the fibers inhibit bacterial growth, and help neutralize odor. </body></html>

Prefix searching

The full text search feature allows you to search for the beginning portion of a term. This is called a **prefix search**. To perform a prefix search, you specify the prefix you want to search for, followed by an asterisk. This is called a **prefix term**.

Keywords for the CONTAINS clause cannot be used for prefix searching unless they are in a phrase. For a list of CONTAINS keywords, see "CONTAINS search condition" [*SQL Anywhere Server - SQL Reference*].

You also can specify multiple prefix terms in a query string, including within phrases (for example, `'"shi* fab"'`).

For complete syntax restrictions for prefix searching, see "Allowed syntax for asterisk (*)" [*SQL Anywhere Server - SQL Reference*].

The following example queries the MarketingInformation table for items that start with the prefix **shi**:

```
SELECT ID, ct.score, Description
   FROM MarketingInformation CONTAINS ( MarketingInformation.Description,
'shi*' ) AS ct
   ORDER BY ct.score DESC;
```

ID	score	Description
906	2.295363835537917	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Visor</title></head><body lang=EN-US> Lightweight 100% organically grown cotton construction. **Shi**elds against sun and precipitation. Metallic ions in the fibers inhibit bacterial growth, and help neutralize odor. </body></html>
901	1.6883275743936228	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Tee **Shi**rt</title></head><body lang=EN-US> We've improved the design of this perennial favorite. A sleek and technical **shi**rt built for the trail, track, or sidewalk. UPF rating of 50+. </body></html>
903	1.6336529491832605	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Tee **Shi**rt</title></head><body lang=EN-US> A sporty, casual **shi**rt made of recycled water bottles. It will serve you equally well on trails or around town. The fabric has a wicking finish to pull perspiration away from your skin. </body></html>
902	1.6181703448678983	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Tee **Shi**rt</title></head><body lang=EN-US> This simple, sleek, and lightweight technical **shi**rt is designed for high-intensity workouts in hot and humid weather. The recycled polyester fabric is gentle on the earth and soft against your skin. </body></html>

ID 906 has the highest score because the term shield occurs less frequently than shirt in the text index.

Prefix searches on GENERIC text indexes

On GENERIC text indexes, the behavior for prefix searches is as follows:

- If a prefix term is longer than the MAXIMUM TERM LENGTH, it is dropped from the query string since there can be no terms in the text index that exceed the MAXIMUM TERM LENGTH. So, on a text index with MAXIMUM TERM LENGTH 3, searching for `'red appl*'` is equivalent to searching for `'red'`.

- If a prefix term is shorter than MINIMUM TERM LENGTH, and is not part of a phrase search, the prefix search proceeds normally. So, on a GENERIC text index where MINIMUM TERM LENGTH is 5, searching for `'macintosh a*'` returns indexed rows that contain macintosh and any terms of length 5 or greater that start with a.

- If a prefix term is shorter than MINIMUM TERM LENGTH, but is part of a phrase search, the prefix term is dropped from the query. So, on a GENERIC text index where MINIMUM TERM LENGTH is 5, searching for `'"macintosh appl* turnover"'` is equivalent to searching for macintosh followed by any term followed by turnover. A row containing `"macintosh turnover"` will not be found; there must be a term between macintosh and turnover.

Prefix searches on NGRAM text indexes

On NGRAM text indexes, prefix searching can return unexpected results since an NGRAM text index contains only n-grams, and contains no information about the beginning of terms. Query terms are also broken into n-grams, and searching is performed using the n-grams not the query terms. Because of this, the following behaviors should be noted:

- If a prefix term is shorter than the n-gram length (MAXIMUM TERM LENGTH), the query returns all indexed rows that contain n-grams starting with the prefix term. For example, on a 3-gram text index, searching for `'ea*'` returns all indexed rows containing n-grams starting with ea. So, if the terms weather and fear were indexed, the rows would be considered matches since their n-grams include eat and ear, respectively.

- If a prefix term is longer than n-gram length, and is not part of a phrase, and not an argument in a proximity search, the prefix term is converted to an n-grammed phrase and the asterisk is dropped. For example, on a 3-gram text index, searching for `'purple blac*'` is equivalent to searching for `'"pur urp rpl ple" AND "bla lac"'`.

- For phrases, the following behavior also takes place:

  - If the prefix term is the only term in the phrase, it is converted to an n-grammed phrase and the asterisk is dropped. For example, on a 3-gram text index, searching for `'"purpl*"'` is equivalent to searching for `'"pur urp rpl"'`.

  - If the prefix term is in the last position of the phrase, the asterisk is dropped and the terms are converted to a phrase of n-grams. For example, on a 3-gram text index, searching for `'"purple blac*"'` is equivalent to searching for `'"pur urp rpl ple bla lac"'`.

  - If the prefix term is not in the last position of the phrase, the phrase is broken up into phrases that are ANDed together. For example, on a 3-gram text index, searching for `'"purp* blac*"'` is equivalent to searching for `'"pur urp" AND "bla lac"'`.

- If a prefix term is an argument in a proximity search, the proximity search is converted to an AND. For example, on a 3-gram text index, searching for `'red NEAR[1] appl*'` is equivalent to searching for `'red AND "app ppl"'`.

See also

- "How to manage text indexes" on page 339
- "CONTAINS search condition" [*SQL Anywhere Server - SQL Reference*]

Proximity searching

The full text search feature allows you to search for terms that are near each other in a single column. This is called a **proximity search**. To perform a proximity search, you specify two terms with either the keyword NEAR between them, or the tilde (~).

You can specify an integer argument with the NEAR keyword to specify the maximum distance. For example, *term1* `NEAR[5]` *term2* finds instances of *term1* that are within five terms of *term2*. The order of terms is not significant; `'term1 NEAR term2'` is equivalent to `'term2 NEAR term1'`.

If you do not specify a distance, the database server uses 10 as the default distance.

You can also specify a tilde (~) instead of the NEAR keyword. For example, `'term1 ~ term2'`. However, you cannot specify a distance when using the tilde form; the default of ten terms is applied.

You cannot specify a phrase as an argument in proximity searches.

In a proximity search, if you specify a prefix term as an argument, the proximity search is converted to an AND expression. For example, on a 3-gram text index, searching for `'red NEAR[1] appl*'` is equivalent to searching for `'red AND "app ppl"'`. Since this is no longer a proximity search, the search is no longer restricted to a single column in the case where multiple columns are specified in the CONTAINS clause.

Examples

Suppose you want to search MarketingInformation.Description for the term fabric within 10 terms of the term skin. You can execute the following statement.

```
SELECT ID, "contains".score, Description
   FROM MarketingInformation CONTAINS ( Description, 'fabric ~ skin' );
```


ID	score	Description
902	1.5572371866083279	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Tee Shirt</title></head><body lang=EN-US> This simple, sleek, and lightweight technical shirt is designed for high-intensity workouts in hot and humid weather. The recycled polyester **fabric** is gentle on the earth and soft against your **skin**. </body></html>

Since the default distance is 10 terms, you did not need to specify a distance. By extending the distance by one term, however, another row is returned:

```
SELECT ID, "contains".score, Description
   FROM MarketingInformation CONTAINS ( Description, 'fabric NEAR[11] skin' );
```

ID	score	Description
903	1.5787803210404958	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Tee Shirt</title></head><body lang=EN-US> A sporty, casual shirt made of recycled water bottles. It will serve you equally well on trails or around town. The **fabric** has a wicking finish to pull perspiration away from your **skin**. </body></html>
902	2.163125855043747	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Tee Shirt</title></head><body lang=EN-US> This simple, sleek, and lightweight technical shirt is designed for high-intensity workouts in hot and humid weather. The recycled polyester **fabric** is gentle on the earth and soft against your **skin**. </body></html>

The score for ID 903 is higher because the terms are closer together.

Boolean searching

You can specify multiple terms separated by Boolean operators when performing full text searches. SQL Anywhere supports the following Boolean operators when performing a full text search: AND, OR, and AND NOT.

For more information about the syntax for boolean searching, see "CONTAINS search condition" [*SQL Anywhere Server - SQL Reference*].

Using the AND operator in full text searches

The AND operator matches a row if it contains both of the terms specified on either side of the AND. You can also use an ampersand (&) for the AND operator. If terms are specified without an operator between them, AND is implied.

The order in which the terms are listed is not important.

For example, each of the following statements finds rows in MarketingInformation.Description that contain the term **fabric** and a term that begins with **ski**:

```
SELECT *
   FROM MarketingInformation
   WHERE CONTAINS ( MarketingInformation.Description, 'ski* AND fabric' );
SELECT *
   FROM MarketingInformation
   WHERE CONTAINS ( MarketingInformation.Description, 'fabric & ski*' );
SELECT *
   FROM MarketingInformation
   WHERE CONTAINS ( MarketingInformation.Description, 'ski* fabric' );
```

Using the OR operator in full text searches

The OR operator matches a row if it contains at least one of the specified search terms on either side of the OR. You can also use a vertical bar (|) for the OR operator; the two are equivalent.

The order in which the terms are listed is not important.

For example, either statement below returns rows in the MarketingInformation.Description that contain either the term **fabric** or a term that starts with **ski**:

```
SELECT *
   FROM MarketingInformation
   WHERE CONTAINS ( MarketingInformation.Description, 'ski* OR fabric' );
SELECT *
   FROM MarketingInformation
   WHERE CONTAINS ( MarketingInformation.Description, 'fabric | ski*' );
```

Using the AND NOT operator in full text searches

The AND NOT operator finds results that match the left argument and do not match the right argument. You can also use a hyphen (-) for the AND NOT operator; the two are equivalent.

For example, the following statements are equivalent and return rows that contain the term **fabric**, but do not contain any terms that begin with **ski**.

```
SELECT *
   FROM MarketingInformation
   WHERE CONTAINS ( MarketingInformation.Description, 'fabric AND NOT
ski*' );
SELECT *
   FROM MarketingInformation
   WHERE CONTAINS ( MarketingInformation.Description, 'fabric -ski*' );
SELECT *
```



```
   FROM MarketingInformation
   WHERE CONTAINS ( MarketingInformation.Description, 'fabric & -ski*' );
```

Combining different boolean operators

The boolean operators can be combined in a query string. For example, the following statements are equivalent and search the MarketingInformation.Description column for items that contain **fabric** and **skin**, but not **cotton**:

```
SELECT *
   FROM MarketingInformation
   WHERE CONTAINS ( MarketingInformation.Description, 'skin fabric -
cotton' );
SELECT *
   FROM MarketingInformation
   WHERE CONTAINS ( MarketingInformation.Description, 'fabric -cotton AND
skin' );
```

The following statements are equivalent and search the MarketingInformation.Description column for items that contain **fabric** or both **cotton** and **skin**:

```
SELECT *
   FROM MarketingInformation
   WHERE CONTAINS ( MarketingInformation.Description, 'fabric | cotton AND
skin' );
SELECT *
   FROM MarketingInformation
   WHERE CONTAINS ( MarketingInformation.Description, 'cotton skin OR
fabric' );
```

Grouping terms and phrases

Terms and expressions can be grouped with parentheses. For example, the following statement searches the MarketingInformation.Description column for items that contain **cotton** or **fabric**, and that have terms that start with **ski**.

```
SELECT ID, Description FROM MarketingInformation
  WHERE CONTAINS( MarketingInformation.Description, '( cotton OR fabric ) AND
shi*' );
```

	Description
902	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Tee Shirt</title></head><body lang=EN-US> This simple, sleek, and lightweight technical **shi**rt is designed for high-intensity workouts in hot and humid weather. The recycled polyester **fabric** is gentle on the earth and soft against your skin. </body></html>

	Description
903	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Tee Shirt</title></head><body lang=EN-US> A sporty, casual **shi**rt made of recycled water bottles. It will serve you equally well on trails or around town. The **fabric** has a wicking finish to pull perspiration away from your skin. </body></html>
906	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Visor</title></head><body lang=EN-US> Light-weight 100% organically grown **cotton** construction. **Shi**elds against sun and precipitation. Metallic ions in the fibers inhibit bacterial growth, and help neutralize odor. </body></html>

Searching across multiple columns

You can perform a full text search across multiple columns in a single query, as long as the columns are part of the same text index.

```
SELECT *
   FROM t
      WHERE CONTAINS ( t.c1, t.c2, 'term1|term2' );

SELECT *
   FROM t
   WHERE CONTAINS( t.c1, 'term1' )
      OR CONTAINS( t.c2, 'term2' );
```

The first query matches if *t1.c1* contains *term1*, or if *t1.c2* contains *term2*.

The second query matches if either *t1.c1* or *t1.c2* contains either *term1* or *term2*. Using the contains in this manner also returns scores for the matches. See "Scoring full text search results" on page 290.

Fuzzy searching

Fuzzy searching can be used to search for misspellings or variations of a word. To do so, use the FUZZY operator followed by a string in double quotes to find an approximate match for the string. For example, `CONTAINS ( Products.Description, 'FUZZY "cotton"' )` returns **cotton** and misspellings such as **coton** or **cotten**.

Note
You can only perform fuzzy searches on text indexes built using the NGRAM term breaker. For more information about the NGRAM term breaker and how it applies to fuzzy searches, see "Text configuration object settings" on page 323.

Using the FUZZY operator is equivalent to breaking the string manually into substrings of length *n* and separating them with OR operators. For example, suppose you have a text index configured with the NGRAM term breaker and a MAXIMUM TERM LENGTH of 3. Specifying `'FUZZY "500 main street"'` is equivalent to specifying `'500 OR mai OR ain OR str OR tre OR ree OR eet'`.

The FUZZY operator is useful in a full text search that returns a score. This is because many approximate matches may be returned, but usually only the matches with the highest scores are meaningful.

View searching

To use a view in a full text search, you must build a text index on the required columns in the base table. For example, suppose you create a text index on the Employees.Address column called EmployeeAddressTxtIdx. Then, you create a view on the Employees table called MyEmployeesView. Using a statement similar to the following, you can query the view using the text index on the underlying table.

```
SELECT COUNT(*) FROM MyEmployeesView WHERE CONTAINS( EmployeeAddressTxtIdx,
'Avenue' );
```

Searching a view using a text index on the underlying base table is restricted as follows:

- The view cannot contain a TOP, FIRST, DISTINCT, GROUP BY, ORDER BY, UNION, INTERSECT, EXCEPT clause, or window function.
- A CONTAINS query can refer to a base table inside a view, but not to a base table inside a view inside another view.

Tutorial: Performing a full text search on a GENERIC text index

Use the following procedure to perform a full text search on a text index that uses a GENERIC term breaker.

See also: "Tutorial: Performing a fuzzy full text search" on page 311.

Perform a full text search on a GENERIC text index

1. Start Interactive SQL and connect to the sample database using the SQL Anywhere 12 Demo data source.
2. Create the text configuration object.

   The following example creates a text configuration object called myTxtConfig. You must include the FROM clause to specify the text configuration object to use as a template.

   ```
   CREATE TEXT CONFIGURATION myTxtConfig FROM default_char;
   ```

3. Customize the text configuration object.

Add a stoplist containing the words because, about, therefore, and only. Then, set the maximum term length to 30. You must do this in separate ALTER TEXT CONFIGURATION statements, as follows:

```
ALTER TEXT CONFIGURATION myTxtConfig
   STOPLIST 'because about therefore only';
ALTER TEXT CONFIGURATION myTxtConfig
   MAXIMUM TERM LENGTH 30;
```

4. Start Sybase Central and connect to the sample database using the SQL Anywhere 12 Demo data source.
5. Create a copy of the MarketingInformation table.
   a. Expand the **Tables** folder.
   b. Right-click **MarketingInformation** and choose **Copy**.
   c. Right-click the **Tables** folder and choose **Paste**.
   d. In the **Name** field, type **MarketingInformation1**.
   e. Click **OK**.
6. In Interactive SQL, execute the following command to populate the new table with data:

```
INSERT INTO MarketingInformation1
   SELECT * FROM MarketingInformation;
```

7. On the Description column of the MarketingInformation1 table in the sample database, create a text index that references the myTxtConfig text configuration object. Set the refresh interval to 24 hours.

```
CREATE TEXT INDEX myTxtIndex ON MarketingInformation1 ( Description )
   CONFIGURATION myTxtConfig
   AUTO REFRESH EVERY 24 HOURS;
```

8. Execute the following statement to refresh the text index:

```
REFRESH TEXT INDEX myTxtIndex ON MarketingInformation1;
```

9. Execute the following statements to test the text index.
   a. This statement searches the text index for the terms **cotton** or **cap**. The results are sorted by score in descending order. **Cap** has a higher score than **cotton** because **cap** occurs less frequently in the text index.

```
SELECT ID, Description, ct.*
   FROM MarketingInformation1
     CONTAINS ( Description, 'cotton | cap' ) ct
   ORDER BY score DESC;
```


ID	Description	Score
905	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Baseball **Cap**</title></head><body lang=EN-US> A lightweight wool **cap** with mesh side vents for breathable comfort during aerobic activities. Moisture-absorbing headband liner. </body></html>	2.2742084275032632
904	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Baseball **Cap**</title></head><body lang=EN-US> This fashionable hat is ideal for glacier travel, sea-kayaking, and hiking. With concealed draw cord for windy days. </body></html>	1.6980426550094467
908	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Sweatshirt</title></head><body lang=EN-US> Lightweight 100% organically grown **cotton** hooded sweatshirt with taped neck seams. Comes pre-washed for softness and to lessen shrinkage. </body></html>	0.9461597363521859
910	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Shorts</title></head><body lang=EN-US> These quick-drying **cotton** shorts provide all day comfort on or off the trails. Now with a more comfortable and stretchy fabric and an adjustable drawstring waist. </body></html>	0.9244136988525732

ID	Description	Score
906	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Visor</title></head><body lang=EN-US> Lightweight 100% organically grown **cotton** construction. Shields against sun and precipitation. Metallic ions in the fibers inhibit bacterial growth, and help neutralize odor. </body></html>	0.9134171046194403
909	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Sweatshirt</title></head><body lang=EN-US> Top-notch construction includes durable topstitched seams for strength with low-bulk, resilient rib-knit collar, cuffs and bottom. An 80% **cotton**/20% polyester blend makes it easy to keep them clean. </body></html>	0.8856420222728282

b. The following statement searches the text index for the term cotton. Rows that also contain the word visor are discarded. The results are not scored because the clause CONTAINS uses a predicate.

```
SELECT ID, Description
   FROM MarketingInformation1
   WHERE CONTAINS( Description, 'cotton -visor' );
```

ID	Description
908	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Sweatshirt</title></head><body lang=EN-US> Lightweight 100% organically grown **cotton** hooded sweatshirt with taped neck seams. Comes pre-washed for softness and to lessen shrinkage. </body></html>
909	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Sweatshirt</title></head><body lang=EN-US> Top-notch construction includes durable topstitched seams for strength with low-bulk, resilient rib-knit collar, cuffs and bottom. An 80% **cotton**/20% polyester blend makes it easy to keep them clean. </body></html>

ID	Description
910	<html><head><meta http-equiv=Content-Type content="text/ html; charset=windows-1252"><title>Shorts</title></ head><body lang=EN-US> These quick-drying **cotton** shorts provide all day comfort on or off the trails. Now with a more comfortable and stretchy fabric and an adjustable drawstring waist.</ span> </body></html>

c. The following statement tests each row for the term **cotton**. If the row contains the term, a 1 appears in the Results column; otherwise, a 0 is returned.

```
SELECT ID, Description, IF CONTAINS ( Description, 'cotton' )
      THEN 1
      ELSE 0
      ENDIF AS Results
   FROM MarketingInformation1;
```

ID	Description	Results
901	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Tee Shirt</title></head><body lang=EN-US> We've improved the design of this perennial favorite. A sleek and technical shirt built for the trail, track, or sidewalk. UPF rating of 50+. </ body></html>	0
902	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Tee Shirt</title></head><body lang=EN-US> This simple, sleek, and lightweight technical shirt is designed for high-intensity workouts in hot and humid weather. The recycled polyester fabric is gentle on the earth and soft against your skin. </ body></html>	0
903	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Tee Shirt</title></head><body lang=EN-US> A sporty, casual shirt made of recycled water bottles. It will serve you equally well on trails or around town. The fabric has a wicking finish to pull perspiration away from your skin. </body></html>	0

ID	Description	Results
904	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Baseball Cap</title></head><body lang=EN-US> This fashionable hat is ideal for glacier travel, sea-kayaking, and hiking. With concealed draw cord for windy days. </body></html>	0
905	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Baseball Cap</title></head><body lang=EN-US> A lightweight wool cap with mesh side vents for breathable comfort during aerobic activities. Moisture-absorbing headband liner. </body></html>	0
906	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Visor</title></head><body lang=EN-US> Lightweight 100% organically grown **cotton** construction. Shields against sun and precipitation. Metallic ions in the fibers inhibit bacterial growth, and help neutralize odor. </body></html>	1
907	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Visor</title></head><body lang=EN-US> A polycarbonate visor with an abrasion-resistant coating on the outside. Great for jogging in the spring, summer, and early fall. The elastic headband has plenty of stretch to give you a snug yet comfortable fit every time you wear it. </body></html>	0
908	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Sweatshirt</title></head><body lang=EN-US> Lightweight 100% organically grown **cotton** hooded sweatshirt with taped neck seams. Comes pre-washed for softness and to lessen shrinkage. </body></html>	1

ID	Description	Results
909	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Sweatshirt</title></head><body lang=EN-US> Top-notch construction includes durable topstitched seams for strength with low-bulk, resilient rib-knit collar, cuffs and bottom. An 80% **cotton**/20% polyester blend makes it easy to keep them clean. </body></html>	1
910	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Shorts</title></head><body lang=EN-US> These quick-drying **cotton** shorts provide all day comfort on or off the trails. Now with a more comfortable and stretchy fabric and an adjustable drawstring waist. </body></html>	1

10. Disconnect from Interactive SQL and Sybase Central.

11. (optional) Restore the sample database (*demo.db*) to its original state by following the steps found here: "Recreate the sample database (demo.db)" [*SQL Anywhere 12 - Introduction*].

See also

- "How to manage text configuration objects" on page 323
- "CREATE TEXT CONFIGURATION statement" [*SQL Anywhere Server - SQL Reference*]
- "ALTER TEXT CONFIGURATION statement" [*SQL Anywhere Server - SQL Reference*]
- "How to manage text indexes" on page 339
- "CREATE TEXT INDEX statement" [*SQL Anywhere Server - SQL Reference*]
- "ALTER TEXT INDEX statement" [*SQL Anywhere Server - SQL Reference*]

Tutorial: Performing a fuzzy full text search

Use the following procedure to perform a fuzzy full text search on a text index that uses an NGRAM term breaker.

See also: "Tutorial: Performing a full text search on a GENERIC text index" on page 305.

Perform a fuzzy full text search on an NGRAM term index

1. Start Interactive SQL and connect to the sample database using the SQL Anywhere 12 Demo data source.

2. Execute the following statement to create a text configuration object called myFuzzyTextConfig.

```
CREATE TEXT CONFIGURATION myFuzzyTextConfig FROM default_char;
```

3. Execute the following statement to change the term breaker to NGRAM and set the maximum term length to 3. Fuzzy searches are performed using n-grams. Separate ALTER TEXT CONFIGURATION statements are used to implement these changes:

```
ALTER TEXT CONFIGURATION myFuzzyTextConfig
   TERM BREAKER NGRAM;
ALTER TEXT CONFIGURATION myFuzzyTextConfig
   MAXIMUM TERM LENGTH 3;
```

4. Start Sybase Central and connect to the sample database using the SQL Anywhere 12 Demo data source.

5. Create a copy of the MarketingInformation table.

   a. In Sybase Central, expand the **Tables** folder.

   b. Right-click **MarketingInformation** and choose **Copy**.

   c. Right-click the **Tables** folder and choose **Paste**.

   d. In the **Name** field, type **MarketingInformation2**. Click **OK**.

6. In Interactive SQL, execute the following command to add data to the MarketingInformation2 table:

```
INSERT INTO MarketingInformation2
   SELECT * FROM MarketingInformation;
```

7. Execute the following command to create a text index on the MarketingInformation2.Description column that references the myFuzzyTextConfig text configuration object:

```
CREATE TEXT INDEX myFuzzyTextIdx ON MarketingInformation2 ( Description )
   CONFIGURATION myFuzzyTextConfig;
```

8. Execute the following statement to check for terms similar to **coten**:

```
SELECT MarketingInformation2.Description, ct.*
    FROM MarketingInformation2 CONTAINS
( MarketingInformation2.Description, 'FUZZY "coten"' ) ct
    ORDER BY ct.score DESC;
```

Description	Score
<html><head><meta http-equiv=**Content**-Type **content**="text/html; charset=windows-1252"><title>Sweatshirt</title></head><body lang=EN-US> Lightweight 100% organically grown **cotton** hooded sweatshirt with taped neck seams. Comes pre-washed for softness and to lessen shrinkage. </body></html>	0.9461597363521859

Description	Score
<html><head><meta http-equiv=**Content**-Type **content**="text/html; charset=windows-1252"><title>Shorts</title></head><body lang=EN-US> These quick-drying **cotton** shorts provide all day comfort on or off the trails. Now with a more comfortable and stretchy fabric and an adjustable drawstring waist. </body></html>	0.9244136988525732
<html><head><meta http-equiv=**Content**-Type **content**="text/html; charset=windows-1252"><title>Visor</title></head><body lang=EN-US> Lightweight 100% organically grown **cotton** construction. Shields against sun and precipitation. Metallic ions in the fibers inhibit bacterial growth, and help neutralize odor. </body></html>	0.9134171046194403
<html><head><meta http-equiv=**Content**-Type **content**="text/html; charset=windows-1252"><title>Sweatshirt</title></head><body lang=EN-US> Top-notch construction includes durable topstitched seams for strength with low-bulk, resilient rib-knit collar, cuffs and bottom. An 80% **cotton**/20% polyester blend makes it easy to keep them clean. </body></html>	0.8856420222728282
<html><head><meta http-equiv=**Content**-Type **content**="text/html; charset=windows-1252"><title>Baseball Cap</title></head><body lang=EN-US> This fashionable hat is ideal for glacier travel, sea-kayaking, and hiking. With concealed draw cord for windy days. </body></html>	0
<html><head><meta http-equiv=**Content**-Type **content**="text/html; charset=windows-1252"><title>Baseball Cap</title></head><body lang=EN-US> A lightweight wool cap with mesh side vents for breathable comfort during aerobic activities. Moisture-absorbing headband liner. </body></html>	0

Description	Score
<html><head><meta http-equiv=**Content**-Type **content**="text/html; charset=windows-1252"><title>Tee Shirt</title></head><body lang=EN-US> We've improved the design of this perennial favorite. A sleek and technical shirt built for the trail, track, or sidewalk. UPF rating of 50+. </body></html>	0
<html><head><meta http-equiv=**Content**-Type **content**="text/html; charset=windows-1252"><title>Tee Shirt</title></head><body lang=EN-US> A sporty, casual shirt made of recycled water bottles. It will serve you equally well on trails or around town. The fabric has a wicking finish to pull perspiration away from your skin. </body></html>	0
<html><head><meta http-equiv=**Content**-Type **content**="text/html; charset=windows-1252"><title>Tee Shirt</title></head><body lang=EN-US> This simple, sleek, and lightweight technical shirt is designed for high-intensity workouts in hot and humid weather. The recycled polyester fabric is gentle on the earth and soft against your skin. </body></html>	0
<html><head><meta http-equiv=**Content**-Type **content**="text/html; charset=windows-1252"><title>Visor</title></head><body lang=EN-US> A polycarbonate visor with an abrasion-resistant coating on the outside. Great for jogging in the spring, summer, and early fall. The elastic headband has plenty of stretch to give you a snug yet comfortable fit every time you wear it. </body></html>	0

Note
The last six rows have terms that contain matching n-grams. However, no scores are assigned to them because all rows in the table contain these terms.

9. Disconnect from Interactive SQL and Sybase Central.

10. (optional) Restore the sample database (*demo.db*) to its original state by following the steps found here: "Recreate the sample database (demo.db)" [*SQL Anywhere 12 - Introduction*].

See also

- "How to manage text configuration objects" on page 323
- "CREATE TEXT CONFIGURATION statement" [*SQL Anywhere Server - SQL Reference*]
- "ALTER TEXT CONFIGURATION statement" [*SQL Anywhere Server - SQL Reference*]
- "How to manage text indexes" on page 339
- "CREATE TEXT INDEX statement" [*SQL Anywhere Server - SQL Reference*]
- "ALTER TEXT INDEX statement" [*SQL Anywhere Server - SQL Reference*]
- "Scoring full text search results" on page 290

Tutorial: Performing a full text search on an NGRAM text index

Use the following procedure to perform a full text search on a text index that uses an NGRAM term breaker. This procedure can also be used to create a full text search of Chinese, Japanese, or Korean data.

In databases with multibyte character sets, some punctuation and space characters such as full width commas and full width spaces may be treated as alphanumeric characters.

See also: "Tutorial: Performing a fuzzy full text search" on page 311.

Perform a full text search on an NGRAM text index

1. Start Interactive SQL and connect to the sample database using the SQL Anywhere 12 Demo data source.

2. Execute the following statement to create an NCHAR text configuration object named myNcharNGRAMTextConfig:

   ```
   CREATE TEXT CONFIGURATION myNcharNGRAMTextConfig FROM default_nchar;
   ```

3. Execute the following statements to change the TERM BREAKER algorithm to NGRAM, and to set MAXIMUM TERM LENGTH (N) to 2.

   ```
   ALTER TEXT CONFIGURATION myNcharNGRAMTextConfig
      TERM BREAKER NGRAM;

   ALTER TEXT CONFIGURATION myNcharNGRAMTextConfig
      MAXIMUM TERM LENGTH 2;
   ```

   For Chinese, Japanese, and Korean data, the recommended value for N is 2 or 3. For searches limited to one or two characters, set the N value to 1. Setting the N value to 1 can cause slower execution of long queries.

4. Start Sybase Central and connect to the sample database using the SQL Anywhere 12 Demo data source.

5. Create a copy of the MarketingInformation table.

   a. Expand the **Tables** folder.

b. Right-click **MarketingInformation** and choose **Copy**.

c. Right-click the **Tables** folder and choose **Paste**.

d. In the **Name** field, type **MarketingInformationNgram**.

e. Click **OK**.

6. In Interactive SQL, execute the following statement to add data to the MarketingInformationNgram table:

```
INSERT INTO MarketingInformationNgram
   SELECT *
      FROM MarketingInformation;
COMMIT;
```

7. Execute the following statement to create an IMMEDIATE REFRESH text index on the MarketingInformationNgram.Description column using the myNcharNGRAMTextConfig text configuration object:

```
CREATE TEXT INDEX ncharNGRAMTextIndex
   ON MarketingInformationNgram( Description )
      CONFIGURATION myNcharNGRAMTextConfig;
```

8. Execute the following statements to test the text index.

a. The following statement searches the 2-GRAM text index for terms containing **sw**. The results are sorted by score in descending order.

```
SELECT M.Description, ct.*
   FROM MarketingInformationNgram AS M
CONTAINS( M.Description, 'sw' ) ct
   ORDER BY ct.score DESC;
```

Description	Score
<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>**Sw**eatshirt</title></head><body lang=EN-US> Lightweight 100% organically grown cotton hooded **Sw**eatshirt with taped neck seams. Comes pre-washed for softness and to lessen shrinkage. </body></html>	2.262071918398649
<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>**Sw**eatshirt</title></head><body lang=EN-US> Top-notch construction includes durable topstitched seams for strength with low-bulk, resilient rib-knit collar, cuffs and bottom. An 80% cotton/20% polyester blend makes it easy to keep them clean. </body></html>	1.5556043490424176

b. The following statement searches for terms containing **ams**. The results are sorted by score in descending order.

```
SELECT M.Description, ct.*
   FROM MarketingInformationNgram AS M
CONTAINS( M.Description, 'ams' ) ct
   ORDER BY ct.score DESC;
```

With the 2-GRAM text index, the previous statement is semantically equivalent to:

```
SELECT M.Description, ct.*
   FROM MarketingInformationNgram AS M
CONTAINS( M.Description, '"am ms"' ) ct
   ORDER BY ct.score DESC;
```

Description	Score
<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Sweatshirt</title></head><body lang=EN-US> Lightweight 100% organically grown cotton hooded sweatshirt with taped neck se**ams**. Comes pre-washed for softness and to lessen shrinkage. </body></html>	1.6619019465461564
<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Sweatshirt</title></head><body lang=EN-US> Top-notch construction includes durable topstitched se**ams** for strength with low-bulk, resilient rib-knit collar, cuffs and bottom. An 80% cotton/20% polyester blend makes it easy to keep them clean. </body></html>	1.5556043490424176

c. The following statement searches for terms with **v** followed by any alphanumeric character. Because **ve** occurs more frequently in the indexed data, rows that contain the 2-gram **ve** are assigned a lower score than rows containing **vi**. The results are sorted by score in descending order.

```
SELECT M.ID, M.Description, ct.*
   FROM MarketingInformationNgram AS M
CONTAINS( M.Description, 'v*' ) ct
   ORDER BY ct.score DESC;
```

ID	Description	Score
901	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Tee Shirt</title></head><body lang=EN-US> We'**ve** im-pro**ve**d the design of this perennial fav**vo**rite. A sleek and technical shirt built for the trail, track, or sidewalk. UPF rating of 50+. </body></html>	3.3416789108071976
907	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>**Vi**sor</title></head><body lang=EN-US> A polycarbonate **vi**sor with an abrasion-resistant coating on the outside. Great for jogging in the spring, summer, and early fall. The elastic headband has plenty of stretch to gi**ve** you a snug yet comfortable fit e**ve**ry time you wear it. </body></html>	2.1123084896159376
905	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Baseball Cap</title></head><body lang=EN-US> A lightweight wool cap with mesh side **ve**nts for breathable comfort during aerobic activ**i**ties. Moisture-absorbing headband liner. </body></html>	1.6750365447462499
910	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Shorts</title></head><body lang=EN-US> These quick-drying cotton shorts pro**vi**de all day comfort on or off the trails. Now with a more comfortable and stretchy fabric and an adjustable drawstring waist. </body></html>	0.9244136988525732

ID	Description	Score
906	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>**Vi**sor</title></head><body lang=EN-US> Lightweight 100% organically grown cotton construction. Shields against sun and precipitation. Metallic ions in the fibers inhibit bacterial growth, and help neutralize odor. </body></html>	0.9134171046194403
904	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Baseball Cap</title></head><body lang=EN-US> This fashionable hat is ideal for glacier tra**ve**l, sea-kayaking, and hiking. With concealed draw cord for windy days. </body></html>	0.7313071661212746
903	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Tee Shirt</title></head><body lang=EN-US> A sporty, casual shirt made of recycled water bottles. It will ser**ve** you equally well on trails or around town. The fabric has a wicking finish to pull perspiration away from your skin. </body></html>	0.6799436746197272

d. The following statements search each row for any terms containing v. After the second statement, the variable contains the string `av OR ev OR iv OR ov OR rv OR ve OR vi OR vo`. The results are sorted by score in descending order. When an n-gram appears in all indexed rows, it is assigned a score of zero.

   This is the only method that allows a single character to be located if it appears before a whitespace or a non-alphanumeric character.

```
CREATE VARIABLE query NVARCHAR (100);
SELECT LIST (term, ' OR ' )
INTO query
   FROM sa_text_index_vocab( 'ncharNGRAMTextIndex',
'MarketingInformationNgram', 'dba' )
   WHERE term LIKE '%v%';
SELECT M.ID, M.Description, ct.*
   FROM MarketingInformationNgram AS M
```

```
CONTAINS( M.Description, query ) ct
ORDER BY ct.score DESC;
```

ID	Description	Score
901	<html><head><meta http-equ**iv**=Content-Type content="text/html; charset=windows-1252"><title>Tee Shirt</title></head><body lang=EN-US> We'**ve** impr**ove**d the design of this perennial f**avo**rite. A sleek and technical shirt built for the trail, track, or sidewalk. UPF rating of 50+. </body></html>	6.654350268810443
907	<html><head><meta http-equ**iv**=Content-Type content="text/html; charset=windows-1252"><title>**Vi**sor</title></head><body lang=EN-US> A polycarbonate **vi**sor with an abrasion-resistant coating on the outside. Great for jogging in the spring, summer, and early fall. The elastic headband has plenty of stretch to g**ive** you a snug yet comfortable fit **eve**ry time you wear it. </body></html>	4.265623837817126
903	<html><head><meta http-equ**iv**=Content-Type content="text/html; charset=windows-1252"><title>Tee Shirt</title></head><body lang=EN-US> A sporty, casual shirt made of recycled water bottles. It will se**rve** you equally well on trails or around town. The fabric has a wicking finish to pull perspiration away from your skin. </body></html>	2.9386676702799504
910	<html><head><meta http-equ**iv**=Content-Type content="text/html; charset=windows-1252"><title>Shorts</title></head><body lang=EN-US> These quick-drying cotton shorts pr**ovi**de all day comfort on or off the trails. Now with a more comfortable and stretchy fabric and an adjustable drawstring waist. </body></html>	2.5481193655722336

ID	Description	Score
904	<html><head><meta http-equ**iv**=Content-Type content="text/html; charset=windows-1252"><title>Baseball Cap</title></head><body lang=EN-US> This fashionable hat is ideal for glacier tr**ave**l, sea-kayaking, and hiking. With concealed draw cord for windy days. </body></html>	2.4293498211307214
905	<html><head><meta http-equ**iv**=Content-Type content="text/html; charset=windows-1252"><title>Baseball Cap</title></head><body lang=EN-US> A lightweight wool cap with mesh side **ve**nts for breathable comfort during aerobic act**ivi**ties. Moisture-absorbing headband liner. </body></html>	1.6750365447462499
906	<html><head><meta http-equ**iv**=Content-Type content="text/html; charset=windows-1252"><title>**Vi**sor</title></head><body lang=EN-US> Lightweight 100% organically grown cotton construction. Shields against sun and precipitation. Metallic ions in the fibers inhibit bacterial growth, and help neutralize odor. </body></html>	0.9134171046194403
902	<html><head><meta http-equ**iv**=Content-Type content="text/html; charset=windows-1252"><title>Tee Shirt</title></head><body lang=EN-US> This simple, sleek, and lightweight technical shirt is designed for high-intensity workouts in hot and humid weather. The recycled polyester fabric is gentle on the earth and soft against your skin. </body></html>	0

ID	Description	Score
908	<html><head><meta http-equ**iv**=Content-Type content="text/html; charset=windows-1252"><title>Sweatshirt</title></head><body lang=EN-US> Lightweight 100% organically grown cotton hooded sweatshirt with taped neck seams. Comes pre-washed for softness and to lessen shrinkage. </body></html>	0
909	<html><head><meta http-equ**iv**=Content-Type content="text/html; charset=windows-1252"><title>Sweatshirt</title></head><body lang=EN-US> Top-notch construction includes durable topstitched seams for strength with low-bulk, resilient rib-knit collar, cuffs and bottom. An 80% cotton/20% polyester blend makes it easy to keep them clean. </body></html>	0

e. The following statement searches the Description column for rows that contain **ea**, **ka**, and **ki**.

```
SELECT M.ID, M.Description, ct.*
   FROM MarketingInformationNgram AS M
CONTAINS( M.Description, 'ea ka ki' ) ct
  ORDER BY ct.score DESC;
```

ID	Description	Score
904	<html><h**ea**d><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Baseball Cap</title></h**ea**d><body lang=EN-US> This fashionable hat is id**ea**l for glacier travel, s**ea**-**ka**yaking, and hi**ki**ng. With conc**ea**led draw cord for windy days. </body></html>	3.4151032739119733

f. The following statement searches the Description column for rows that contain **ve** and **vi**, but not **gg**.

```
SELECT M.ID, M.Description, ct.*
   FROM MarketingInformationNgram AS M
CONTAINS( M.Description, 've & vi -gg' ) ct
   ORDER BY ct.score DESC;
```


ID	Description	Score
905	<html><head><meta http-equiv=Content-Type content="text/html; charset=windows-1252"><title>Baseball Cap</title></head><body lang=EN-US> A lightweight wool cap with mesh side **ve**nts for breathable comfort during aerobic activ**i**ties. Moisture-absorbing headband liner. </body></html>	1.6750365447462499

9. Disconnect from Interactive SQL and Sybase Central.
10. (optional) Restore the sample database (*demo.db*) to its original state by following the steps found here: "Recreate the sample database (demo.db)" [*SQL Anywhere 12 - Introduction*].

How to manage text configuration objects

A text configuration object controls what terms go into a text index when it is built or refreshed, and how a full text query is interpreted. The settings for each text configuration object are stored as a row in the ISYSTEXTCONFIG system table.

When the database server creates or refreshes a text index, it uses the settings for the text configuration object specified when the text index was created. If you did not specify a text configuration object when creating the text index, the database server chooses one of the default text configuration objects, based on the type of data in the columns being indexed. SQL Anywhere provides two default text configuration objects. See "Example text configuration objects" on page 331.

To view settings for existing text configuration objects, query the SYSTEXTCONFIG system view. See "SYSTEXTCONFIG system view" [*SQL Anywhere Server - SQL Reference*].

Text configuration object settings

SQL Anywhere provides two default text configuration objects, default_char for use with CHAR data, and default_nchar for use with NCHAR and CHAR data. Note that while default_nchar can be used with any data, character set conversion will be performed. For information about the settings for default_char and default_nchar text configuration objects, see "Default text configuration objects" on page 332.

You can test how a text configuration object affects term breaking using the sa_char_terms and sa_nchar_terms system procedures. See "sa_char_terms system procedure" [*SQL Anywhere Server - SQL Reference*], and "sa_nchar_terms system procedure" [*SQL Anywhere Server - SQL Reference*].

The following table explains text configuration object settings and how they impact what is indexed and how a full text search query is interpreted for text indexes using the text configuration. For examples, see "Example text configuration objects" on page 331.

- **External prefilter algorithm (PREFILTER)** **Prefiltering** is the process of extracting text data from a file types such as Word, PDF, HTML, and XML. In the context of text indexing, prefiltering allows you to extract only the data you want indexed, and avoid indexing unnecessary content such HTML tags. For certain types of documents (for example, Microsoft Word documents), prefiltering is required to make full text indexes useful.

  SQL Anywhere does not provide a built-in prefilter feature. However, you can create an external prefilter library to perform prefiltering according to your requirements, and then alter your text configuration object to point to it.

  The following table explains the impact that the value of PREFILTER EXTERNAL NAME has on text indexing and on how query strings are handled:

Text indexes	Query strings
GENERIC and NGRAM text indexes An external prefilter takes an input value (a document) and filters it according to the rules specified by the prefilter library. The resulting text is then passed to the term breaker before building or updating the text index.	**GENERIC and NGRAM text indexes** Query strings are not passed through a prefilter, so the setting of the PREFILTER EXTERNAL NAME clause has no impact on query strings.

  See also: "External prefilter libraries" on page 348, and "PREFILTER EXTERNAL NAME clause, ALTER TEXT CONFIGURATION statement" [*SQL Anywhere Server - SQL Reference*].

  The *ExternalLibrariesFullText* directory in your SQL Anywhere install contains prefilter and term breaker sample code for you to explore. This directory is found under your *Samples* directory. For the location of your *Samples* directory, see "Samples directory" [*SQL Anywhere Server - Database Administration*].

- **Term breaker algorithm (TERM BREAKER)** The TERM BREAKER setting specifies the algorithm to use for breaking strings into terms. The choices are GENERIC for storing terms, or NGRAM for storing n-grams. For GENERIC, you can use the built-in term breaker algorithm, or an external term breaker. See "TERM BREAKER clause, ALTER TEXT CONFIGURATION statement" [*SQL Anywhere Server - SQL Reference*].

  The following table explains the impact that the value of TERM BREAKER has on text indexing and on how query strings are handled:

Text indexes	Query strings
○ **GENERIC text index** Performance of GENERIC text indexes can be faster than NGRAM text indexes. However, you cannot perform fuzzy searches on GENERIC text indexes. When building a GENERIC text index using the built-in algorithm, groups of alphanumeric characters appearing between non-alphanumeric characters are processed as terms by the database server, and have positions assigned to them. When building a GENERIC text index using a term breaker external library, terms and their positions are defined by the external library. Once the terms have been identified by the term breaker, any term that exceeds the term length restrictions or that is found in the stoplist, is counted but not inserted in the text index. ○ **NGRAM text index** An **n-gram** is a group of characters of length *n* where *n* is the value of MAXIMUM TERM LENGTH. When building an NGRAM text index, the database server treats as a term any group of alphanumeric characters between non-alphanumeric characters. Once the terms are defined, the database server breaks the terms into n-grams. In doing so, terms shorter than *n*, and n-grams that are in the stoplist, are discarded. For example, for an NGRAM text index with MAXIMUM TERM LENGTH 3, the string 'my red table' is represented in the text index as the following n-grams: red tab abl ble. For n-grams, the positional information of the n-grams is stored, not the positional information for the original terms.	When parsing a CONTAINS query, the database server extracts keywords and special characters from the query string and then applies the term breaker algorithm to the remaining terms. For example, if the query string is `'ab_cd* AND b*'`, the * and the keyword AND are extracted, and the character strings ab_cd and b are given to the term breaker algorithm to parse separately. For more information about keywords and special characters in full text search, see "CONTAINS search condition" [*SQL Anywhere Server - SQL Reference*]. ○ **GENERIC text index** When querying a GENERIC text index, terms in the query string are processed in the same manner as if they were being indexed. Matching is performed by comparing query terms to terms in the text index. ○ **NGRAM text index** When querying an NGRAM text index, terms in the query string are processed in the same manner as if they were being indexed. Matching is performed by comparing n-grams from the query terms to n-grams from the indexed terms. For information about how prefix searching is performed in NGRAM text indexes, see "Prefix searching" on page 297.

If not defined, the default for TERM BREAKER is taken from the setting in the default text configuration object. If a term breaker is not defined in the default text configuration object, the internal term breaker is used. See "Default text configuration objects" on page 332.

See also: "TERM BREAKER clause, ALTER TEXT CONFIGURATION statement" [*SQL Anywhere Server - SQL Reference*].

- **Minimum term length setting (MINIMUM TERM LENGTH)** The MINIMUM TERM LENGTH setting specifies the minimum length, in characters, for terms inserted in the index or searched for in a full text query. MINIMUM TERM LENGTH is not relevant for NGRAM text indexes.

  MINIMUM TERM LENGTH has special implications on prefix searching. See "Prefix searching" on page 297.

  The value of MINIMUM TERM LENGTH must be greater than 0. If you set it higher than MAXIMUM TERM LENGTH, then MAXIMUM TERM LENGTH is automatically adjusted to be equal to MINIMUM TERM LENGTH.

  If not defined, the default for MINIMUM TERM LENGTH is taken from the setting in the default text configuration object, which is typically 1. See "Default text configuration objects" on page 332.

  The following table explains the impact that the value of MINIMUM TERM LENGTH has on text indexing and on how query strings are handled:

Text indexes	Query strings
GENERIC text index For GENERIC text indexes, the text index will not contain words shorter than MINIMUM TERM LENGTH. **NGRAM text index** For NGRAM text indexes, this setting is ignored.	**GENERIC text index** When querying a GENERIC text index, query terms shorter than MINIMUM TERM LENGTH are ignored because they cannot exist in the text index. **NGRAM text index** The MINIMUM TERM LENGTH setting has no impact on full text queries on NGRAM text indexes.

  See also: "MINIMUM TERM LENGTH clause, ALTER TEXT CONFIGURATION statement" [*SQL Anywhere Server - SQL Reference*].

- **Maximum term length setting (MAXIMUM TERM LENGTH)** The MAXIMUM TERM LENGTH setting is used differently depending on the term breaker algorithm.

  The value of MAXIMUM TERM LENGTH must be less than or equal to 60. If you set it lower than the MINIMUM TERM LENGTH, then MINIMUM TERM LENGTH is automatically adjusted to be equal to MAXIMUM TERM LENGTH.

  The default for this setting is taken from the setting in the default text configuration object, which is typically 20. See "Default text configuration objects" on page 332.

  If not defined, the default for MAXIMUM TERM LENGTH is taken from the setting in the default text configuration object, which is typically 20. See "Default text configuration objects" on page 332.

The following table explains the impact that the value of MAXIMUM TERM LENGTH has on text indexing and on how query strings are handled:

Text indexes	Query strings
GENERIC text indexes For GENERIC text indexes, MAXIMUM TERM LENGTH specifies the maximum length, in characters, for terms inserted in the text index. **NGRAM text index** For NGRAM text indexes, MAXIMUM TERM LENGTH determines the length of the n-grams that terms are broken into. An appropriate choice of length for n-grams depends on the language. Typical values are 4 or 5 characters for English, and 2 or 3 characters for Chinese.	**GENERIC text indexes** For GENERIC text indexes, query terms longer than MAXIMUM TERM LENGTH are ignored because they cannot exist in the text index. **NGRAM text index** For NGRAM text indexes, query terms are broken into n-grams of length *n*, where *n* is the same as MAXIMUM TERM LENGTH. Then, the database server uses the n-grams to search the text index. Terms shorter than MAXIMUM TERM LENGTH are ignored because they will not match the n-grams in the text index. Therefore, proximity searches do not work unless arguments are prefixes of length *n*.

See also: "MAXIMUM TERM LENGTH clause, ALTER TEXT CONFIGURATION statement" [*SQL Anywhere Server - SQL Reference*].

- **Stoplist setting (STOPLIST)** The stoplist setting specifies the terms that must not be indexed.

  If not defined, the default for this setting is taken from the setting in the default text configuration object, which typically has an empty stoplist. See "Default text configuration objects" on page 332.

STOPLIST impact to text index	STOPLIST impact to query terms
GENERIC text indexes For GENERIC text indexes, terms that are in the stoplist are not inserted into the text index. **NGRAM text index** For NGRAM text indexes, the text index will not contain the n-grams formed from the terms in the stoplist.	**GENERIC text indexes** For GENERIC text indexes, query terms that are in the stoplist are ignored because they cannot exist in the text index. **NGRAM text index** Terms in the stoplist are broken into n-grams and the n-grams are used for the term filtering. Likewise, query terms are broken into n-grams and any that match n-grams in the stoplist are dropped because they cannot exist in the text index.

The settings in the text configuration object are applied to the stoplist when it is parsed. That is, the specified specified term breaker and the min/max length settings are applied.

Stoplists in NGRAM text indexes can cause unexpected results because the stoplist is stored in n-gram form, and not the stoplist terms you specified. For example, in an NGRAM text index where MAXIMUM TERM LENGTH is 3, if you specify `STOPLIST 'there'`, the following n-grams are stored as the stoplist: the her ere. This impacts the ability to query for any terms that contain the n-grams the, her, and ere.

Note
The same restrictions with regards to specifying string literals also apply to stoplists. For example, apostrophes must be escaped, and so on. For more information on formatting string literals, see "String literals" [*SQL Anywhere Server - SQL Reference*].

The Samples directory contains sample code that loads stoplists for several languages. These sample stoplists are recommended for use only on GENERIC text indexes. For the location of the Samples directory, see "Samples directory" [*SQL Anywhere Server - Database Administration*].

See also: "STOPLIST clause, ALTER TEXT CONFIGURATION statement" [*SQL Anywhere Server - SQL Reference*].

- **Date, time, and timestamp format settings** When a text configuration object is created, the values for date_format, time_format, timestamp_format, and timestamp_with_time_zone_format options for the current connection are stored with the text configuration object. These option values control how DATE, TIME, and TIMESTAMP columns are formatted for the text indexes built using the text configuration object. You cannot explicitly set these option values for the text configuration object; the settings reflect those in effect for the connection that created the text configuration object. However, you can change them. See "How to alter a text configuration object" on page 329, and the SAVE OPTION VALUES clause of the "ALTER TEXT CONFIGURATION statement" [*SQL Anywhere Server - SQL Reference*].

See also

- "Default text configuration objects" on page 332
- "How to create a text configuration object" on page 328
- "How to alter a text configuration object" on page 329
- "Example text configuration objects" on page 331
- "Fuzzy searching" on page 304
- "How to manage text indexes" on page 339
- "CONTAINS search condition" [*SQL Anywhere Server - SQL Reference*]
- "CREATE TEXT CONFIGURATION statement" [*SQL Anywhere Server - SQL Reference*]
- "ALTER TEXT CONFIGURATION statement" [*SQL Anywhere Server - SQL Reference*]
- "DROP TEXT CONFIGURATION statement" [*SQL Anywhere Server - SQL Reference*]
- "SYSTEXTIDX system view" [*SQL Anywhere Server - SQL Reference*]

How to create a text configuration object

When you create a text configuration object using SQL statements, you use another text configuration object as a template. Then, you alter the configuration settings and create your text index using the new text configuration.

When you create a text configuration object in Sybase Central, the **Create Text Configuration Object Wizard** allows you to configure settings during creation.

To create a text configuration object (Sybase Central)

1. Use the SQL Anywhere 12 plug-in to connect to the database as a user with DBA authority.
2. Right-click **Text Configuration Objects** and choose **New** » **Text Configuration Object**.
3. Follow the instructions in the **Create Text Configuration Object Wizard**.

   The new text configuration object appears in the **Text Configuration Objects** pane.

To create a text configuration object (SQL)

1. Connect to the database as a user with DBA authority.
2. Execute a CREATE TEXT CONFIGURATION statement.

   For example, the following statement creates a text configuration object called myTxtConfig using the default_char text configuration object as a template:

   ```
   CREATE TEXT CONFIGURATION myTxtConfig FROM default_char;
   ```

See also

- "Text configuration object settings" on page 323
- "Example text configuration objects" on page 331
- "How to view text configuration objects in the database" on page 331
- "CREATE TEXT CONFIGURATION statement" [*SQL Anywhere Server - SQL Reference*]
- "ALTER TEXT CONFIGURATION statement" [*SQL Anywhere Server - SQL Reference*]
- "Default text configuration objects" on page 332

How to alter a text configuration object

A text index is dependent on the text configuration object used to create it so you must be sure to truncate or drop dependent text indexes. Also, if you intend to change the date or time format options that are saved for the text configuration object, you must connect to the database with the options set to the desired settings.

To alter a text configuration object (SQL)

1. Connect to the database as a user with DBA authority, or as owner of the text configuration object. If you intend to change the date and time format options for the text configuration object, ensure you connect to the database with the options set to the desired values.
2. Truncate text indexes that are dependent on the text configuration object. If a text index is defined as IMMEDIATE REFRESH, you must drop it instead.
3. Execute an ALTER TEXT CONFIGURATION statement. For example, the following statement alters the minimum term length for the myTxtConfig text configuration object:

```
ALTER TEXT CONFIGURATION myTxtConfig
MINIMUM TERM LENGTH 2;
```

To alter a text configuration object (Sybase Central)

1. Use the SQL Anywhere 12 plug-in to connect to the database as a user with DBA authority or as the owner of the text configuration object. If you intend to change the date and time format options for the text configuration object, ensure you connect to the database with the options set to the desired values.
2. In the left pane, click **Text Configuration Objects**.
3. Right-click the text configuration object and choose **Properties**.
4. Edit the text configuration object properties and click **OK**.

See also

- “Text configuration object settings” on page 323
- “Example text configuration objects” on page 331
- “How to view text configuration objects in the database” on page 331
- “CREATE TEXT CONFIGURATION statement” [*SQL Anywhere Server - SQL Reference*]
- “ALTER TEXT CONFIGURATION statement” [*SQL Anywhere Server - SQL Reference*]
- “Default text configuration objects” on page 332

Text configuration objects and database options

When a text configuration object is created, the current settings for the date_format, time_format, and timestamp_format database options are stored with the text configuration object. This is because these settings affect string conversions when creating and refreshing the text indexes that depend on the text configuration object.

Storing the settings with the text configuration object allows you change the settings for these database options without causing a change to the format of data stored in the dependent text indexes.

If you want to change the format of the strings representing the dates and times in a text index, you must do the following:

1. Drop the text index, the text configuration object and all its dependent text indexes.
2. Drop the default text configuration object that you used to create the text configuration object and all its dependent text indexes.
3. Change the database options to the format you want.
4. Create a text configuration object.
5. Create a text index using the new text configuration object.

Note
The conversion_error option must be set to ON when creating or refreshing a text index.

See also

- "Text configuration object settings" on page 323
- "date_format option" [*SQL Anywhere Server - Database Administration*]
- "time_format option" [*SQL Anywhere Server - Database Administration*]
- "timestamp_format option" [*SQL Anywhere Server - Database Administration*]
- "conversion_error option" [*SQL Anywhere Server - Database Administration*]

How to view text configuration objects in the database

You can view information about text configuration objects in the database using Sybase Central, or by using a SQL statement.

To view settings for a text configuration object (Sybase Central)

1. Use the SQL Anywhere 12 plug-in to connect to the database as a user with DBA authority or as the owner of the text configuration object.
2. In the left pane, click **Text Configuration Objects**.
3. Double-click the text configuration object to view its settings.

To view settings for a text configuration object (SQL)

1. Connect to the database as a user with DBA authority, or as the owner of the text configuration object.
2. Query the SYSTEXTCONFIG system view, as follows:

```
SELECT * FROM SYSTEXTCONFIG;
```

See also

- "Text configuration object settings" on page 323
- "SYSTEXTCONFIG system view" [*SQL Anywhere Server - SQL Reference*]

Example text configuration objects

For in-depth descriptions of text configuration object settings and how they impact the contents of a text index and the results returned when querying a text index, see "Text configuration object settings" on page 323.

For a list of all text configuration objects in the database and the settings they contain, query the SYSTEXTCONFIG system view (for example, `SELECT * FROM SYSTEXTCONFIG`). See "SYSTEXTCONFIG system view" [*SQL Anywhere Server - SQL Reference*].

You can test how a text configuration object would break a string into terms using the sa_char_terms and sa_nchar_terms system procedures. See "sa_char_terms system procedure" [*SQL Anywhere Server - SQL Reference*], and "sa_nchar_terms system procedure" [*SQL Anywhere Server - SQL Reference*].

Default text configuration objects

SQL Anywhere provides two default text configuration objects, **default_nchar** and **default_char** for use with NCHAR and non-NCHAR data, respectively. These configurations are created the first time you attempt to create a text configuration object or text index. If you delete one by mistake, it is recreated the next time you attempt to create a text configuration object or text index.

The settings for default_char and default_nchar at the time of installation are shown in the table below. These settings were chosen because they were best suited for most character-based languages. It is strongly recommended that you do not change the settings in the default text configuration objects.

Setting	Installed value
TERM BREAKER	0 (GENERIC)
MINIMUM TERM LENGTH	1
MAXIMUM TERM LENGTH	20
STOPLIST	(empty)

If you delete a default text configuration object, it is automatically recreated the next time you create a text index or text configuration object. See "DROP TEXT CONFIGURATION statement" [*SQL Anywhere Server - SQL Reference*].

When a default text configuration object is created by the database server, the database options that affect how date and time values are converted to strings are saved to the text configuration object from the current connection. See "Text configuration objects and database options" on page 330.

For a description of text configuration object settings, see "Text configuration object settings" on page 323.

Example text configuration objects

The following table shows the settings for different text configuration objects and how the settings impact what is indexed and how a full text query string is interpreted. All the examples use the string `'I'm not sure I understand'`.

Configuration settings	Terms that are indexed	Query interpretation
TERM BREAKER GENERIC MINIMUM TERM LENGTH 1 MAXIMUM TERM LENGTH 20 STOPLIST "	`I m not sure I understand`	`("I m" AND NOT sure) AND I AND understand'` Note that the 'not' in the original string gets interpreted as an operator, not the word 'not'.
TERM BREAKER GENERIC MINIMUM TERM LENGTH 2 MAXIMUM TERM LENGTH 20 STOPLIST 'not and'	`sure understand`	`'understand'.` Note that 'sure' gets dropped because 'not' is interpreted as an operator (AND NOT) between phrase "i am" and "sure". Since the phrase "i am" contains terms that are too short and are dropped, the right side of the AND NOT condition ('sure') is also dropped. This leaves only 'understand'.

TERM BREAKER NGRAM
 	
MAXIMUM TERM LENGTH 3
 	
STOPLIST 'not and'
	`sur ure und nde der ers rst sta tan`	`'und AND nde AND der AND ers AND rst AND sta AND tan'.`
		
		For a fuzzy search:
		
		`'und OR nde OR der OR ers OR rst OR sta OR tan'`

TERM BREAKER GENERIC MINIMUM TERM LENGTH 1 MAXIMUM TERM LENGTH 20 STOPLIST 'not and'	`I m sure I understand`	`'("I m" AND NOT sure) AND I AND understand'.`

Configuration settings	Terms that are indexed	Query interpretation
TERM BREAKER NGRAM MAXIMUM TERM LENGTH 20 STOPLIST 'not and'	Nothing is indexed because no term is equal to or longer than 20 characters. This illustrates how differently MAXIMUM TERM LENGTH impacts GENERIC and NGRAM text indexes; on NGRAM text indexes, MAXIMUM TERM LENGTH sets the length of the n-grams inserted into the text index.	The search returns an empty result set because no n-grams of 20 characters can be formed from the query string.

You can test how a text configuration object would break a string into terms using the sa_char_terms and sa_nchar_terms system procedures. See "sa_char_terms system procedure" [*SQL Anywhere Server - SQL Reference*], and "sa_nchar_terms system procedure" [*SQL Anywhere Server - SQL Reference*].

Example CONTAINS string interpretations

The following table provides examples of how the settings of the text configuration object strings are interpreted.

The parenthetical numbers in the Interpreted string column reflect the position information stored for each term. The numbers are for illustration purposes in the documentation. The actual stored terms do not include the parenthetical numbers.

Configuration settings	String	Interpreted String
TERM BREAKER GENERIC MINIMUM TERM LENGTH 3 MAXIMUM TERM LENGTH 20	`'w*'`	`'"w*(1)"'`
	`'we*'`	`'"we*(1)"'`
	`'wea*'`	`'"wea*(1)"'`
	`'we* -the'`	`'"we*(1)" -"the(1)"'`
	`'we* the'`	`"we*(1)" & "the(1)"'`
	`'for* | wonderl*'`	`'"for*(1)" | "wonderl*(1)"'`
	`'wonderlandwonderlandwonderland*'`	`''`

Configuration settings	String	Interpreted String
	'"tr* weather"'	'"weather(1)"'
	'"tr* the weather"'	'"the(1) weather(2)"'
	'"wonderlandwonderland-wonderland* wonderland"'	'"wonder-land(1)"'
	'"wonderlandwonderland-wonderland* weather"'	'"weather(1)"'
	'"the_wonderlandwonder-landwonderland* weather"'	'"the(1) weather(3)"'
	'the_wonderlandwonder-landwonderland* weather'	'"the(1)" & "weather(1)"'
	'"light_a* the end" & tun-nel'	'"light(1) the(3) end(4)" & "tunnel(1)"'
	light_b* the end" & tun-nel'	'"light(1) the(3) end(4)" & "tunnel(1)"'
	'"light_at_b* end"'	'"light(1) end(4)"'
	'and_te*'	'"and(1) te*(2)"'
	'a_long_and_t* & journey'	'"long(2) and(3) t*(4)" & "jour-ney(1)"'
	'weather -is'	'"weather(1)"'
TERM BREAKER NGRAM MAXIMUM TERM LENGTH 3	'w*'	'"w*(1)"'
	'we*'	'"we*(1)"'
	'wea*	'"wea(1)"'
	'we* -the'	'"we*(1)" -"the(1)"'

Configuration settings	String	Interpreted String
	'we* the'	'"we*(1)" & "the(1)"'
	'for | la*'	'"for(1)" | "la*(1)"'
	'weath*'	'"wea(1) eat(2) ath(3)"'
	'"ful weat*"'	'"ful(1) wea(2) eat(3)"'
	'"wo* la*"'	'"wo*(1)" & "la*(2)"'
	'"la* won* "'	'"la*(1)" & "won(2)"'
	'"won* weat*"'	'"won(1)" & "wea(2) eat(3)"'
	'"won* weat"'	'"won(1)" & "wea(2) eat(3)"'
	'"wo* weat*"'	'"wo*(1)" & "wea(2) eat(3)"'
	'"weat* wo* "'	'"wea(1) eat(2)" & "wo*(3)"'
	'"wo* weat"'	'"wo*(1)" & "wea(2) eat(3)"'
	'"weat wo* "'	'"wea(1) eat(2) wo*(3)"'
	'w* NEAR[1] f*'	'"w*(1)" & "f*(1)"'

Configuration settings	String	Interpreted String
	'weat* NEAR[1] f*'	"wea(1) eat(2)" & "f*(1)"'
	'f* NEAR[1] weat*'	'"f*(1)" & "wea(1) eat(2)"'
	'weat NEAR[1] f*'	'"wea(1) eat(2)" & "f*(1)"'
	'f* NEAR[1] weat'	'"f*(1)" & "wea(1) eat(2)"'
	'for NEAR[1] weat*'	'"for(1)" & "wea(1) eat(2)"'
	'weat* NEAR[1] for'	'"wea(1) eat(2)" & "for(1)"'
	'and_tedi*'	'"and(1) ted(2) edi(3)"'
	'and_t*'	'"and(1) t*(2)"'
	'"and_tedi*"'	'"and(1) ted(2) edi(3)"'
	'"and-t*"'	'"and(1) t*(2)"'
	'"ligh* at_the_end of_the tun* nel"'	'"lig(1) igh(2)" & ("the(4) end(5) the(7) tun(8)" & "nel(9)")'

Configuration settings	String	Interpreted String
	'"ligh* at_the_end_of_the_tun* nel"'	'"lig(1) igh(2)" & ("the(4) end(5) the(7) tun(8)" & "nel(9)")'
	'"at_the_end of_the tun* ligh* nel"'	'"the(2) end(3) the(5) tun(6)" & ("lig(7) igh(8)" & "nel(9)")'
	'l* NEAR[1] and_t*'	"l*(1)" & "and(1) t*(2)"'
	'long NEAR[1] and_t*'	'"lon(1) ong(2)" & "and(1) t*(2)"'
	'end NEAR[3] tunne*'	'"end(1)" & "tun(1) unn(2) nne(3)"'
TERM BREAKER NGRAM MAXIMUM TERM LENGTH 3 SKIPPED TOKENS IN TABLE AND IN QUERIES	'"cat in a hat"'	'"cat(1) hat(4)"'
	'"cat in_a hat"'	'"cat(1) hat(4)"'
	'"cat_in_a_hat"'	'"cat(1) hat(4)"'
	'"cat_in a_hat"'	'"cat(1) hat(4)"'
	'cat in a hat'	'"cat(1)" & "hat(1)"'
	'cat in_a hat'	'"cat(1)" & "hat(1)"'

Configuration settings	String	Interpreted String
	'"ice hat"'	'"ice(1) hat(2)"'
	'ice NEAR[1] hat'	'"ice(1)" NEAR[1] "hat(1)"'
	'ear NEAR[2] hat'	'"ear(1)" NEAR[2] "hat(1)"'
	'"ear a hat"'	'"ear(1) hat(3)"'
	'"cat hat"'	'"cat(1) hat(2)"'
	'cat NEAR[1] hat'	'"cat(1)" NEAR[1] "hat(1)"'
	'ear NEAR[1] hat'	'"ear(1)" NEAR[1] "hat(1)"'
	'"ear hat"'	'"ear(1) hat(2)"'
	'"wear a a hat"'	'"wea(1) ear(2) hat(5)"'
	'weather -is'	'"wea(1) eat(2) ath(3) the(4) her(5)"'

You can test how a text configuration object would break a string into terms using the sa_char_terms and sa_nchar_terms system procedures. See “sa_char_terms system procedure” [*SQL Anywhere Server - SQL Reference*], and “sa_nchar_terms system procedure” [*SQL Anywhere Server - SQL Reference*].

How to manage text indexes

When you perform a full text search, you are searching a **text index** (not table rows). So, before you can perform a full text search, you must create a text index on the columns you want to search. A text index

stores positional information for terms in the indexed columns. Queries that use text indexes can be faster than those that must scan all the values in the table.

When you create a text index, you can specify which **text configuration object** to use when creating and refreshing the text index. A text configuration object contains settings that affect how an index is built. If you do not specify a text configuration object, the database server uses a default configuration object. See "How to manage text configuration objects" on page 323.

You can also specify a **refresh type** for the text index. The refresh type defines how often the text index is refreshed. A more recently refreshed text index returns more accurate results. However, refreshing takes time and can impede performance. For example, frequent updates to an indexed table can impact performance if the text index is configured to refresh each time the underlying data changes. See "Text index refresh types" on page 340.

To view settings for existing text indexes, use the sa_text_index_stats system procedure. See "sa_text_index_stats system procedure" [*SQL Anywhere Server - SQL Reference*].

Text index refresh types

When you create a text index, you must also choose a **refresh type**. There are three refresh types supported for text indexes: immediate, automatic, and manual. You define the refresh type for a text index at creation time. With the exception of immediate text indexes, you can change the refresh type after creating the text index.

For information on how to set the refresh type see "CREATE TEXT INDEX statement" [*SQL Anywhere Server - SQL Reference*], and "ALTER TEXT INDEX statement" [*SQL Anywhere Server - SQL Reference*].

- **IMMEDIATE REFRESH** IMMEDIATE REFRESH text indexes are refreshed when data in the underlying table changes, and are recommended only when the data must always be up-to-date, when the indexed columns are relatively short, or when the data changes are infrequent.

  The default refresh type for text indexes is IMMEDIATE REFRESH.

  If you have an AUTO REFRESH or MANUAL REFRESH text index, you cannot alter it to be an IMMEDIATE REFRESH text index. Instead, you must drop and recreate it as an IMMEDIATE REFRESH text index.

  IMMEDIATE REFRESH text indexes support all isolation levels. They are populated at creation time, and an exclusive lock is held on the table during this initial refresh.

- **AUTO REFRESH** AUTO REFRESH text indexes are refreshed automatically at a time interval that you specify, and are recommended when some data staleness is acceptable. A query on a stale index returns matching rows that have not been changed since the last refresh. So, rows that have been inserted, deleted, or updated since the last refresh are not returned by a query.

  AUTO REFRESH text indexes may also be refreshed more often than the interval specified when the following conditions are true:

- the time since the last refresh is larger than the refresh interval.
- the total length of all pending rows (pending_length as returned by the sa_text_index_stats system procedure) exceeds 20% of the total index size (doc_length as returned by sa_text_index_stats).
- the deleted length is 50% of the total index size (doc_length). A full rebuild of the text index is completed.

AUTO REFRESH text indexes are refreshed using isolation level 0.

An AUTO REFRESH text index contains no data at creation time, and is not available for use until after the first refresh, which takes place usually within the first minute after the text index is created. You can also refresh an AUTO REFRESH text index manually using the REFRESH TEXT INDEX statement.

AUTO REFRESH text indexes are not refreshed during a reload unless the -g option is specified for dbunload. See "Unload utility (dbunload)" [*SQL Anywhere Server - Database Administration*].

- **MANUAL REFRESH** MANUAL REFRESH text indexes are refreshed only when you refresh them, and are recommended if data in the underlying table is rarely changed, or if a greater degree of data staleness is acceptable, or if you want to refresh after an event or a condition is met. A query on a stale index returns matching rows that have not been changed since the last refresh. So, rows that have been inserted, deleted, or updated since the last refresh are not returned by a query.

  You can define your own strategy for refreshing MANUAL REFRESH text indexes. In the following example, all MANUAL REFRESH text indexes are refreshed using a refresh interval that is passed as an argument, and rules that are similar to those used for AUTO REFRESH text indexes.

```
CREATE PROCEDURE refresh_manual_text_indexes(
   refresh_interval UNSIGNED INT )
BEGIN
 FOR lp1 AS c1 CURSOR FOR
   SELECT ts.*
   FROM SYS.SYSTEXTIDX ti JOIN sa_text_index_stats( ) ts
   ON ( ts.index_id = ti.index_id )
   WHERE ti.refresh_type = 1 -- manual refresh indexes only
 DO
   BEGIN
    IF last_refresh IS null
    OR cast(pending_length as float) / (
       IF doc_length=0 THEN NULL ELSE doc_length ENDIF) > 0.2
    OR DATEDIFF( MINUTE, CURRENT TIMESTAMP, last_refresh )
       > refresh_interval THEN
     EXECUTE IMMEDIATE 'REFRESH TEXT INDEX ' || text-index-name || ' ON "'
      || table-owner || '"."' || table-name || '"';
    END IF;
   END;
  END FOR;
END;
```

At any time, you can use the sa_text_index_stats system procedure to decide if a refresh is needed, and whether the refresh should be a complete rebuild or an incremental update. See "sa_text_index_stats system procedure" [*SQL Anywhere Server - SQL Reference*].

A MANUAL REFRESH text index contains no data at creation time, and is not available for use until you refresh it. To refresh a MANUAL REFRESH text index, use the REFRESH TEXT INDEX statement.

MANUAL REFRESH text indexes are not refreshed during a reload unless the -g option is specified for dbunload. See "Unload utility (dbunload)" [*SQL Anywhere Server - Database Administration*].

See also

- "How to create a text index" on page 342
- "Text configuration object settings" on page 323
- "sa_text_index_stats system procedure" [*SQL Anywhere Server - SQL Reference*]
- "REFRESH TEXT INDEX statement" [*SQL Anywhere Server - SQL Reference*]
- "isolation_level option" [*SQL Anywhere Server - Database Administration*]

How to create a text index

You can create text indexes on columns of any type. Columns that are not of type VARCHAR or NVARCHAR are converted to strings during indexing. See "Data type conversions" [*SQL Anywhere Server - SQL Reference*].

Text indexes consume disk space and need to be refreshed. Create them only on the columns that are required to support your queries.

You cannot create a text index on a materialized view, a regular view, or a temporary table.

Do not create more than one text index referencing a column since this can return unexpected results.

To create a text index (Sybase Central)

1. Use the SQL Anywhere 12 plug-in to connect to the database as a user with DBA authority or as the owner of the table on which you are creating the text index.
2. Click the **Text Indexes** tab.
3. Choose **File** » **New** » **Text Index**.
4. Follow the instructions in the **Create Index Wizard**.

   The new text index appears on the **Text Indexes** tab. It also appears in the **Text Indexes** folder.
5. If you created an immediate refresh text index, it is automatically populated with data. For other refresh types, you must refresh the text index by right-clicking it and choosing **Refresh Data**.

To create a new text index (SQL)

1. Connect to the database as a user with DBA authority, or as the owner of the table on which you are creating the text index.

2. Execute a CREATE TEXT INDEX statement. See "CREATE TEXT INDEX statement" [*SQL Anywhere Server - SQL Reference*].

3. If you created an immediate refresh text index, it is automatically populated with data. For other refresh types, you must refresh the text index by executing a REFRESH TEXT INDEX statement. See "REFRESH TEXT INDEX statement" [*SQL Anywhere Server - SQL Reference*].

See also

- "Text index refresh types" on page 340

How to refresh a text index

You can only refresh text indexes that are defined as AUTO REFRESH and MANUAL REFRESH.

To refresh a text index (Sybase Central)

1. Use the SQL Anywhere 12 plug-in to connect to the database as a user with DBA authority or as the owner of the table on which the text index is built.

2. In the left pane, click **Text Indexes**.

3. Right-click the text index and choose **Refresh Data**.

4. Select an isolation level for the refresh and click **OK**.

To refresh a text index (SQL)

1. Connect to the database as a user with DBA authority, or as the owner of the table on which the text index is built.

2. Execute a REFRESH TEXT INDEX statement.

   For example, the following statement refreshes a text index called MarketingTextIndex on the Description column of the MarketingInformation table in the sample database:

   ```
   REFRESH TEXT INDEX MarketingTextIndex ON MarketingInformation
      WITH ISOLATION LEVEL SNAPSHOT;
   ```

   See "REFRESH TEXT INDEX statement" [*SQL Anywhere Server - SQL Reference*].

See also

- "Text index refresh types" on page 340

Altering text indexes overview

You can alter the following characteristics of a text index:

- **Refresh type** You can change the refresh type from AUTO REFRESH to MANUAL REFRESH, and vice versa. Use the REFRESH clause of the ALTER TEXT INDEX statement to change the refresh type. See "ALTER TEXT INDEX statement" [*SQL Anywhere Server - SQL Reference*].

  You cannot change a text index to, or from, IMMEDIATE REFRESH; to make this change, you must drop the text index and recreate it. See "DROP TEXT INDEX statement" [*SQL Anywhere Server - SQL Reference*], and "CREATE TEXT INDEX statement" [*SQL Anywhere Server - SQL Reference*].

- **Name** You can rename the text index using the RENAME clause of the ALTER TEXT INDEX statement. See "ALTER TEXT INDEX statement" [*SQL Anywhere Server - SQL Reference*].

- **Content** With the exception of the column list, settings that control what is indexed are stored in a text configuration object. If you want to change what is indexed, you alter the text configuration object that a text index refers to. You must truncate dependent text indexes before you can alter the text configuration object, and refresh the text index after altering the text configuration object. For immediate refresh text indexes, you must drop the text index and recreate it after you alter the text configuration object.

  See:

  - "How to manage text configuration objects" on page 323
  - "TRUNCATE TEXT INDEX statement" [*SQL Anywhere Server - SQL Reference*]
  - "ALTER TEXT INDEX statement" [*SQL Anywhere Server - SQL Reference*]
  - "REFRESH TEXT INDEX statement" [*SQL Anywhere Server - SQL Reference*]
  - "sa_refresh_text_indexes system procedure" [*SQL Anywhere Server - SQL Reference*]

You cannot alter a text index to refer to a different text configuration object. If you want a text index to refer to another text configuration object, drop the text index and recreate it specifying the new text configuration object.

How to alter a text index

You can change the name of a text index, or change its refresh type.

To alter the refresh type for a text index (Sybase Central)

1. Use the SQL Anywhere 12 plug-in to connect to the database as a user with DBA authority or as the owner of the table on which the text index is built.
2. In the left pane, click **Text Indexes**.
3. Right-click the text index and choose **Properties**.
4. Edit the text index properties and click **OK**.

To alter the refresh type for a text index (SQL)

1. Connect to the database as a user with DBA authority, or as the owner of the text index.

2. Execute an ALTER TEXT INDEX statement. See "ALTER TEXT INDEX statement" [*SQL Anywhere Server - SQL Reference*].

To rename a text index (Sybase Central)

1. Use the SQL Anywhere 12 plug-in to connect to the database as a user with DBA authority or as the owner of the table on which the text index is built.
2. In the left pane, click **Text Indexes**.
3. Right-click the text index and choose **Properties**.
4. Click the **General** tab and type a new name for the text index.
5. Click **OK**.

To rename a text index (SQL)

1. Connect to the database as a user with DBA authority, or as the owner of the text index.
2. Execute an ALTER TEXT INDEX statement. See "ALTER TEXT INDEX statement" [*SQL Anywhere Server - SQL Reference*].

See also

- "How to manage text indexes" on page 339

How to view text index info in the database

You can view information about text indexes in the database using Sybase Central, or by using a SQL statement.

To view text indexes in the database (Sybase Central)

1. Connect to the database as a user with DBA authority or as the owner of the text index.
2. In the left pane, click **Text Indexes**.
3. To view the terms in the text index, double-click the text index in the left pane, and then choose the **Vocabulary** tab in the right pane.
4. To view the text index settings, such as the refresh type or the text configuration object that the index refers to, right-click the text index and choose **Properties**.

To view text indexes in the database (SQL)

1. Connect to the database as a user with DBA authority, or as the owner text index.

2. Call the sa_text_index_stats system procedure, as follows:

```
CALL sa_text_index_stats( );
```

To retrieve text index creation options

- When a text index is created, the current database options are stored with the text index. To retrieve the option settings used during text index creation, execute the following statement:

```
SELECT b.object_id, b.table_name, a.option_id, c.option_name,
a.option_value
FROM SYSMVOPTION a, SYSTAB b, SYSMVOPTIONNAME c
WHERE a.view_object_id=b.object_id
AND b.table_type=5;
```

  A table_type of 5 in the SYSTAB view is a text index.

See also

- "sa_text_index_stats system procedure" [*SQL Anywhere Server - SQL Reference*]
- "SYSMVOPTION system view" [*SQL Anywhere Server - SQL Reference*]
- "SYSMVOPTIONNAME system view" [*SQL Anywhere Server - SQL Reference*]
- "SYSTAB system view" [*SQL Anywhere Server - SQL Reference*]

External term breaker and prefilter libraries

Why use an external term breaker or prefilter library

Full text search in SQL Anywhere is performed using a text index. Each value in a column on which a text index has been built is referred to as a **document**. When a text index is created, each document is processed by a built-in term breaker to determine the **terms** (also referred to as **tokens**) contained in the document, and the positions of the terms in the document. The built-in term breaker also performs term breaking on the documents (text components) of a query string. For example, the query string 'rain or shine' consists of two documents, 'rain' and 'shine', connected by the OR operator. The built-in term breaker algorithm is also used to break into terms the stoplist and the input of a sa_char_terms system procedure.

Depending on the needs of your application, you may find some behaviors of the built-in term breaker to be undesirable or limiting. For example, the built-in term breaker does not offer language-specific term breaking. Here are some other reasons you may want to implement custom **term breakering**:

- **No language-specific term breaking** Linguistic rules with respect to what constitutes a term differs from one language to another. Consequently, term breaking rules are different from one language to another. The built-in term breaker does not offer language-specific term breaking rules.

- **Handling of words with apostrophes** The word "they'll" is treated as "they ll" by the GENERIC term breaker. However, you could design a custom term breaker that treats the apostrophe as part of the word.

- **No support for term replacement** You cannot specify replacements for a term. For example, when indexing the word "they'll", you might want to store it as two terms: they and will. Likewise, you may want to use term replacement to perform a case insensitive search on a case sensitive database.

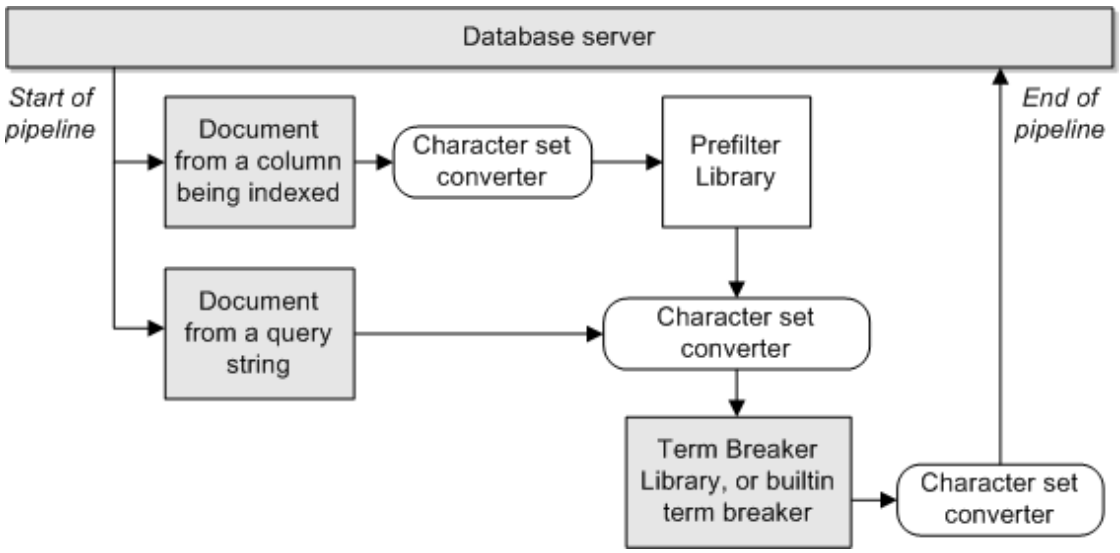
SQL Anywhere also allows you to use external prefilter libraries to perform **prefiltering** on data before it is indexed. Prefiltering allows you to extract only the textual content you want indexed from a document. For example, suppose you want to create a text index on a column containing XML values. A prefilter allows you to filter out the XML tags so that they are not indexed with the content.

SQL Anywhere provides an API you can use to access custom and 3rd party prefilter and term breaker libraries when creating and updating full text indexes. This means you can use external libraries to take document formats like XML, PDF, and Word and remove unwanted terms and content before indexing their content.

Some sample prefilter and term breaker libraries are included in your *Samples* directory to help you design your own, or you can use the API to access 3rd party libraries. If Microsoft Office is installed on the system running the database server then IFilters for Office documents such as Word and Excel are available. If the server has Acrobat Reader installed, then a PDF IFilter is likely available.

Full text pipeline workflow

The following diagram shows how data is converted from a document to a stream of terms to index within SQL Anywhere. The workflow for creating a text index, updating it, and querying it, is referred to as the **pipeline**. The mandatory parts of the pipeline are depicted in light gray. Arrows show the flow of data through the pipeline. Function calls are propagated in the opposite direction.



High level view of how the pipeline works

1. The processing of each document is initiated by the database server calling the begin_document method on the end of the pipeline, which is either the term breaker or the character set converter. Each

component in the pipeline calls begin_document on its own producer, in order to satisfy, and before returning from its, begin_document method invocation.

2. The database server calls get_words on the end of the pipeline after the begin_document completes successfully.
   - While executing get_words, the term breaker calls get_next_piece on its producer to get data to process. If a prefilter exists in the pipeline, the data is filtered by it during the get_next_piece call.
   - The term breaker breaks the data it receives from its producer into terms according to its term breaking rules.
3. The database server applies the minimum and maximum term length settings, as well as the stoplist restrictions to the terms returned from get_words call.
4. The database server continues to call get_words until no more terms are returned. At that point, the database server calls end_document. This call is propagated through the pipeline in the same manner as the begin_document call.

Note
Character set converters are transparently added to the pipeline by the database server where necessary.

Prefilter and term breaker code samples

The *ExternalLibrariesFullText* directory in your SQL Anywhere install contains prefilter and term breaker sample code for you to explore. This directory is found under your *Samples* directory. For the location of your *Samples* directory, see "Samples directory" [*SQL Anywhere Server - Database Administration*].

See also

- "External prefilter library workflow" on page 349
- "External term breaker library workflow" on page 351

External prefilter libraries

How to configure SQL Anywhere to use an external prefilter

SQL Anywhere does not provide a built-in prefilter algorithm. To have data pass through an external prefilter library, you specify the library and its entry point function using the ALTER TEXT CONFIGURATION statement, similar to the following:

```
ALTER TEXT CONFIGURATION my_text_config
   PREFILTER EXTERNAL NAME 'my_prefilter@myprefilterLibrary.dll'
```

This example tells the database server to use the my_prefilter entry point function in the *myprefilterLibrary.dll* library to obtain a prefilter instance to use when building or updating a text index using the my_text_config text configuration object. The interface for a prefilter is described here: "a_text_source interface" on page 355.

See also: "ALTER TEXT CONFIGURATION statement" [*SQL Anywhere Server - SQL Reference*].

How to design an external prefilter library

The prefilter library must be implemented in C/C++, and must:

- include the prefilter interface definition header file, *extpfapi1.h*.
- implement the a_text_source interface. See “a_text_source interface” on page 355.
- provide an entry point function that initializes and returns an instance of a_text_source (prefilter) and the label of the character set supported by the prefilter.

Calling sequence for the prefilter

The following calling sequence is executed by the consumer of the prefilter for each document being processed:

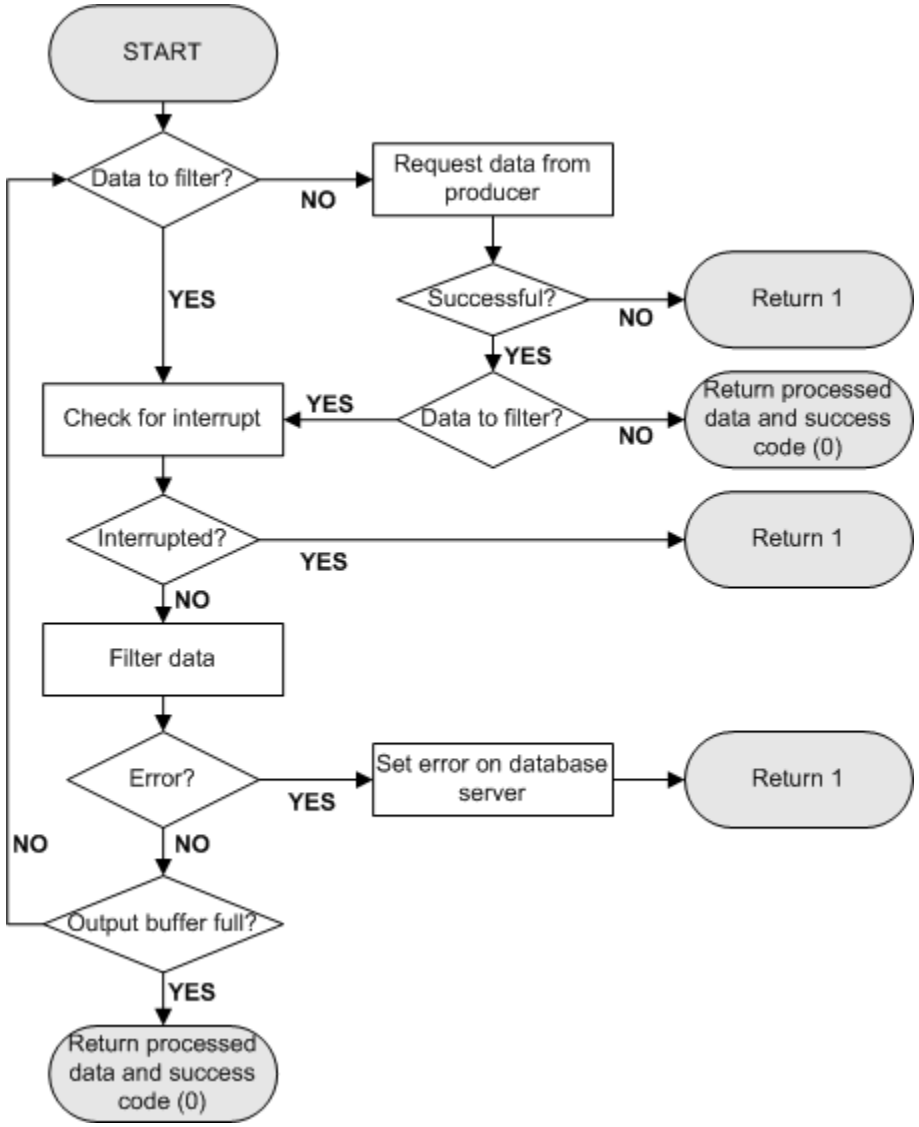
```
begin_document(a_text_source*)
get_next_piece(a_text_source*, buffer**, len*)
get_next_piece(a_text_source*, buffer**, len*)
...
end_document(a_text_source*)
```

Note
end_document can be called multiple times without an intervening begin_document call. For example, if one of the documents to be indexed is empty, the database server may call end_document for that document without calling begin_document.

The get_next_piece function should filter out the unnecessary data such as formatting information and images from the incoming byte stream and return the next chunk of filtered data in a self-allocated buffer.

External prefilter library workflow

The following flow chart shows the logic flow when the get_next_piece function is called:



See also

- “a_text_source interface” on page 355
- “Full text pipeline workflow” on page 347
- “Prefilter and term breaker code samples” on page 348

External term breaker libraries

How to configure SQL Anywhere to use an external term breaker

By default, when you create a text configuration object, a built-in term breaker is used for data associated with that text configuration object. To have data instead pass through an external term breaker library, you specify the library and its entry point function using the ALTER TEXT CONFIGURATION statement, similar to the following:

```
ALTER TEXT CONFIGURATION my_text_config
   TERM BREAKER GENERIC EXTERNAL NAME 'my_termbreaker@termbreaker'
```

This example tells the database server to use the my_termbreaker entry point function in the termbreaker library to obtain a term breaker instance to use when building, updating, or querying a text index associated with the my_text_config text configuration object, when parsing the text configuration object's stoplist, and when processing input to the sa_char_terms system procedure. The interface for a term breaker is described here: "a_word_source interface" on page 359.

See also: "ALTER TEXT CONFIGURATION statement" [*SQL Anywhere Server - SQL Reference*].

How to design an external term breaker library

The external term breaker library must be implemented in C/C++, and must:

- include of the term breaker interface definition header file, *exttbapi1.h*.
- implement the a_word_source interface. See "a_word_source interface" on page 359.
- provide an entry point function that initializes and returns an instance of a_word_source (term breaker) and the label of the character set supported by the term breaker.

Calling sequence for the term breaker

The following calling sequence is executed by the consumer of the term breaker for each document being processed:

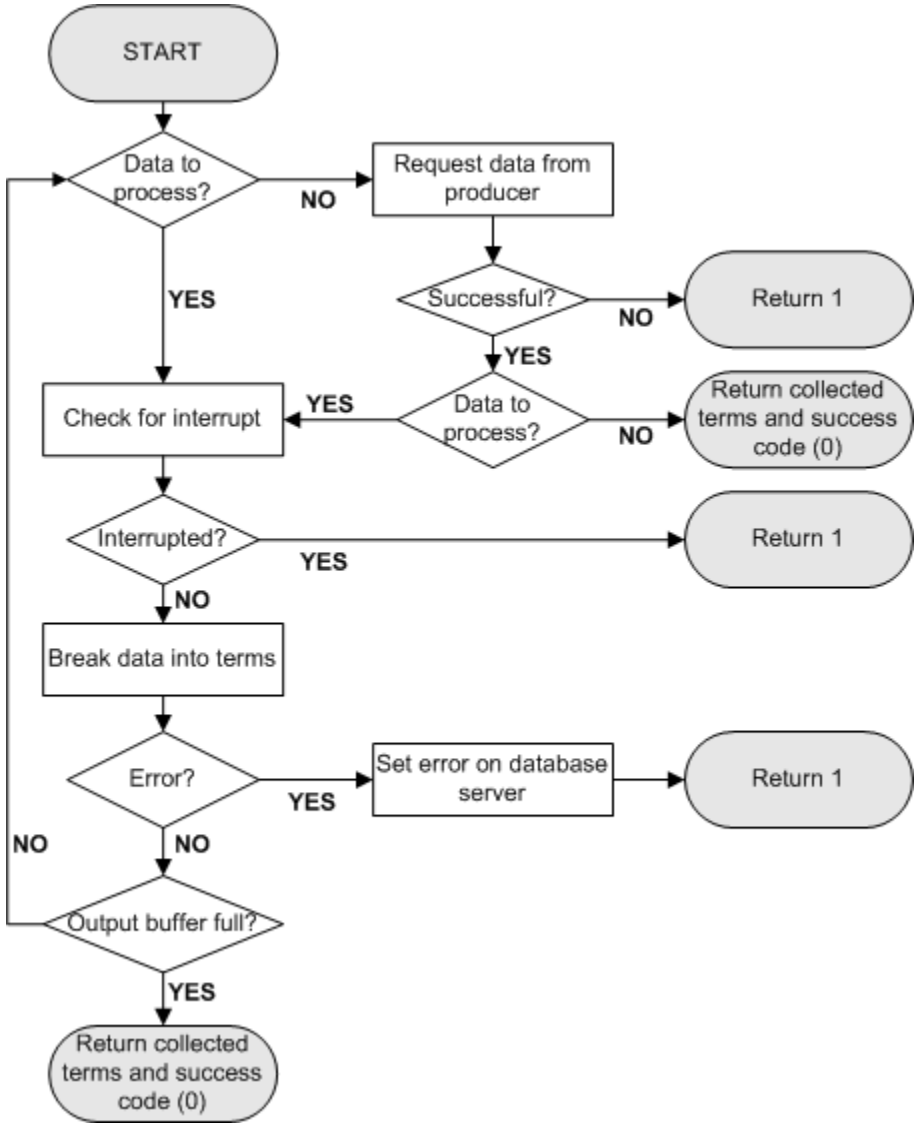
```
begin_document(a_word_source*, asql_uint32);
get_words(a_word_source*, a_term**, uint32 *num_words)
get_words(a_word_source*, a_term**, uint32 *num_words)
...
end_document(a_word_source*)
```

The get_words function must call get_next_piece on its producer to get data to break into terms until the array of a_term structures is filled, or there is no more data to process.

Note
end_document can be called multiple times without an intervening begin_document call. For example, if one of the documents to be indexed is empty, the database server may call end_document for that document without calling begin_document.

External term breaker library workflow

The following flow chart shows the logic flow when the get_words function is called:



See also

- “a_word_source interface” on page 359
- “a_text_source interface” on page 355
- “a_term structure” on page 361
- “Full text pipeline workflow” on page 347
- “Prefilter and term breaker code samples” on page 348

API for external full text libraries

The following steps need to be completed to create and use a prefilter or term breaker external library with text indexes:

- Implement the interfaces published by SQL Anywhere in C/C++.
- Create a dynamically loadable library by compiling and linking the code written in the above step.
- Create the text configuration object in the database and then modify it to specify the entry point function in the external library for prefilter and/or term breaker.

  The entry point functions are used to obtain the prefilter and term breaker objects to be used while inserting/deleting text index entries when underlying documents (column values) are modified. In the case of an external term breaker library, the entry point function is also used to parse queries over the text indexes that use the term breaker.

a_server_context structure

Several callbacks are supported by the database server and are exposed to the full text external libraries through the a_server_context structure to perform the following tasks:

- Error reporting
- Interrupt processing
- Message logging

Syntax

```
typedef struct a_server_context {
   void ( SQL_CALLBACK *set_error )(
                                    a_server_context *This
                                    , a_sql_uint32 error_code
                                    , const char* error_string
                                    , short str_len );
   void ( SQL_CALLBACK *log_message )(
                                    a_server_context *This
                                    , const char* message
                                    , short msg_len );
   a_sql_uint32 ( SQL_CALLBACK *get_is_cancelled )(
                                    a_server_context *This );
   void      *_context;
} a_server_context;
```

Members

Member name	Type	Description
set_error	`void`	This method allows external prefilters and term breakers to set an error in the database server by providing an error code and error string. The database server rolls back the current operation and returns the error code and string to the user in the following form: `"Error from external library: -<error_code>: <error_string>"` error_code must be a positive integer greater than 17000. error_string must be a null-terminated string. str_len is the length of error_string, in bytes.
log_message	`void`	This method allows external prefilters and term breakers to log messages to the database server log. *message* must be a null-terminated string. msg_len is the length of message, in bytes.
get_is_cancelled	`a_sql_uint32`	External prefilters and term breakers must periodically call this method to check if the current operation has been interrupted. This method returns 1 if the current operation was interrupted, and 0 if it was not interrupted. In the case of returning 1, the caller should stop further processing and return immediately.
_context	`void`	For internal use. A pointer to the database server context.

Remarks

The a_server_context structure is defined by a header file named *exttxtcmn.h*, in the *SDK\Include* subdirectory of your SQL Anywhere installation directory.

The external library should not be using any operating system synchronization primitives when calling the methods exposed by a_server_context structure.

a_init_pre_filter structure

Structure used for negotiating the input and output requirements for instances of an external prefilter entry point function. This structure is passed in as a parameter to the prefilter entry point function.

Syntax

```
typedef struct a_init_pre_filter {
   a_text_source          *in_text_source;   /* IN */
```



```
    a_text_source          *out_text_source;  /* OUT */
    const char             *desired_charset;  /* IN */
    char                   *actual_charset;   /* OUT */
    short                  is_binary;         /* IN */
} a_init_pre_filter;
```

Members

Name	Type	Description
in_text_source	`a_text_source *`	The pointer to the producer of the external prefilter (a_text_source object) to be created. Specified by the caller of the prefilter entry point function.
out_text_source	`a_text_source *`	The pointer to the external prefilter (a_text_source object) specifying by the prefilter entry point function.
desired_charset	`const char *`	The character set the caller of the entry point function expects the output of the prefilter to be in. If is_binary flag is 0, this is also the character set the input to the prefilter will be in, unless negotiated otherwise.
actual_charset	`char *`	The character set (specified by the external library as part of negotiation) the external prefilter library will produce its output in. If is_binary is 0, this is also the actual character set of the input to the prefilter. Note that it is preferable that the library accept and produce the data in desired_charset, if possible.
is_binary	`short`	Whether the input data is in binary (1) or in desired_charset (0). If the data is in binary, the database server does not introduce character set conversion before the prefilter on the pipeline.

Remarks

The a_init_pre_filter structure is defined by a header file named *extpfapiv1.h*, in the *SDK\Include* subdirectory of your SQL Anywhere installation directory.

See also

- “a_text_source interface” on page 355
- “a_word_source interface” on page 359
- “Prefilter entry point function” on page 364

a_text_source interface

The interface that an external prefilter library must implement to perform document prefiltering for full text index population or updating.

Syntax

```
typedef struct a_text_source {
   a_sql_uint32 ( SQL_CALLBACK *begin_document )( a_text_source *This );
   a_sql_uint32 ( SQL_CALLBACK *get_next_piece )(
                               a_text_source *This
                               , unsigned char ** buffer
                               , a_sql_uint32* len );
   a_sql_uint32 ( SQL_CALLBACK *end_document )( a_text_source *This);
   a_sql_uint64 ( SQL_CALLBACK *get_document_size )( a_text_source *This );
   a_sql_uint32 ( SQL_CALLBACK *fini_all )( a_text_source *This );
   a_server_context     *_context;

  // Only one of the following two members can have a valid pointer in a given
implementation.
   // These members point to the current module's producer
   a_text_source        *_my_text_producer;
   a_word_source        *_my_word_source;

   // Following members have been reserved for
   // future use ONLY
   a_text_source        *_my_text_consumer;
   a_word_source        *_my_word_consumer;
} a_text_source;
```

Members

Member	Type	Description
begin_document	a_sql_uint32	Performs the necessary setup steps for processing a document. This method returns 0 on success, and 1 if an error occurred or if no more documents are available.
get_next_piece	a_sql_uint32	Returns a fragment of the filtered input byte stream along with the length of the fragment. This method will be called multiple times for a given document, and should return subsequent chunks of the document at each call until all the input data for a document is consumed, or an error occurs. buffer is the OUT parameter to be populated by the prefilter to point to the produced data. Memory is managed by the prefilter. len is the OUT parameter indicating the length of the produced data.
end_document	a_sql_uint32	Marks completion of filtering for the given document, and performs document-specific cleanup, if necessary.
get_document_size	a_sql_uint64	Returns the total length of the document (in bytes) as produced by the prefilter. The a_text_source object must keep a current count of the total number of bytes produced by it so far for the current document.

Member	Type	Description
fini_all	`a_sql_uint32`	Called by the database server after the processing of all the documents is done and the pipeline is about to be closed. fini_all performs the final cleanup steps.
_context	`a_server_context *`	Use this member to hold the database server context that is provided to the entry point function within the a_init_prefilter structure. The prefilter module uses this context to establish direct communication with the database server.
_my_text_producer	`a_text_source *`	Use this member to store the pointer to the a_text_source producer of the prefilter that is provided to the entry point function within the a_init_prefilter structure. This pointer may be replaced by the database server after the entry point function has been executed if character set conversion is required. Therefore, only this pointer to the text producer can be used by the prefilter.
_my_word_producer	`a_word_source *`	Reserved for future use and should be initialized to NULL.
_my_text_consumer	`a_text_source *`	Reserved for future use and should be initialized to NULL.
_my_word_consumer	`a_word_source *`	Reserved for future use and should be initialized to NULL.

Remarks

The a_text_source interface is stream-based-data. The data is pulled from the producer in sequence; each byte is only seen once.

The a_text_source interface is defined by a header file named *extpfapiv1.h*, in the *SDK\Include* subdirectory of your SQL Anywhere installation directory.

The external library should not be holding any operating system synchronization primitives across function calls.

See also

- “a_server_context structure” on page 353
- “a_init_pre_filter structure” on page 354
- “Prefilter entry point function” on page 364

a_init_term_breaker structure

Structure used for negotiating the input and output requirements for instances of an external term breaker. This structure is passed as a parameter to the term breaker entry point function.

Syntax

```
typedef struct a_init_term_breaker {
   a_text_source               *in_text_source;    /* IN */
   a_word_source               *out_word_source;   /* OUT */
   const char                  *desired_charset;   /* IN */
   char                        *actual_charset;    /* OUT */
   const a_term_breaker_for    term_breaker_for;   /* IN */
} a_init_term_breaker;
```

Members

Member	Type	Description
in_text_source	`a_text_source *`	The pointer to the producer of the external term breaker (a_text_source object) to be created.
out_word_source	`a_word_source *`	The pointer to the external term breaker (a_word_source object) specified by the entry point function.
desired_charset	`const char *`	The character set the caller of the entry point function expects the output of the term breaker to be in. If is_binary flag is 0, this is also the character set the input to the term breaker will be in, unless negotiated otherwise.
actual_charset	`char *`	The character set (specified by the external library as part of negotiation) the external term breaker library will produce its output in. If is_binary is 0, this is also the actual character set of the input to the term breaker. Note that it is preferable that the library accept and produce the data in desired_charset, if possible.
term_breaker_for	`a_term_breaker_for`	The purpose for initializing the term breaker: • **TERM_BREAKER_FOR_LOAD** Used for create, insert, update, and delete operations on the text index. Input may be prefiltered if a prefilter is specified. • **TERM_BREAKER_FOR_QUERY** Used for parsing of query elements, stoplist, and input to the sa_char_term system procedure. In the case of TERM_BREAKER_FOR_QUERY, no prefiltering takes place, even if an external prefilter library is specified for the text index.

Remarks

The a_init_term_breaker structure is defined by a header file named *exttbapiv1.h*, in the *SDK\Include* subdirectory of your SQL Anywhere installation directory.

See also

- "sa_char_terms system procedure" [*SQL Anywhere Server - SQL Reference*]
- "a_term_breaker_for enumeration" on page 359
- "Term breaker entry point function" on page 365
- "a_text_source interface" on page 355
- "a_word_source interface" on page 359

a_term_breaker_for enumeration

Used to specify whether the pipeline is built for use during update or querying of the text index.

Parameters

- **TERM_BREAKER_FOR_LOAD** Used for create, insert, update, and delete operations on the text index.
- **TERM_BREAKER_FOR_QUERY** Used for parsing of query elements, stoplist, and input to the sa_char_term system procedure. In the case of TERM_BREAKER_FOR_QUERY, no prefiltering takes place, even if an external prefilter library is specified for the text index.

Remarks

The database server sets the value for a_init_term_breaker::term_breaker_for when it initializes the external term breaker.

```
typedef enum a_term_breaker_for {
    TERM_BREAKER_FOR_LOAD = 0,
    TERM_BREAKER_FOR_QUERY
} a_term_breaker
```

The a_term_breaker_for enumeration is defined by a header file named *exttbapiv1.h*, in the *SDK\Include* subdirectory of your SQL Anywhere installation directory.

See also

- "a_init_term_breaker structure" on page 357
- "sa_char_terms system procedure" [*SQL Anywhere Server - SQL Reference*]

a_word_source interface

The interface that an external term breaker library must implement to perform term breaking for text index operations.

Syntax

```
typedef struct a_word_source {
   a_sql_uint32 ( SQL_CALLBACK *begin_document )(
                                a_word_source *This
                                , a_sql_uint32 has_prefix );
   a_sql_uint32 ( SQL_CALLBACK *get_words )(
                                a_word_source *This
```

```
                                , a_term** words
                                ,a_sql_uint32 *num_words );
    a_sql_uint32 ( SQL_CALLBACK *end_document )(
                                a_word_source *This );
    a_sql_uint32 ( SQL_CALLBACK *fini_all )(
                                a_word_source *This );

    a_server_context    *_context;

    // Only one of the following pointers can be valid
    // in any implementation.
    // For example: if the producer for this module is
    // a a_text_source, then only my_text_producer will
    // be a valid pointer whereas my_word_producer
    // should be assigned a NULL
    a_text_source       *_my_text_producer;
    a_word_source       *_my_word_producer;

    // Following members have been reserved for
    // future use ONLY
    a_text_source       *_my_text_consumer;
    a_word_source       *_my_word_consumer;
} a_word_source;
```

Members

Member	Type	Description
begin_document	a_sql_uint32	Performs the necessary setup steps for processing a document. The parameter has_prefix is set to 1, not true, or TRUE if the document being tokenized is a prefix query term. If has_prefix is set to TRUE, the term breaker must return at least one term (possibly empty). has_prefix can only be 1, not true, or TRUE, if the purpose of pipeline initialization is TERM_BREAKER_FOR_QUERY. The result of prefix tokenization is treated as a phrase with the last term of the phrase being the actual prefix string.
get_words	a_sql_uint32	Returns a pointer to an array of a_term structures. This method is called in a loop for a given document until all the contents of the document has been broken into terms. The database server expects that two immediately consecutive terms in a document have positions differing by 1. If the term breaker is performing its own stoplist processing, it is possible that the difference between two consecutive terms returned is more than 1; this is expected and acceptable. However, in other cases where numbers are not consecutive with positions differing by 1, the arbitrary positions can affect how full text queries are executed and can cause unexpected results for subsequent full text queries.

Member	Type	Description
end_document	`a_sql_uint32`	Marks completion of processing of the document by the pipeline, and performs document-specific cleanup.
fini_all	`a_sql_uint32`	Called by the database server after processing of all the documents is done and the pipeline is about to be closed. fini_all performs the final cleanup steps.
_context	`a_server_context *`	The database server context that is provided to the entry point function within the a_init_term_breaker structure. The term breaker module uses this context to establish direct communication with the database server.
_my_text_producer	`a_text_source *`	Pointer to the a_text_source producer of the term breaker that is provided to the entry point function within the a_init_term_breaker structure. This pointer may be replaced by the database server after the entry point function has been executed if character set conversion is required. Therefore, only this pointer to the text producer can be used by the term breaker.
_my_word_producer	`a_word_source *`	Reserved for future use and should be initialized to NULL.
_my_text_consumer	`a_text_source *`	Reserved for future use and should be initialized to NULL.
_my_word_consumer	`a_word_source *`	Reserved for future use and should be initialized to NULL.

Remarks

The a_word_source interface is defined by a header file named *exttbapiv1.h*, in the *SDK\Include* subdirectory of your SQL Anywhere installation directory.

The external library should not be holding any operating system synchronization primitives across function calls.

See also

- "a_server_context structure" on page 353
- "a_term structure" on page 361
- "a_init_term_breaker structure" on page 357
- "a_text_source interface" on page 355
- "Term breaker entry point function" on page 365

a_term structure

The a_term structure stores a term, its length, and its position.

Syntax

```
typedef struct a_term {
   unsigned char   * term;
   a_sql_uint32    len;
   a_sql_uint32    ch_len;
   a_sql_uint64    pos;
} a_term;
```

Members

Member	Type	Description
term	`unsigned char *`	The term to be indexed.
len	`a_sql_uint32`	Length of the term, in bytes.
ch_len	`a_sql_uint32`	Length of the term, in characters.
pos	`a_sql_uint64`	Position of the term within the document. The database server expects that two immediately consecutive terms in a document have positions differing by 1. If the term breaker is performing its own stoplist processing, it is possible that the difference between two consecutive terms returned is more than 1; this is expected and acceptable. However, in other cases where numbers are not consecutive with positions differing by 1, the arbitrary positions can affect how full text queries are executed and can cause unexpected results for subsequent full text queries.

Remarks

Each a_term structure represents a term annotated with its byte length, character length, and its position in the document.

A pointer to an array of a_term elements is returned in the OUT parameter by the get_words method implemented as part of the a_word_source interface.

The a_term structure is defined by a header file named *exttbapiv1.h*, in the *SDK\Include* subdirectory of your SQL Anywhere installation directory.

extpf_use_new_api entry point function (prefilters)

Notifies the database server about the interface version implemented in the external prefilter library. Currently, only version 1 interfaces are supported.

This function is required for an external prefilter library.

Syntax

extern "C" a_sql_uint32 (extpf_use_new_api)();

Returns

The function returns an unsigned 32-bit integer. The returned value must be the interface version number, EXTPF_V1_API defined in *extpfapi1.h*.

Remarks

A typical implementation of this function is as follows:

```
extern "C" a_sql_uint32 ( extpf_use_new_api )( void )
{
   return EXTPF_V1_API;
}
```

exttb_use_new_api entry point function (term breakers)

Provides information about the interface version implemented in the external term breaker library. Currently, only version 1 interfaces are supported.

This function is required for an external term breaker library.

Syntax

extern "C" a_sql_uint32 (exttb_use_new_api)();

Returns

The function returns an unsigned 32-bit integer. The returned value must be the interface version number, EXTTB_V1_API defined in *exttbapi1.h*.

Remarks

A typical implementation of this function is as follows:

```
extern "C" a_sql_uint32 ( exttb_use_new_api )( void )
{
   return EXTTB_V1_API;
}
```

extfn_post_load_library global entry point function

If this function is implemented and exposed in the external library, it is executed by the database server after the external library has been loaded and the version check has been performed, and before any other function defined in the external library is called.

This function is required only if there is a library-specific requirement to do library-wide setup before any function within the library is called.

Syntax

extern "C" void (extfn_post_load_library)();

Remarks

Both external term breaker and prefilter libraries can implement this function.

extfn_pre_unload_library global entry point function

If this function is implemented and exposed in the external library, it is executed by the database server immediately before unloading the external library.

This function is required only if there is a library-specific requirement to do library-wide cleanup before the library is unloaded.

Syntax

extern "C" void (extfn_pre_unload_library)();

Remarks

Both external term breaker and prefilter libraries can implement this function.

Prefilter entry point function

Entry point function that initializes an instance of an external prefilter and negotiates the character set of the data.

Syntax

extern "C" a_sql_uint32 (SQL_CALLBACK **entry-point-function*)(a_init_pre_filter **data*);

Returns

1 on error and 0 on successful execution

Parameters

- **entry-point-function** The name of the entry point function for the prefilter.
- **data** A pointer to an a_init_pre_filter structure. See "a_init_pre_filter structure" on page 354.

Remarks

This function must be implemented in the external prefilter library, and needs to be re-entrant as it can be executed on multiple threads simultaneously.

The caller of the function (database server) provides a pointer to an a_text_source object that will serve as the producer for the prefilter. The caller also provides the character set of the input.

This function provides a pointer to the external prefilter (a_text_source structure). It also negotiates the character set of the input (if it is not binary) and output data by changing the actual_charset field, if necessary.

Note that if desired_charset and actual_charset are not the same, the database server will perform character set conversion on the input data, unless `data->is_binary` field is 1. This means that if `is_binary` is 0, input data will be in the character specified by `actual_charset`.

Note that requiring character set conversion can cause a degradation in performance.

This entry point function is specified by the user by calling ALTER TEXT CONFIGURATION...PREFILTER EXTERNAL NAME. See "PREFILTER EXTERNAL NAME clause, ALTER TEXT CONFIGURATION statement" [*SQL Anywhere Server - SQL Reference*].

See also

- "a_init_pre_filter structure" on page 354
- "a_text_source interface" on page 355
- "PREFILTER EXTERNAL NAME clause, ALTER TEXT CONFIGURATION statement" [*SQL Anywhere Server - SQL Reference*]

Term breaker entry point function

Entry point function that initializes an instance of an external term breaker and negotiates the character set of the data.

Syntax

extern "C" a_sql_uint32 (SQL_CALLBACK *_entry-point-function_)(a_init_term_breaker *_data_);

Returns

1 on error and 0 on successful execution

Parameters

- **entry-point-function** The name of the entry point function for the term breaker.
- **data** A pointer to an a_init_term_breaker structure. See "a_init_term_breaker structure" on page 357.

Remarks

This function must be implemented in the external term breaker library, and needs to be re-entrant as it can be executed on multiple threads simultaneously.

The caller of the function provides a pointer to an a_text_source object that will serve as the producer for the term breaker. The caller should also provide the character set of the input.

This function provides to the caller a pointer to an external term breaker (a_word_source structure) and the supported character set.

If desired_charset and actual_charset are not the same, the database server converts the term breaker input to the character set specified by actual_charset.

Note that character set conversion can cause a degradation in performance.

See also

- “a_word_source interface” on page 359
- “a_text_source interface” on page 355
- “a_init_term_breaker structure” on page 357

Summarizing, grouping, and sorting query results

Summarizing query results using aggregate functions

Aggregate functions display summaries of the values in specified columns. You can also use the GROUP BY clause, HAVING clause, and ORDER BY clause to group and sort the results of queries using aggregate functions, and the UNION operator to combine the results of queries.

When an ORDER BY clause contains constants, they are interpreted by the optimizer and then replaced by an equivalent ORDER BY clause. For example, the optimizer interprets ORDER BY 'a' as ORDER BY expression.

A query block containing more than one aggregate function with valid ORDER BY clauses can be executed if the ORDER BY clauses can be logically combined into a single ORDER BY clause. For example, the following clauses:

```
ORDER BY expression1, 'a', expression2
```

```
ORDER BY expression1, 'b', expression2, 'c', expression3
```

are subsumed by the clause:

```
ORDER BY expression1, expression2, expression3
```

You can apply aggregate functions to all the rows in a table, to a subset of the table specified by a WHERE clause, or to one or more groups of rows in the table. From each set of rows to which an aggregate function is applied, SQL Anywhere generates a single value.

The following are some of the supported aggregate functions:

- **AVG(expression)** The mean of the supplied expression over the returned rows.
- **COUNT(expression)** The number of rows in the supplied group where the expression is not NULL.
- **COUNT(*)** The number of rows in each group.
- **LIST(string-expr)** A string containing a comma-separated list composed of all the values for *string-expr* in each group of rows.
- **MAX(expression)** The maximum value of the expression, over the returned rows.
- **MIN(expression)** The minimum value of the expression, over the returned rows.
- **STDDEV(expression)** The standard deviation of the expression, over the returned rows.

- **SUM(expression)** The sum of the expression, over the returned rows.
- **VARIANCE(expression)** The variance of the expression, over the returned rows.

For a complete list of aggregate functions, see "Aggregate functions" [*SQL Anywhere Server - SQL Reference*].

You can use the optional keyword DISTINCT with AVG, SUM, LIST, and COUNT to eliminate duplicate values before the aggregate function is applied.

The expression to which the syntax statement refers is usually a column name. It can also be a more general expression.

For example, with this statement you can find what the average price of all products would be if one dollar were added to each price:

```
SELECT AVG ( UnitPrice + 1 )
FROM Products;
```

Example

The following query calculates the total payroll from the annual salaries in the Employees table:

```
SELECT SUM( Salary )
FROM Employees;
```

To use aggregate functions, you must give the function name followed by an expression on whose values it will operate. The expression, which is the Salary column in this example, is the function's argument and must be specified inside parentheses.

Where you can use aggregate functions

The aggregate functions can be used in a select list, as in the previous examples, or in the HAVING clause of a select statement that includes a GROUP BY clause. See "The HAVING clause: selecting groups of data" on page 376.

You cannot use aggregate functions in a WHERE clause or in a JOIN condition. However, a SELECT statement with aggregate functions in its select list often includes a WHERE clause that restricts the rows to which the aggregate is applied.

If a SELECT statement includes a WHERE clause, but not a GROUP BY clause, an aggregate function produces a single value for the subset of rows that the WHERE clause specifies.

Whenever an aggregate function is used in a SELECT statement that does not include a GROUP BY clause, it produces a single value. This is true whether it is operating on all the rows in a table or on a subset of rows defined by a where clause.

You can use more than one aggregate function in the same select list, and produce more than one scalar aggregate in a single SELECT statement.

Aggregate functions and outer references

SQL Anywhere follows SQL/2008 standards for clarifying the use of aggregate functions when they appear in a subquery. These changes affect the behavior of statements written for previous versions of the software: previously correct queries may now produce error messages, and result sets may change.

When an aggregate function appears in a subquery, and the column referenced by the aggregate function is an outer reference, the entire aggregate function itself is now treated as an outer reference. This means that the aggregate function is now computed in the outer block, not in the subquery, and becomes a constant within the subquery.

The following restrictions apply to the use of outer reference aggregate functions in subqueries:

- The outer reference aggregate function can only appear in subqueries that are in the SELECT list or HAVING clause, and these clauses must be in the immediate outer block.
- Outer reference aggregate functions can only contain one outer column reference.
- Local column references and outer column references cannot be mixed in the same aggregate function.

Some problems related to the new standards can be circumvented by rewriting the aggregate function so that it only includes local references. For example, the subquery `(SELECT MAX(S.y + R.y) FROM S)` contains both a local column reference (S.y) and an outer column reference (R.y), which is now illegal. It can be rewritten as `(SELECT MAX(S.y) + R.y FROM S)`. In the rewrite, the aggregate function has only a local column reference. The same sort of rewrite can be used when an outer reference aggregate function appears in clauses other than SELECT or HAVING.

Example

The following query produced valid results in earlier versions of SQL Anywhere:

```
SELECT Name,
       ( SELECT SUM( p.Quantity )
             FROM SalesOrderItems )
   FROM Products p;
```

Name	SUM(p.Quantity)
Tee shirt	30,716
Tee shirt	59,238

In later versions, the same query produces the error message `SQL Anywhere Error -149: Function or column reference to 'name' must also appear in a GROUP BY.` The reason that the statement is no longer valid is that the outer reference aggregate function `sum(p.Quantity)` is now computed in the outer block. In later versions, the query is semantically equivalent to the following (except that Z does not appear as part of the result set):

```
SELECT Name,
       SUM( p.Quantity ) AS Z,
       ( SELECT Z
             FROM SalesOrderItems )
   FROM Products p;
```


Since the outer block now computes an aggregate function, the outer block is treated as a grouped query and column name must appear in a GROUP BY clause to appear in the SELECT list.

Aggregate functions and data types

Some aggregate functions have meaning only for certain kinds of data. For example, you can use SUM and AVG with numeric columns only.

However, you can use MIN to find the lowest value—the one closest to the beginning of the alphabet—in a character column:

```
SELECT MIN( Surname )
   FROM Contacts;
```

Using COUNT(*)

COUNT(*) returns the number of rows in the specified table without eliminating duplicates. It counts each row separately, including rows that contain NULL. This function does not require an expression as an argument because, by definition, it does not use information about any particular column.

The following statement finds the total number of employees in the Employees table:

```
SELECT COUNT(*)
   FROM Employees;
```

Like other aggregate functions, you can combine COUNT(*) with other aggregate functions in the select list, with WHERE clauses, and so on. For example:

```
SELECT COUNT(*), AVG( UnitPrice )
   FROM Products
   WHERE UnitPrice > 10;
```

COUNT(*)	AVG(Products.UnitPrice)
5	18.2

Using aggregate functions with DISTINCT

The DISTINCT keyword is optional with SUM, AVG, and COUNT. When you use DISTINCT, duplicate values are eliminated before calculating the sum, average, or count. For example, to find the number of different cities in which there are contacts, execute the following statement:

```
SELECT COUNT( DISTINCT City )
   FROM Contacts;
```

COUNT(DISTINCT Contacts.City)
16

You can use more than one aggregate function with DISTINCT in a query. Each DISTINCT is evaluated independently. For example:

```
SELECT COUNT( DISTINCT GivenName ) "first names",
       COUNT( DISTINCT Surname ) "last names"
   FROM Contacts;
```

first names	last names
48	60

Aggregate functions and NULL

Any NULLS in the column on which the aggregate function is operating are ignored for the function except COUNT(*), which includes them. If all the values in a column are NULL, COUNT(column_name) returns 0.

If no rows meet the conditions specified in the WHERE clause, COUNT returns a value of 0. The other functions all return NULL. Here are examples:

```
SELECT COUNT( DISTINCT Name )
   FROM Products
   WHERE UnitPrice > 50;
```

COUNT(DISTINCT Name)
0

```
SELECT AVG( UnitPrice )
   FROM Products
   WHERE UnitPrice > 50;
```

AVG(Products.UnitPrice)
(NULL)

The GROUP BY clause: Organizing query results into groups

The GROUP BY clause divides the output of a table into groups. You can group rows by one or more column names, or by the results of computed columns.

Order of clauses
If a WHERE clause and a GROUP BY clause are present, the WHERE clause must appear before the GROUP BY clause. A GROUP BY clause, if present, must always appear before a HAVING clause. If a HAVING clause is specified but a GROUP BY clause is not, a GROUP BY () clause is assumed.

HAVING clauses and WHERE clauses can both be used in a single query. Conditions in the HAVING clause logically restrict the rows of the result only after the groups have been constructed. Criteria in the WHERE clause are logically evaluated before the groups are constructed, and so save time.

Understanding which queries are valid and which are not can be difficult when the query involves a GROUP BY clause. This section describes a way to think about queries with GROUP BY so that you may understand the results and the validity of queries better.

How queries with GROUP BY are executed

This section uses the ROLLUP sub-clause of the GROUP BY clause in the explanation and example. For more information on the ROLLUP clause, see "Using ROLLUP and CUBE as a shortcut to GROUPING SETS" on page 450.

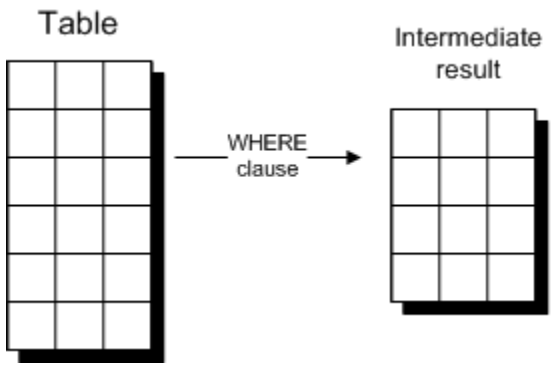
Consider a single-table query of the following form:

SELECT *select-list*
  FROM *table*
  WHERE *where-search-condition*
  GROUP BY [*group-by-expression* | **ROLLUP** *(group-by-expression)*]
  HAVING *having-search-condition*

This query is executed in the following manner:

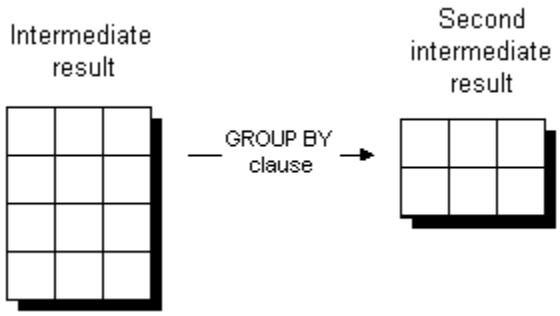
1. **Apply the WHERE clause**

   This generates an intermediate result that contains only some of the rows of the table.



2. **Partition the result into groups**

This action generates an intermediate result with one row for each group as dictated by the GROUP BY clause. Each generated row contains the *group-by-expression* for each group, and the computed aggregate functions in the *select-list* and *having-search-condition*.



3. **Apply any ROLLUP operation**

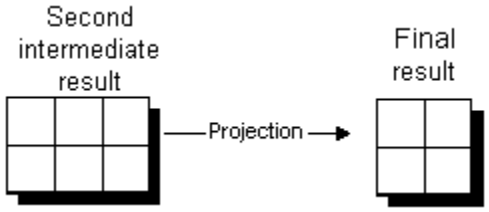
   Subtotal rows computed as part of a ROLLUP operation are added to the result set.

4. **Apply the HAVING clause**

   Any rows from this second intermediate result that do not meet the criteria of the HAVING clause are removed at this point.

5. **Project out the results to display**

   This action takes from step 3 only those columns that need to be displayed in the result set of the query —that is, it takes only those columns corresponding to the expressions from the *select-list*.



This process makes requirements on queries with a GROUP BY clause:

- The WHERE clause is evaluated first. Therefore, any aggregate functions are evaluated only over those rows that satisfy the WHERE clause.

- The final result set is built from the second intermediate result, which holds the partitioned rows. The second intermediate result holds rows corresponding to the *group-by-expression*. Therefore, if an expression that is not an aggregate function appears in the *select-list*, then it must also appear in the *group-by-expression*. No function evaluation can be performed during the projection step.

- An expression can be included in the *group-by-expression* but not in the *select-list*. It is projected out in the result.

Using GROUP BY with multiple columns

You can list more than one expression in the GROUP BY clause—that is, you can group a table by any combination of expressions.

The following query lists the average price of products, grouped first by name and then by size:

```
SELECT Name, Size, AVG( UnitPrice )
   FROM Products
   GROUP BY Name, Size;
```

Name	Size	AVG(Products.UnitPrice)
Baseball Cap	One size fits all	9.5
Sweatshirt	Large	24
Tee Shirt	Large	14
Tee Shirt	One size fits all	14
...	...	...

WHERE clause and GROUP BY

You can use a WHERE clause in a statement with GROUP BY. The WHERE clause is evaluated before the GROUP BY clause. Rows that do not satisfy the conditions in the WHERE clause are eliminated before any grouping is done. Here is an example:

```
SELECT  Name, AVG( UnitPrice )
   FROM Products
   WHERE ID > 400
   GROUP BY Name;
```

Only the rows with ID values of more than 400 are included in the groups that are used to produce the query results.

Example

The following query illustrates the use of WHERE, GROUP BY, and HAVING clauses in one query:

```
SELECT Name, SUM( Quantity )
   FROM Products
   WHERE Name LIKE '%shirt%'
   GROUP BY Name
   HAVING SUM( Quantity ) > 100;
```

Name	SUM(Products.Quantity)
Tee Shirt	157

In this example:

- The WHERE clause includes only rows that have a name including the word *shirt* (Tee Shirt, Sweatshirt).
- The GROUP BY clause collects the rows with a common name.
- The SUM aggregate calculates the total quantity of products available for each group.
- The HAVING clause excludes from the final results the groups whose inventory totals do not exceed 100.

Using GROUP BY with aggregate functions

A GROUP BY clause almost always appears in statements that include aggregate functions, in which case the aggregate produces a value for each group. These values are called **vector aggregates**. (A **scalar aggregate** is a single value produced by an aggregate function without a GROUP BY clause.)

Example

The following query lists the average price of each kind of product:

```
SELECT Name, AVG( UnitPrice ) AS Price
   FROM Products
   GROUP BY Name;
```

Name	Price
Tee Shirt	12.333333333
Baseball Cap	9.5
Visor	7
Sweatshirt	24
...	...

The vector aggregates produced by SELECT statements with aggregates and a GROUP BY appear as columns in each row of the results. By contrast, the scalar aggregates produced by queries with aggregates and no GROUP BY also appear as columns, but with only one row. For example:

```
SELECT AVG( UnitPrice )
   FROM Products;
```


AVG(Products.UnitPrice)
13.3

GROUP BY and the SQL/2008 standard

The SQL/2008 standard is considerably more restrictive in its syntax than what is supported by SQL Anywhere. In the SQL/2008 standard, GROUP BY requires the following:

- Each *group-by-term* specified in a GROUP BY clause must be a *column reference*: that is, a reference to a column from a table referenced in the query FROM clause. These expressions are termed **grouping columns**.
- An expression in a SELECT list, HAVING clause, or ORDER BY clause that is not an aggregate function must be a grouping column, or only reference grouping columns. However, if optional SQL/2008 language feature T301, "Functional dependencies" is supported, then such a reference can refer to columns from the query FROM clause that are functionally determined by grouping columns.

In a GROUP BY clause in SQL Anywhere, *group-by-term* can be an arbitrary expression involving column references, literal constants, variables or host variables, and scalar and user-defined functions. For example, this query partitions the Employee table into three groups based on the Salary column, producing one row per group:

```
SELECT COUNT() FROM Employees
   GROUP BY (
      IF SALARY < 25000
         THEN 'low range'
      ELSE IF Salary < 50000
         THEN 'mid range'
      ELSE 'high range'
         ENDIF
      ENDIF);
```

To include the partitioning value in the query result, you must add a *group-by-term* to the query SELECT list. To be syntactically valid, SQL Anywhere ensures that the syntax of the SELECT list item and *group-by-term* are identical. However, syntactically large SQL constructions may fail this analysis; moreover, expressions involving subqueries never compare equal.

In the example below, SQL Anywhere detects that the two IF expressions are identical, and computes the result without error:

```
SELECT (IF SALARY < 25000 THEN 'low range' ELSE IF Salary < 50000 THEN 'mid
range' ELSE 'high range' ENDIF ENDIF), COUNT()
FROM Employees
GROUP BY (IF SALARY < 25000 THEN 'low range' ELSE IF Salary < 50000 THEN 'mid
range' ELSE 'high range' ENDIF ENDIF);
```

However, this query contains a subquery in the GROUP BY clause that returns an error:

```
SELECT (Select State from Employees e WHERE e.EmployeeID = e2.EmployeeID),
      COUNT()
```

```
    FROM Employees e2
    GROUP BY (Select State from Employees e WHERE EmployeeID = e2.EmployeeID)
```

A more concise approach is to alias the SELECT list expression, and refer to the alias in the GROUP BY clause. Using an alias permits the SELECT list and the GROUP BY clause to contain correlated subqueries. SELECT list aliases used in this fashion are a vendor extension:

```
SELECT (
   IF SALARY < 25000
      THEN 'low range'
   ELSE IF Salary < 50000
      THEN 'mid range'
   ELSE 'high range'
      ENDIF
   ENDIF) AS Salary_range,
   COUNT() FROM Employees GROUP BY Salary_Range;
```

While SQL Anywhere does not support all facets of SQL/2008 language feature T301 (Functional dependencies), SQL Anywhere does offer some support for derived values based on GROUP BY terms. SQL Anywhere supports SELECT list expressions that refer to GROUP BY terms, literal constants, and (host) variables, with or without scalar functions that may modify those values. As an example, the following query lists the number of employees by city/state combination:

```
SELECT City || ' ' || State, SUBSTRING(City,1,3), COUNT()
FROM Employees
GROUP BY City, State
```

For additional standards compliance information about using SQL Anywhere aggregate functions, see "Aggregate functions and outer references" on page 368.

See also

- "GROUP BY clause" [*SQL Anywhere Server - SQL Reference*]

The HAVING clause: selecting groups of data

The HAVING clause restricts the rows returned by a query. It sets conditions for the GROUP BY clause similar to the way in which WHERE sets conditions for the SELECT clause.

The HAVING clause search conditions are identical to WHERE search conditions except that WHERE search conditions cannot include aggregates. For example, the following usage is allowed:

```
HAVING AVG( UnitPrice ) > 20
```

The following usage is not allowed:

```
WHERE AVG( UnitPrice ) > 20
```

Using HAVING with aggregate functions

The following statement is an example of simple use of the HAVING clause with an aggregate function.

To list those products available in more than one size or color, you need a query to group the rows in the Products table by name, but eliminate the groups that include only one distinct product:


```
SELECT Name
   FROM Products
   GROUP BY Name
   HAVING COUNT( * ) > 1;
```

Name
Tee Shirt
Baseball Cap
Visor
Sweatshirt

For information about when you can use aggregate functions in HAVING clauses, see "Where you can use aggregate functions" on page 367.

Using HAVING without aggregate functions

The HAVING clause can also be used without aggregates.

The following query groups the products, and then restricts the result set to only those groups for which the name starts with B.

```
SELECT Name
   FROM Products
   GROUP BY Name
   HAVING Name LIKE 'B%';
```

Name
Baseball Cap

More than one condition in HAVING

More than one search condition can be included in the HAVING clause. They are combined with the AND, OR, or NOT operators, as in the following example.

To list those products available in more than one size or color, for which one version costs more than $10, you need a query to group the rows in the Products table by name, but eliminate the groups that include only one distinct product, and eliminate those groups for which the maximum unit price is under $10.

```
SELECT Name
   FROM Products
   GROUP BY Name
   HAVING COUNT( * ) > 1
   AND MAX( UnitPrice ) > 10;
```

Name
Tee Shirt

Name
Sweatshirt

The ORDER BY clause: sorting query results

The ORDER BY clause allows sorting of query results by one or more columns. Each sort can be ascending (ASC) or descending (DESC). If neither is specified, ASC is assumed.

A simple example

The following query returns results ordered by name:

```
SELECT ID, Name
   FROM Products
   ORDER BY Name;
```

ID	Name
400	Baseball Cap
401	Baseball Cap
700	Shorts
600	Sweatshirt
...	...

Sorting by more than one column

If you name more than one column in the ORDER BY clause, the sorts are nested.

The following statement sorts the shirts in the Products table first by name in ascending order, then by Quantity (descending) within each name:

```
SELECT ID, Name, Quantity
   FROM Products
   WHERE Name like '%shirt%'
   ORDER BY Name, Quantity DESC;
```

ID	Name	Quantity
600	Sweatshirt	39
601	Sweatshirt	32
302	Tee Shirt	75

ID	Name	Quantity
301	Tee Shirt	54
...	...	...

Using the column position

You can use the position number of a column in a select list instead of the column name. Column names and select list numbers can be mixed. Both of the following statements produce the same results as the preceding one.

```
SELECT ID, Name, Quantity
   FROM Products
   WHERE Name like '%shirt%'
   ORDER BY 2, 3 DESC;
SELECT ID, Name, Quantity
   FROM Products
   WHERE Name like '%shirt%'
   ORDER BY 2, Quantity DESC
```

Most versions of SQL require that ORDER BY items appear in the select list, but SQL Anywhere has no such restriction. The following query orders the results by Quantity, although that column does not appear in the select list:

```
SELECT ID, Name
   FROM Products
   WHERE Name like '%shirt%'
   ORDER BY 2, Quantity DESC;
```

ORDER BY and NULL

With ORDER BY, NULL sorts before all other values in ascending sort order.

ORDER BY and case sensitivity

The effects of an ORDER BY clause on mixed-case data depend on the database collation and case sensitivity specified when the database is created.

Explicitly limiting the number of rows returned by a query

You can use the FIRST or TOP keywords to limit the number of rows included in the result set of a query. These keywords are for use with queries that include an ORDER BY clause.

Examples

The following query returns information about the employee that appears first when employees are sorted by last name:

```
SELECT FIRST *
   FROM Employees
   ORDER BY Surname;
```

The following query returns the first five employees as sorted by last name:

```
SELECT TOP 5 *
   FROM Employees
   ORDER BY Surname;
```

When you use TOP, you can also use START AT to provide an offset. The following statement lists the fifth and sixth employees sorted in descending order by last name:

```
SELECT TOP 2 START AT 5 *
   FROM Employees
   ORDER BY Surname DESC;
```

FIRST and TOP should be used only in conjunction with an ORDER BY clause to ensure consistent results. Use of FIRST or TOP without an ORDER BY triggers a syntax warning, and will likely yield unpredictable results.

Note
The START AT value must be greater than 0. The TOP value must be greater than 0 when a constant and greater or equal to 0 when a variable.

ORDER BY and GROUP BY

You can use an ORDER BY clause to order the results of a GROUP BY in a particular way.

Example

The following query finds the average price of each product and orders the results by average price:

```
SELECT Name, AVG( UnitPrice )
   FROM Products
   GROUP BY Name
   ORDER BY AVG( UnitPrice );
```

Name	AVG(Products.UnitPrice)
Visor	7
Baseball Cap	9.5
Tee Shirt	12.333333333
Shorts	15
...	...

Performing set operations on query results with UNION, INTERSECT, and EXCEPT

The operators described in this section perform set operations on the results of two or more queries. While many of the operations can also be performed using operations in the WHERE clause or HAVING clause, there are some operations that are very difficult to perform in any way other than using these set-based operators. For example:

- When data is not normalized, you may want to assemble seemingly disparate information into a single result set, even though the tables are unrelated.
- NULL is treated differently by set operators than in the WHERE clause or HAVING clause. In the WHERE clause or HAVING clause, two null-containing rows with identical non-null entries are not seen as identical, as the two NULL values are not defined to be identical. The set operators see two such rows as the same.

See also

- "Set operators and NULL" on page 383
- "EXCEPT statement" [*SQL Anywhere Server - SQL Reference*]
- "INTERSECT statement" [*SQL Anywhere Server - SQL Reference*]
- "UNION statement" [*SQL Anywhere Server - SQL Reference*]

Combining sets with the UNION clause

The UNION operator combines the results of two or more queries into a single result set.

By default, the UNION operator removes duplicate rows from the result set. If you use the ALL option, duplicates are not removed. The columns in the final result set have the same names as the columns in the first result set. Any number of union operators may be used.

By default, a statement containing multiple UNION operators is evaluated from left to right. Parentheses may be used to specify the order of evaluation.

For example, the following two expressions are not equivalent, due to the way that duplicate rows are removed from result sets:

```
x UNION ALL ( y UNION z )
(x UNION ALL y) UNION z
```

In the first expression, duplicates are eliminated in the UNION between y and z. In the UNION between that set and x, duplicates are not eliminated. In the second expression, duplicates are included in the union between x and y, but are then eliminated in the subsequent union with z.

Using EXCEPT and INTERSECT

The EXCEPT clause lists the differences between two result sets. The following general construction lists all those rows that appear in the result set of query-1, but not in the result set of query-2.

query-1
EXCEPT
query-2

The INTERSECT clause lists the rows that appear in each of two result sets. The following general construction lists all those rows that appear in the result set of both query-1 and query-2.

query-1
INTERSECT
query-2

Like the UNION clause, both EXCEPT and INTERSECT take the ALL modifier, which prevents the elimination of duplicate rows from the result set.

See also

- "EXCEPT statement" [*SQL Anywhere Server - SQL Reference*]
- "INTERSECT statement" [*SQL Anywhere Server - SQL Reference*]

Rules for set operations

The following rules apply to UNION, EXCEPT, and INTERSECT statements:

- **Same number of items in the select lists** All select lists in the queries must have the same number of expressions (such as column names, arithmetic expressions, and aggregate functions). The following statement is invalid because the first select list is longer than the second:

```
SELECT store_id, city, state
   FROM stores
UNION
   SELECT store_id, city
     FROM stores_east;
```

- **Data types must match** Corresponding expressions in the SELECT lists must be of the same data type, or an implicit data conversion must be possible between the two data types, or an explicit conversion should be supplied.

  For example, a UNION, INTERSECT, or EXCEPT is not possible between a column of the CHAR data type and one of the INT data type, unless an explicit conversion is supplied. However, a set operation is possible between a column of the MONEY data type and one of the INT data type.

- **Column ordering** You must place corresponding expressions in the individual queries of a set operation in the same order, because the set operators compare the expressions one to one in the order given in the individual queries in the SELECT clauses.

- **Multiple set operations** You can string several set operations together, as in the following example:

```
SELECT City AS Cities
   FROM Contacts
   UNION
      SELECT City
         FROM Customers
   UNION
      SELECT City
         FROM Employees;
```


For UNION statements, the order of queries is not important. For INTERSECT, the order is important when there are two or more queries. For EXCEPT, the order is always important.

- **Column headings** The column names in the table resulting from a UNION are taken from the first individual query in the statement. If you want to define a new column heading for the result set, you can do so in the select list of the first query, as in the following example:

```
SELECT City AS Cities
   FROM Contacts
   UNION
      SELECT City
         FROM Customers;
```

In the following query, the column heading remains as City, as it is defined in the first query of the UNION clause.

```
SELECT City
   FROM Contacts
   UNION
      SELECT City AS Cities
         FROM Customers;
```

Alternatively, you can use the WITH clause to define the column names. For example:

```
WITH V( Cities )
AS ( SELECT City
     FROM Contacts
     UNION
        SELECT City
         FROM Customers )
SELECT * FROM V;
```

- **Ordering the results** You can use the WITH clause of the SELECT statement to order the column names in the select list. For example:

```
WITH V( CityName )
AS ( SELECT City AS Cities
     FROM Contacts
     UNION
        SELECT City
           FROM Customers )
SELECT * FROM V
   ORDER BY CityName;
```

Alternatively, you can use a single ORDER BY clause at the end of the list of queries, but you must use integers rather than column names, as in the following example:

```
SELECT City AS Cities
   FROM Contacts
   UNION
      SELECT City
        FROM Customers
   ORDER BY 1;
```

Set operators and NULL

NULL is treated differently by the set operators UNION, EXCEPT, and INTERSECT than it is in search conditions. This difference is one of the main reasons to use set operators.

When comparing rows, set operators treat NULL values as equal to each other. In contrast, when NULL is compared to NULL in a search condition the result is unknown (not true).

One result of this difference is that the number of rows in the result set for `query-1 EXCEPT ALL query-2` is *always* the difference in the number of rows in the result sets of the individual queries.

For example, consider two tables T1 and T2, each with the following columns:

```
col1 INT,
col2 CHAR(1)
```

The tables and data are set up as follows:

```
CREATE TABLE T1 (col1 INT, col2 CHAR(1));
CREATE TABLE T2 (col1 INT, col2 CHAR(1));
INSERT INTO T1 (col1, col2) VALUES(1, 'a');
INSERT INTO T1 (col1, col2) VALUES(2, 'b');
INSERT INTO T1 (col1) VALUES(3);
INSERT INTO T1 (col1) VALUES(3);
INSERT INTO T1 (col1) VALUES(4);
INSERT INTO T1 (col1) VALUES(4);
INSERT INTO T2 (col1, col2) VALUES(1, 'a');
INSERT INTO T2 (col1, col2) VALUES(2, 'x');
INSERT INTO T2 (col1) VALUES(3);
```

The data in the tables is as follows:

- Table T1.

col1	col2
1	a
2	b
3	(NULL)
3	(NULL)
4	(NULL)
4	(NULL)

- Table T2

col1	col2
1	a

col1	col2
2	x
3	(NULL)

One query that asks for rows in T1 that also appear in T2 is as follows:

```
SELECT T1.col1, T1.col2
   FROM T1 JOIN T2
   ON T1.col1 = T2.col1
   AND T1.col2 = T2.col2;
```

T1.col1	T1.col2
1	a

The row (3, NULL) does not appear in the result set, as the comparison between NULL and NULL is not true. In contrast, approaching the problem using the INTERSECT operator includes a row with NULL:

```
SELECT col1, col2
   FROM T1
   INTERSECT
      SELECT col1, col2
         FROM T2;
```

col1	col2
1	a
3	(NULL)

The following query uses search conditions to list rows in T1 that do not appear in T2:

```
SELECT col1, col2
   FROM T1
   WHERE col1 NOT IN (
       SELECT col1
         FROM T2
         WHERE T1.col2 = T2.col2 )
   OR col2 NOT IN (
       SELECT col2
         FROM T2
         WHERE T1.col1 = T2.col1 );
```

col1	col2
2	b
3	(NULL)
4	(NULL)

col1	col2
3	(NULL)
4	(NULL)

The NULL-containing rows from T1 are not excluded by the comparison. In contrast, approaching the problem using EXCEPT ALL excludes NULL-containing rows that appear in both tables. In this case, the (3, NULL) row in T2 is identified as the same as the (3, NULL) row in T1.

```
SELECT col1, col2
   FROM T1
   EXCEPT ALL
      SELECT col1, col2
         FROM T2;
```

col1	col2
2	b
3	(NULL)
4	(NULL)
4	(NULL)

The EXCEPT operator is more restrictive still. It eliminates both (3, NULL) rows from T1 and excludes one of the (4, NULL) rows as a duplicate.

```
SELECT col1, col2
   FROM T1
   EXCEPT
      SELECT col1, col2
         FROM T2;
```

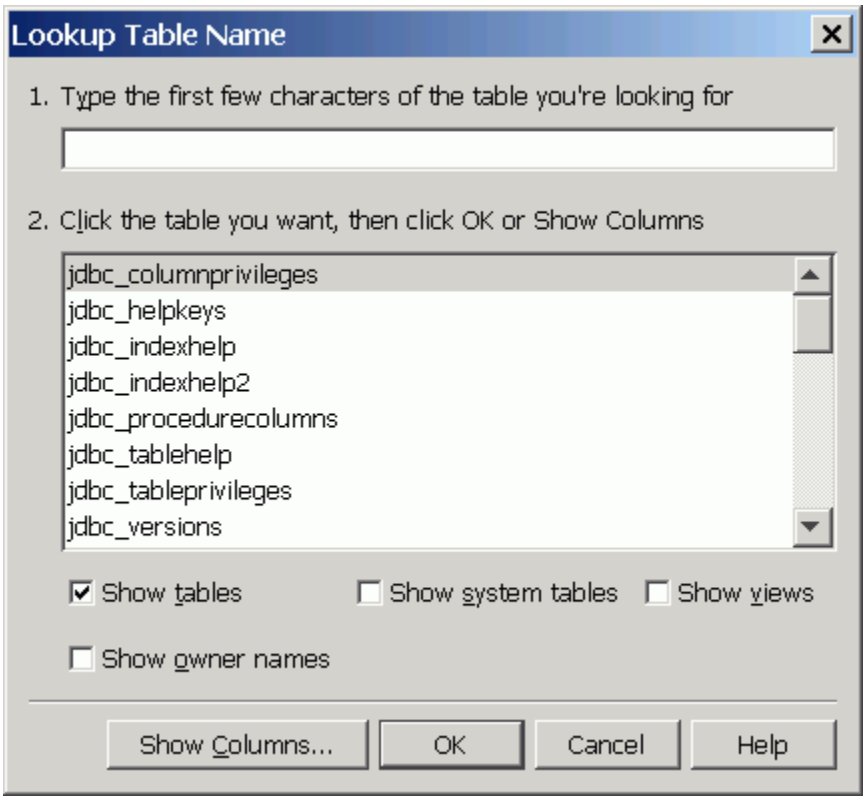
col1	col2
2	b
4	(NULL)

Joins: Retrieving data from several tables

When you create a database, you normalize the data by placing information specific to different objects in different tables, rather than in one large table with many redundant entries. Therefore, to retrieve related data from more than one table, you perform a join operation using the SQL JOIN operator. A join operation recreates a larger table using the information from two or more tables (or views). Using different joins, you can construct a variety of these virtual tables, each suited to a particular task.

Displaying a list of tables

In Interactive SQL, press F7 to display a list of tables in the database you are connected to.



Select a table and click **Show Columns** to see the columns for that table. Press Esc to return to the table list; press Esc again to return to the **SQL Statements** pane. Press Enter to copy the selected table or column name into the **SQL Statements** pane at the current cursor position.

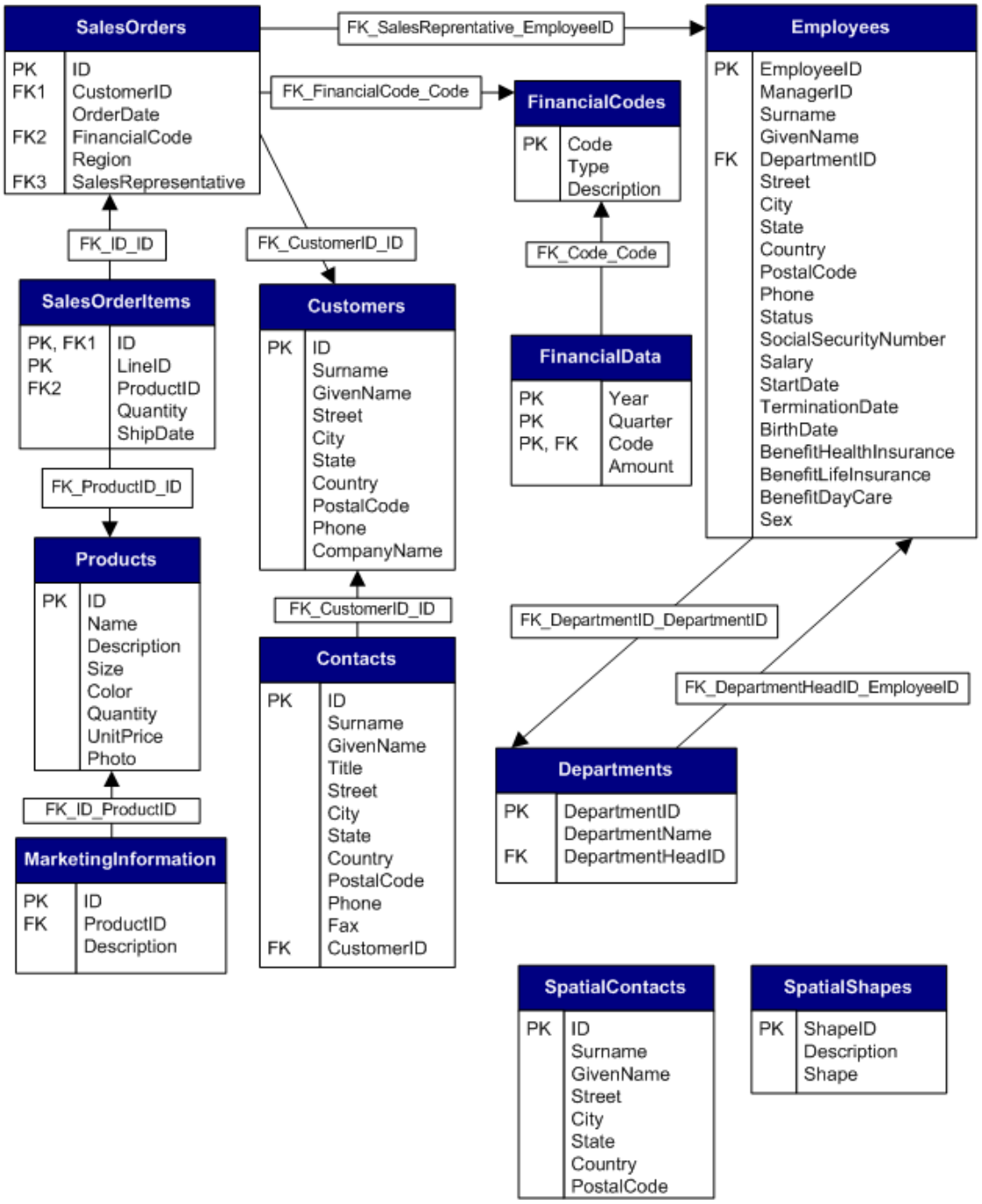
Press Esc to leave the list.

For more information about the tables in the SQL Anywhere sample database, see "Tutorial: Using the sample database" [*SQL Anywhere Server - Database Administration*].

Sample database schema

In the following diagram, the SQL Anywhere sample database is shown with the names of the foreign keys that relate the tables. These foreign key role names are required for some advanced joins.

For more information about role names, see "Key joins when there are multiple foreign key relationships" on page 418.



How joins work

A **join** is an operation that combines the rows in tables by comparing the values in specified columns. This section is an overview of SQL Anywhere join syntax.

A relational database stores information about different types of objects in different tables. For example, information particular to employees appears in one table, and information that pertains to departments in another. The Employees table contains information such as employee names and addresses. The Departments table contains information about one department, such as the name of the department and who the department head is.

Most questions can only be answered using a combination of information from different tables. For example, to answer the question "Who manages the Sales department?", you use the Departments table to identify the correct employee, and then look up the employee name in the Employees table.

Joins are a means of answering such questions by forming a new virtual table that includes information from multiple tables. For example, you could create a list of the department heads by combining the information contained in the Employees table and the Departments table. You specify which tables contain the information you need using the FROM clause.

To make the join useful, you must combine the correct columns of each table. To list department heads, each row of the combined table should contain the name of a department and the name of the employee who manages it. You control how columns are matched in the composite table by either specifying a particular type of join operation or using the ON clause.

See also

- "FROM clause" [*SQL Anywhere Server - SQL Reference*]

The FROM clause

Use the FROM clause to specify which base tables, temporary tables, views or derived tables to join. The FROM clause can be used in SELECT or UPDATE statements. An abbreviated syntax for the FROM clause is as follows:

FROM *table-expression*, ...

where:

table-expression :
table-name
| *view-name*
| *derived-table-name*
| *lateral-derived-table-name*
| *join-expression*
| **(** *table-expression*, … **)**
| *openstring-expression*
| *apply-expression*

table-name or *view-name*:
[*owner***.**] *table-or-view-name* [[**AS**] *correlation-name*]

derived-table-name :
(*select-statement* **)** [**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
| **FULL** [**OUTER**]
| **LEFT** [**OUTER**]
| **RIGHT** [**OUTER**]

apply-expression :
table-expression { **CROSS** | **OUTER** } **APPLY** *table-expression*

join-condition :

See "Search conditions" [*SQL Anywhere Server - SQL Reference*].

Notes

You cannot use an ON clause with CROSS JOIN.

For more syntax information, see "FROM clause" [*SQL Anywhere Server - SQL Reference*].

Join conditions

Tables can be joined using **join conditions**. A join condition is simply a search condition. It chooses a subset of rows from the joined tables based on the relationship between values in the columns. For example, the following query retrieves data from the Products and SalesOrderItems tables.

```
SELECT *
FROM Products JOIN SalesOrderItems
   ON Products.ID = SalesOrderItems.ProductID;
```

The join condition in this query is

```
Products.ID = SalesOrderItems.ProductID
```

This join condition means that rows can be combined in the result set only if they have the same product ID in both tables.

Join conditions can be explicit or generated. An **explicit join condition** is a join condition that is put in an ON clause or a WHERE clause. The following query uses an ON clause. It produces a cross product of the two tables (all combinations of rows), but with rows excluded if the ID numbers do not match. The result is a list of customers with details of their orders.

```
SELECT *
FROM Customers
```



```
JOIN SalesOrders
ON SalesOrders.CustomerID = Customers.ID;
```

A **generated join condition** is a join condition that is automatically created when you specify KEY JOIN or NATURAL JOIN. For key joins, the generated join condition is based on the foreign key relationships between the tables. For natural joins, the generated join condition is based on columns that have the same name.

Tip
Both key join syntax and natural join syntax are shortcuts: you get identical results from using the keyword JOIN without KEY or NATURAL, and then explicitly stating the same join condition in an ON clause.

When you use an ON clause with a key join or natural join, the join condition that is used is the **conjunction** of the explicitly specified join condition with the generated join condition. This means that the join conditions are combined with the keyword AND.

Joined tables

SQL Anywhere supports the following classes of joined tables.

- **CROSS JOIN** This type of join of two tables produces all possible combinations of rows from the two tables. The size of the result set is the number of rows in the first table multiplied by the number of rows in the second table. A cross join is also called a cross product or Cartesian product. You cannot use an ON clause with a cross join.
- **KEY JOIN** This type of join condition uses the foreign key relationships between the tables. Key join is the default when the JOIN keyword is used without specifying a join type (such a INNER, OUTER, and so on) and there is no ON clause.
- **NATURAL JOIN** This join is automatically generated based on columns having the same name.
- **Join using an ON clause** This type of join results from explicit specification of the join condition in an ON clause. When used with a key join or natural join, the join condition contains both the generated join condition and the explicit join condition. When used with the keyword JOIN without the keywords KEY or NATURAL, there is no generated join condition. See "Explicit join conditions (the ON clause)" on page 394.

Inner and outer joins

Key joins, natural joins and joins with an ON clause may be qualified by specifying INNER, LEFT OUTER, RIGHT OUTER, or FULL OUTER. The default is INNER. When using the keywords LEFT, RIGHT or FULL, the keyword OUTER is optional.

In an inner join, each row in the result satisfies the join condition.

In a left or right outer join, all rows are preserved for one of the tables, and for the other table nulls are returned for rows that do not satisfy the join condition. For example, in a right outer join the right side is preserved and the left side is null-supplying.

In a full outer join, all rows are preserved for both of the tables, and nulls are supplied for rows that do not satisfy the join condition.

Joining two tables

To understand how a simple inner join is computed, consider the following query. It answers the question: which product sizes have been ordered in the same quantity as the quantity in stock?

```
SELECT DISTINCT Name, Size,
    SalesOrderItems.Quantity
FROM Products JOIN SalesOrderItems
ON Products.ID = SalesOrderItems.ProductID
  AND Products.Quantity = SalesOrderItems.Quantity;
```

name	Size	Quantity
Baseball Cap	One size fits all	12
Visor	One size fits all	36

You can interpret the query as follows. Note that this is a conceptual explanation of the processing of this query, used to illustrate the semantics of a query involving a join. It does not represent how SQL Anywhere actually computes the result set.

- Create a cross product of the Products table and SalesOrderItems table. A cross product contains every combination of rows from the two tables.
- Exclude all rows where the product IDs are not identical (because of the join condition `Products.ID = SalesOrderItems.ProductID`).
- Exclude all rows where the quantity is not identical (because of the join condition `Products.Quantity = SalesOrderItems.Quantity`).
- Create a result table with three columns: Products.Name, Products.Size, and SalesOrderItems.Quantity.
- Exclude all duplicate rows (because of the DISTINCT keyword).

For a description of how outer joins are computed, see "Outer joins" on page 399.

Joining more than two tables

With SQL Anywhere, there is no fixed limit on the number of tables you can join.

When joining more than two tables, parentheses are optional. If you do not use parentheses, SQL Anywhere evaluates the statement from left to right. Therefore, `A JOIN B JOIN C` is equivalent to `( A JOIN B ) JOIN C`. Also, the following two SELECT statements are equivalent:


```
SELECT *
FROM A JOIN B JOIN C JOIN D;

SELECT *
FROM ( ( A JOIN B ) JOIN C ) JOIN D;
```

Whenever more than two tables are joined, the join involves table expressions. In the example `A JOIN B JOIN C`, the table expression `A JOIN B` is joined to C. This means, conceptually, that A and B are joined, and then the result is joined to C.

The order of joins is important if the table expression contains outer joins. For example, `A JOIN B LEFT OUTER JOIN C` is interpreted as `(A JOIN B) LEFT OUTER JOIN C`. This means that the table expression `A JOIN B` is joined to C. The table expression `A JOIN B` is preserved and table C is null-supplying.

For more information about outer joins, see "Outer joins" on page 399.

For more information about how SQL Anywhere performs a key join of table expressions, see "Key joins of table expressions" on page 421.

For more information about how SQL Anywhere performs a natural join of table expressions, see "Natural joins of table expressions" on page 415.

Join compatible data types

When you join two tables, the columns you compare must have the same or compatible data types.

For more information about data type conversion in joins, see "Comparisons between data types" [*SQL Anywhere Server - SQL Reference*].

Using joins in delete, update, and insert statements

You can use joins in DELETE, UPDATE, INSERT, and SELECT statements. You can update some cursors that contain joins if the ansi_update_constraints option is set to Off. This is the default for databases created before SQL Anywhere 7. For databases created with version 7 or later, the default is Cursors. See "ansi_update_constraints option" [*SQL Anywhere Server - Database Administration*].

Non-ANSI joins

SQL Anywhere supports ISO/ANSI standards for joins. It also supports the following non-standard joins:

- "Transact-SQL outer joins (*= or =*)" on page 403
- "Duplicate correlation names in joins (star joins)" on page 407
- "Key joins" on page 417

You can use the REWRITE function to see the ANSI equivalent of a non-ANSI join. See "REWRITE function [Miscellaneous]" [*SQL Anywhere Server - SQL Reference*].

Explicit join conditions (the ON clause)

Instead of, or along with, a key or natural join, you can specify a join using an explicit join condition. You specify a join condition by inserting an ON clause immediately after the join. The join condition always refers to the join immediately preceding it. The ON clause applies a restriction to the rows in a join, in much the same way that the WHERE clause applies restrictions to the rows of a query.

The ON clause allows you to construct more useful joins than the CROSS JOIN. For example, you can apply the ON clause to a join of the SalesOrders and Employees table to retrieve only those rows for which the SalesRepresentative in the SalesOrders table is the same as the one in the Employees table in every row of the result. Then each row contains information about an order and the sales representative responsible for it.

For example, in the following query, the first ON clause is used to join SalesOrders to Customers. The second ON clause is used to join the table expression (SalesOrders JOIN Customers) to the base table SalesOrderItems.

```
SELECT *
FROM SalesOrders JOIN Customers
    ON SalesOrders.CustomerID = Customers.ID
  JOIN SalesOrderItems
    ON SalesOrderItems.ID = SalesOrders.ID;
```

Referencing tables in an ON clause

The tables that are referenced in an ON clause must be part of the join that the ON clause modifies. For example, the following is invalid:

```
FROM ( A KEY JOIN B ) JOIN ( C JOIN D ON A.x = C.x )
```

The problem is that the join condition `A.x = C.x` references table A, which is not part of the join it modifies (in this case, `C JOIN D`).

However, as of the ANSI/ISO standard SQL99 and Adaptive Server Anywhere 7.0, there is an exception to this rule: if you use commas between table expressions, an ON condition of a join can reference a table that precedes it syntactically in the FROM clause. Therefore, the following is valid:

```
FROM (A KEY JOIN B) , (C JOIN D ON A.x = C.x)
```

See also: "Key joins" on page 417.

For more information about commas, see "Commas" on page 398.

Example

The following example joins the SalesOrders table with the Employees table. Each row in the result reflects rows in the SalesOrders table where the value of the SalesRepresentative column matched the value of the EmployeeID column of the Employees table.

```
SELECT Employees.Surname, SalesOrders.ID, SalesOrders.OrderDate
FROM SalesOrders
JOIN Employees
ON SalesOrders.SalesRepresentative = Employees.EmployeeID;
```

Surname	ID	OrderDate
Chin	2008	4/2/2001
Chin	2020	3/4/2001
Chin	2032	7/5/2001
Chin	2044	7/15/2000
Chin	2056	4/15/2001
...	...	...

Following are some notes about this example:

- The results of this query contain only 648 rows (one for each row in the SalesOrders table). Of the 48,600 rows in the cross product, only 648 of them have the employee number equal in the two tables.
- The ordering of the results has no meaning. You could add an ORDER BY clause to impose a particular order on the query.
- The ON clause includes columns that are not included in the final result set.

Generated joins and the ON clause

Key joins are the default if the keyword JOIN is used and no join type is specified—unless you use an ON clause. If you use an ON clause with an unspecified JOIN, key join is not the default and no generated join condition is applied.

For example, the following is a key join, because key join is the default when the keyword JOIN is used and there is no ON clause:

```
SELECT *
FROM A JOIN B;
```

The following is a join between table A and table B with the join condition `A.x = B.y`. It is not a key join.

```
SELECT *
FROM A JOIN B ON A.x = B.y;
```

If you specify a KEY JOIN or NATURAL JOIN and use an ON clause, the final join condition is the conjunction of the generated join condition and the explicit join condition(s). For example, the following statement has two join conditions: one generated because of the key join, and one explicitly stated in the ON clause.

```
SELECT *
FROM A KEY JOIN B ON A.x = B.y;
```

If the join condition generated by the key join is `A.w = B.z`, then the following statement is equivalent:

```
SELECT *
FROM A JOIN B
  ON A.x = B.y
  AND A.w = B.z;
```

For more information about key joins, see "Key joins" on page 417.

Types of explicit join conditions

Most join conditions are based on equality, and so are called **equijoins**. For example,

```
SELECT *
FROM Departments JOIN Employees
   ON Departments.DepartmentID = Employees.DepartmentID;
```

However, you do not have to use equality (=) in a join condition. You can use any search condition, such as conditions containing LIKE, SOUNDEX, BETWEEN, > (greater than), and != (not equal to).

Example

The following example answers the question: For which products has someone ordered more than the quantity in stock?

```
SELECT DISTINCT Products.Name
FROM Products JOIN SalesOrderItems
ON Products.ID = SalesOrderItems.ProductID
   AND SalesOrderItems.Quantity > Products.Quantity;
```

For more information about search conditions, see "Search conditions" [*SQL Anywhere Server - SQL Reference*].

Using the WHERE clause for join conditions

Except when using outer joins, you can specify join conditions in the WHERE clause instead of the ON clause. However, you should be aware that there may be semantic differences between the two if the query contains outer joins.

The ON clause is part of the FROM clause, and so is processed before the WHERE clause. This does not make a difference to results except for outer joins, where using the WHERE clause can convert the join to an inner join.

When deciding whether to put join conditions in an ON clause or WHERE clause, keep the following rules in mind:

- When you specify an outer join, putting a join condition in the WHERE clause may convert the outer join to an inner join.

  For more information about the WHERE clause and outer joins, see "Outer joins and join conditions" on page 400.

- Conditions in an ON clause can only refer to tables that are in the table expressions joined by the associated JOIN. However, conditions in a WHERE clause can refer to any tables, even if they are not part of the join.

- You cannot use an ON clause with the keywords CROSS JOIN, but you can always use a WHERE clause.

- When join conditions are in an ON clause, key join is not the default. However, key join can be the default if join conditions are put in a WHERE clause.

  For more information about the conditions under which key join is the default, see "Key joins" on page 417.

In the examples in this documentation, join conditions are put in an ON clause. In examples using outer joins, this is necessary. In other cases it is done to make it obvious that they are join conditions and not general search conditions.

Cross joins

A cross join of two tables produces all possible combinations of rows from the two tables. A cross join is also called a cross product or Cartesian product.

Each row of the first table appears once with each row of the second table. So, the number of rows in the result set is the product of the number of rows in the first table and the number of rows in the second table, minus any rows that are omitted because of restrictions in a WHERE clause.

You cannot use an ON clause with cross joins. However, you can put restrictions in a WHERE clause.

Inner and outer modifiers do not apply to cross joins

Except in the presence of additional restrictions in the WHERE clause, all rows of both tables always appear in the result set of cross joins. So, the keywords INNER, LEFT OUTER and RIGHT OUTER are not applicable to cross joins.

For example, the following statement joins two tables.

```
SELECT *
FROM A CROSS JOIN B;
```

The result set from this query includes all columns in A and all columns in B. There is one row in the result set for each combination of a row in A and a row in B. If A has *n* rows and B has *m* rows, the query returns *n* x *m* rows.

Commas

A comma works like a join operator, but is not one. A comma creates a cross product exactly as the keyword CROSS JOIN does. However, join keywords create table expressions, and commas create lists of table expressions.

In the following simple inner join of two tables, a comma and the keywords CROSS JOIN are equivalent:

```
SELECT *
FROM A CROSS JOIN B CROSS JOIN C
WHERE A.x = B.y;
```

and

```
SELECT *
FROM A, B, C
WHERE A.x = B.y;
```

Generally, you can use a comma instead of the keywords CROSS JOIN. The comma syntax is equivalent to cross join syntax, except for generated join conditions in table expressions using commas.

For information about how commas work with generated join conditions, see "Key joins of table expressions" on page 421.

In the syntax of star joins, commas have a special use. For more information, see "Duplicate correlation names in joins (star joins)" on page 407.

Inner and outer joins

The keywords INNER, LEFT OUTER, RIGHT OUTER, and FULL OUTER may be used to modify key joins, natural joins, and joins with an ON clause. The default is INNER. These modifiers do not apply to cross joins.

Inner joins

By default, joins are **inner joins**. This means that rows are included in the result set only if they satisfy the join condition.

Example

For example, each row of the result set of the following query contains the information from one Customers row and one SalesOrders row, satisfying the key join condition. If a particular customer has

placed no orders, the condition is not satisfied and the result set does not contain the row corresponding to that customer.

```
SELECT GivenName, Surname, OrderDate
FROM Customers KEY INNER JOIN SalesOrders
ORDER BY OrderDate;
```

GivenName	Surname	OrderDate
Hardy	Mums	2000-01-02
Aram	Najarian	2000-01-03
Tommie	Wooten	2000-01-03
Alfredo	Margolis	2000-01-06
...	...	...

Because inner joins and key joins are the defaults, you obtain the same results as above using the FROM clause as follows:

```
SELECT GivenName, Surname, OrderDate
FROM Customers JOIN SalesOrders
ORDER BY OrderDate;
```

Outer joins

Typically, you create joins that return rows only if they satisfy join conditions; these are called inner joins, and are the default join used when querying. However, sometimes you may want to preserve all the rows in one table. To do this, you use an **outer join**.

A left or right **outer join** of two tables preserves all the rows in one table, and supplies nulls for the other table when it does not meet the join condition. A **left outer join** preserves every row in the left-hand table, and a **right outer join** preserves every row in the right-hand table. In a **full outer join**, all rows from both tables are preserved and both tables are null-supplying.

The table expressions on either side of a left or right outer join are referred to as **preserved** and **null-supplying**. In a left outer join, the left-hand table expression is preserved and the right-hand table expression is null-supplying. In a full outer join both left-hand and right-hand table expressions are preserved and both are null-supplying.

For information about creating outer joins with Transact-SQL syntax, see “Transact-SQL outer joins (*= or =*)” on page 403.

See also: “Key joins” on page 417.

Example

The following statement includes all customers. If a particular customer has not placed an order, each column in the result that corresponds to order information contains the NULL value.

```
SELECT Surname, OrderDate, City
FROM Customers LEFT OUTER JOIN SalesOrders
   ON Customers.ID = SalesOrders.CustomerID
WHERE Customers.State = 'NY'
ORDER BY OrderDate;
```

Surname	OrderDate	City
Thompson	(NULL)	Bancroft
Reiser	2000-01-22	Rockwood
Clarke	2000-01-27	Rockwood
Mentary	2000-01-30	Rockland
...	...	...

You can interpret the outer join in this statement as follows. Note that this is a conceptual explanation, and does not represent how SQL Anywhere actually computes the result set.

- Return one row for every sales order placed by a customer. More than one row is returned when the customer placed two or more sales orders, because a row is returned for each sales order. This is the same result as an inner join. The ON condition is used to match customer and sales order rows. The WHERE clause is not used for this step.
- Include one row for every customer who has not placed any sales orders. This ensures that every row in the Customers table is included. For all these rows, the columns from SalesOrders are filled with nulls. These rows are added because the keyword OUTER is used, and would not have appeared in an inner join. Neither the ON condition nor the WHERE clause is used for this step.
- Exclude every row where the customer does not live in New York, using the WHERE clause.

Outer joins and join conditions

A common mistake with outer joins is the placement of the join condition. If you place restrictions on the null-supplying table in a WHERE clause, the join is usually equivalent to an inner join.

The reason for this is that most search conditions cannot evaluate to TRUE when any of their inputs are NULL. The WHERE clause restriction on the null-supplying table compares values to NULL, resulting in the elimination of the row from the result set. The rows in the preserved table are not preserved and so the join is an inner join.

The exception to this is comparisons that can evaluate to true when any of their inputs are NULL. These include IS NULL, IS UNKNOWN, IS FALSE, IS NOT TRUE, and expressions involving ISNULL or COALESCE.

Example

For example, the following statement computes a left outer join.


```
SELECT *
FROM Customers KEY LEFT OUTER JOIN SalesOrders
   ON SalesOrders.OrderDate < '2000-01-03';
```

In contrast, the following statement creates an inner join.

```
SELECT Surname, OrderDate
FROM Customers KEY LEFT OUTER JOIN SalesOrders
   WHERE SalesOrders.OrderDate < '2000-01-03';
```

The first of these two statements can be thought of as follows: First, left-outer join the Customers table to the SalesOrders table. The result set includes every row in the Customers table. For those customers who have no orders before January 3 2000, fill the sales order fields with nulls.

In the second statement, first left-outer join Customers and SalesOrders. The result set includes every row in the Customers table. For those customers who have no orders, fill the sales order fields with nulls. Next, apply the WHERE condition by selecting only those rows in which the customer has placed an order since January 3 2000. For those customers who have not placed orders, these values are NULL. Comparing any value to NULL evaluates to UNKNOWN. So, these rows are eliminated and the statement reduces to an inner join.

For more information about search conditions, see "Search conditions" [*SQL Anywhere Server - SQL Reference*].

Understanding complex outer joins

The order of joins is important when a query includes table expressions using outer joins. For example, `A JOIN B LEFT OUTER JOIN C` is interpreted as `(A JOIN B) LEFT OUTER JOIN C`. This means that the table expression `(A JOIN B)` is joined to C. The table expression `(A JOIN B)` is preserved and table C is null-supplying.

Consider the following statement, in which A, B and C are tables:

```
SELECT *
FROM A LEFT OUTER JOIN B RIGHT OUTER JOIN C;
```

To understand this statement, first remember that SQL Anywhere evaluates statements from left to right, adding parentheses. This results in

```
SELECT *
FROM (A LEFT OUTER JOIN B) RIGHT OUTER JOIN C;
```

Next, you may want to convert the right outer join to a left outer join so that both joins are the same type. To do this, simply reverse the position of the tables in the right outer join, resulting in:

```
SELECT *
FROM C LEFT OUTER JOIN (A LEFT OUTER JOIN B);
```

A is the preserved table and B is the null-supplying table for the nested outer join. C is the preserved table for the first outer join.

You can interpret this join as follows:

- Join A to B, preserving all rows in A.
- Next, join C to the results of the join of A and B, preserving all rows in C.

The join does not have an ON clause, and so is by default a key join. The way SQL Anywhere generates join conditions for this type of join is explained in "Key joins of table expressions that do not contain commas" on page 422.

In addition, the join condition for an outer join must only include tables that have previously been referenced in the FROM clause. This restriction is according to the ANSI/ISO standard, and is enforced to avoid ambiguity. For example, the following two statements are syntactically incorrect, because C is referenced in the join condition before the table itself is referenced.

```
SELECT *
FROM (A LEFT OUTER JOIN B ON B.x = C.x) JOIN C;
```

and

```
SELECT *
FROM A LEFT OUTER JOIN B ON A.x = C.x, C;
```

Outer joins of views and derived tables

Outer joins can also be specified for views and derived tables.

The statement

```
SELECT *
FROM V LEFT OUTER JOIN A ON (V.x = A.x);
```

can be interpreted as follows:

- Compute the view V.
- Join all the rows from the computed view V with A by preserving all the rows from V, using the join condition `V.x = A.x`.

Example

The following example defines a view called V that returns the employee IDs and department names of women who make over $60000.

```
CREATE VIEW V AS
SELECT Employees.EmployeeID, DepartmentName
  FROM Employees JOIN Departments
    ON Employees.DepartmentID = Departments.DepartmentID
  WHERE Sex = 'F' and Salary > 60000;
```

Next, use this view to add a list of the departments where the women work and the regions where they have sold. The view V is preserved and SalesOrders is null-supplying.

```
SELECT DISTINCT V.EmployeeID, Region, V.DepartmentName
  FROM V LEFT OUTER JOIN SalesOrders
    ON V.EmployeeID = SalesOrders.SalesRepresentative;
```


EmployeeID	Region	DepartmentName
243	(NULL)	R & D
316	(NULL)	R & D
529	(NULL)	R & D
902	Eastern	Sales
...	...	...

Transact-SQL outer joins (*= or =*)

Note
Support for the Transact-SQL outer join operators *= and =* is deprecated and will be removed in a future release.

In accordance with ANSI/ISO SQL standards, SQL Anywhere supports the LEFT OUTER, RIGHT OUTER, and FULL OUTER keywords. For compatibility with Adaptive Server Enterprise before version 12, SQL Anywhere also supports the Transact-SQL counterparts of these keywords, *= and =*, providing the tsql_outer_joins database option is set to On. See "tsql_outer_joins option" [*SQL Anywhere Server - Database Administration*].

There are some limitations and potential problems with the Transact-SQL semantics. For a detailed discussion of Transact-SQL outer joins, see the whitepaper "Semantics and Compatibility of Transact-SQL Outer Joins" at http://www.sybase.com/detail?id=1017447.

In the Transact-SQL dialect, you create outer joins by supplying a comma-separated list of tables in the FROM clause, and using the special operators *= or =* in the WHERE clause. In Adaptive Server Enterprise before version 12, the join condition must appear in the WHERE clause (ON was not supported).

Caution
When you are creating outer joins, do not mix *= syntax with ON clause syntax. This restriction also applies to views that are referenced in the query.

Example

The following left outer join lists all customers and finds their order dates (if any):

```
SELECT GivenName, Surname, OrderDate
FROM Customers, SalesOrders
WHERE Customers.ID *= SalesOrders.CustomerID
ORDER BY OrderDate;
```

This statement is equivalent to the following statement, in which ANSI/ISO syntax is used:

```
SELECT GivenName, Surname, OrderDate
FROM Customers LEFT OUTER JOIN SalesOrders
```

```
ON Customers.ID = SalesOrders.CustomerID
ORDER BY OrderDate;
```

Transact-SQL outer join limitations

Note
Support for Transact-SQL outer join operators *= and =* is deprecated and will be removed in a future release.

There are several restrictions for Transact-SQL outer joins:

- If you specify an outer join and a qualification on a column from the null-supplying table of the outer join, the results may not be what you expect. The qualification in the query does not exclude rows from the result set, but rather affects the values that appear in the rows of the result set. For rows that do not meet the qualification, a NULL value appears in the null-supplying table.

- You cannot mix ANSI/ISO SQL syntax and Transact-SQL outer join syntax in a single query. If a view is defined using one dialect for an outer join, you must use the same dialect for any outer-join queries on that view.

- A null-supplying table cannot participate in both a Transact-SQL outer join and a regular join or two outer joins. For example, the following WHERE clause is not allowed, because table S violates this limitation.

  ```
  WHERE R.x *= S.x
  AND S.y = T.y
  ```

  When you cannot rewrite your query to avoid using a table in both an outer join and a regular join clause, you must divide your statement into two separate queries, or use only ANSI/ISO SQL syntax.

- You cannot use a subquery that contains a join condition involving the null-supplying table of an outer join. For example, the following WHERE clause is not allowed:

  ```
  WHERE R.x *= S.y
  AND EXISTS ( SELECT *
               FROM T
               WHERE T.x = S.x )
  ```

Using views with Transact-SQL outer joins

If you define a view with an outer join, and then query the view with a qualification on a column from the null-supplying table of the outer join, the results may not be what you expect. The query returns all rows from the null-supplying table. Rows that do not meet the qualification show a NULL value in the appropriate columns of those rows.

The following rules determine what types of updates you can make to columns through views that contain outer joins:

- INSERT and DELETE statements are not allowed on outer join views.
- UPDATE statements are allowed on outer join views. If the view is defined WITH CHECK option, the update fails if any of the affected columns appears in the WHERE clause in an expression that includes columns from more than one table.

How NULL affects Transact-SQL joins

NULL values in tables or views being joined never match each other in a Transact-SQL outer join. The result of comparing a NULL value with any other NULL value is FALSE.

Specialized joins

This section describes how to create some specialized joins such as self-joins, star joins, and joins using derived tables.

Self-joins

In a **self-join**, a table is joined to itself by referring to the same table using a different correlation name.

Example 1

The following self-join produces a list of pairs of employees. Each employee name appears in combination with every employee name.

```
SELECT a.GivenName, a.Surname,
      b.GivenName, b.Surname
FROM Employees AS a CROSS JOIN Employees AS b;
```

GivenName	Surname	GivenName	Surname
Fran	Whitney	Fran	Whitney
Fran	Whitney	Matthew	Cobb
Fran	Whitney	Philip	Chin
Fran	Whitney	Julie	Jordan
...	...	...	...

Since the Employees table has 75 rows, this join contains 75 x 75 = 5625 rows. It includes, as well, rows that list each employee with themselves. For example, it contains the row

GivenName	Surname	GivenName	Surname
Fran	Whitney	Fran	Whitney

If you want to exclude rows that contain the same name twice, add the join condition that the employee IDs should not be equal to each other.

```
SELECT a.GivenName, a.Surname,
      b.GivenName, b.Surname
FROM Employees AS a CROSS JOIN Employees AS b
WHERE a.EmployeeID != b.EmployeeID;
```

Without these duplicate rows, the join contains 75 x 74 = 5550 rows.

This new join contains rows that pair each employee with every other employee, but because each pair of names can appear in two possible orders, each pair appears twice. For example, the result of the above join contains the following two rows.

GivenName	Surname	GivenName	Surname
Matthew	Cobb	Fran	Whitney
Fran	Whitney	Matthew	Cobb

If the order of the names is not important, you can produce a list of the (75 x 74)/2 = 2775 unique pairs.

```
SELECT a.GivenName, a.Surname,
      b.GivenName, b.Surname
FROM Employees AS a CROSS JOIN Employees AS b
WHERE a.EmployeeID < b.EmployeeID;
```

This statement eliminates duplicate lines by selecting only those rows in which the EmployeeID of employee a is less than that of employee b.

Example 2

The following self-join uses the correlation names report and manager to distinguish two instances of the Employees table, and creates a list of employees and their managers.

```
SELECT report.GivenName, report.Surname,
   manager.GivenName, manager.Surname
FROM Employees AS report JOIN Employees AS manager
   ON (report.ManagerID = manager.EmployeeID)
ORDER BY report.Surname, report.GivenName;
```

This statement produces the result shown partially below. The employee names appear in the two left-hand columns, and the names of their managers are on the right.

GivenName	Surname	GivenName	Surname
Alex	Ahmed	Scott	Evans

GivenName	Surname	GivenName	Surname
Joseph	Barker	Jose	Martinez
Irene	Barletta	Scott	Evans
Jeannette	Bertrand	Jose	Martinez
...	...	...	...

Duplicate correlation names in joins (star joins)

The reason for using duplicate table names is to create a **star join**. In a star join, one table or view is joined to several others.

To create a star join, you use the same table name, view name, or correlation name more than once in the FROM clause. This is an extension to the ANSI/ISO SQL standard. The ability to use duplicate names does not add any additional functionality, but it makes it easier to formulate certain queries.

The duplicate names must be in different joins for the syntax to make sense. When a table name or view name is used twice in the same join, the second instance is ignored. For example, `FROM A,A` and `FROM A CROSS JOIN A` are both interpreted as `FROM A`.

The following example, in which A, B and C are tables, is valid in SQL Anywhere. In this example, the same instance of table A is joined both to B and C. Note that a comma is required to separate the joins in a star join. The use of a comma in star joins is specific to the syntax of star joins.

```
SELECT *
FROM A LEFT OUTER JOIN B ON A.x = B.x,
     A LEFT OUTER JOIN C ON A.y = C.y;
```

The next example is equivalent.

```
SELECT *
FROM A LEFT OUTER JOIN B ON A.x = B.x,
     C RIGHT OUTER JOIN A ON A.y = C.y;
```

Both of these are equivalent to the following standard ANSI/ISO syntax. (The parentheses are optional.)

```
SELECT *
FROM (A LEFT OUTER JOIN B ON A.x = B.x)
LEFT OUTER JOIN C ON A.y = C.y;
```

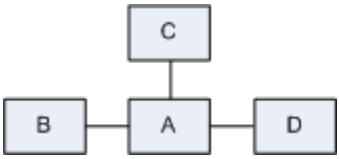
In the next example, table A is joined to three tables: B, C and D.

```
SELECT *
FROM A JOIN B ON A.x = B.x,
     A JOIN C ON A.y = C.y,
     A JOIN D ON A.w = D.w;
```

This is equivalent to the following standard ANSI/ISO syntax. (The parentheses are optional.)

```
SELECT *
FROM ((A JOIN B ON A.x = B.x)
JOIN C ON A.y = C.y)
JOIN D ON A.w = D.w;
```

With complex joins, it can help to draw a diagram. The previous example can be described by the following diagram, which illustrates that tables B, C and D are joined via table A.



Note
You can use duplicate table names only if the extended_join_syntax option is On (the default).

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

Example 1

Create a list of the names of the customers who placed orders with Rollin Overbey. Notice that one of the tables in the FROM clause, Employees, does not contribute any columns to the results. Nor do any of the columns that are joined—such as Customers.ID or Employees.EmployeeID—appear in the results. Nonetheless, this join is possible only using the Employees table in the FROM clause.

```
SELECT Customers.GivenName, Customers.Surname,
   SalesOrders.OrderDate
FROM   SalesOrders KEY JOIN Customers,
      SalesOrders KEY JOIN Employees
WHERE Employees.GivenName = 'Rollin'
  AND Employees.Surname = 'Overbey'
ORDER BY SalesOrders.OrderDate;
```

GivenName	Surname	OrderDate
Tommie	Wooten	2000-01-03
Michael	Agliori	2000-01-08
Salton	Pepper	2000-01-17
Tommie	Wooten	2000-01-23
...	...	...

Following is the equivalent statement in standard ANSI/ISO syntax:

```
SELECT Customers.GivenName, Customers.Surname,
  SalesOrders.OrderDate
FROM SalesOrders JOIN Customers
  ON SalesOrders.CustomerID =
```



```
   Customers.ID
JOIN Employees
  ON SalesOrders.SalesRepresentative =
   Employees.EmployeeID
WHERE Employees.GivenName = 'Rollin'
  AND Employees.Surname = 'Overbey'
ORDER BY SalesOrders.OrderDate;
```

Example 2

This example answers the question: How much of each product has each customer ordered, and who is the manager of the salesperson who took the order?

To answer the question, start by listing the information you need to retrieve. In this case, it is product, quantity, customer name, and manager name. Next, list the tables that hold this information. They are Products, SalesOrderItems, Customers, and Employees. When you look at the structure of the SQL Anywhere sample database, you see that these tables are all related through the SalesOrders table. You can create a star join on the SalesOrders table to retrieve the information from the other tables. See "Sample database schema" on page 387.

In addition, you need to create a self-join to get the name of the manager, because the Employees table contains ID numbers for managers and the names of all employees, but not a column listing only manager names. For more information, see "Self-joins" on page 405.

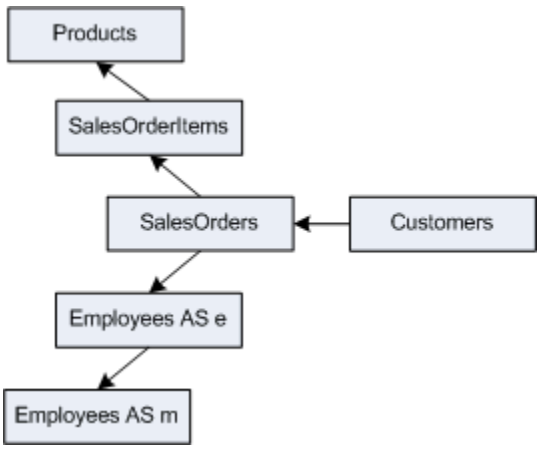
The following statement creates a star join around the SalesOrders table. The joins are all outer joins so that the result set will include all customers. Some customers have not placed orders, so the other values for these customers are NULL. The columns in the result set are Customers, Products, Quantity ordered, and the name of the manager of the salesperson.

```
SELECT Customers.GivenName, Products.Name,
  SUM(SalesOrderItems.Quantity), m.GivenName
FROM SalesOrders
    KEY RIGHT OUTER JOIN Customers,
   SalesOrders
    KEY LEFT OUTER JOIN SalesOrderItems
    KEY LEFT OUTER JOIN Products,
      SalesOrders
    KEY LEFT OUTER JOIN Employees AS e
    LEFT OUTER JOIN Employees AS m
        ON (e.ManagerID = m.EmployeeID)
WHERE Customers.State = 'CA'
GROUP BY Customers.GivenName, Products.Name, m.GivenName
ORDER BY SUM(SalesOrderItems.Quantity) DESC,
   Customers.GivenName;
```

GivenName	Name	SUM(SalesOrderItems.Quantity)	GivenName
Sheng	Baseball Cap	240	Moira
Laura	Tee Shirt	192	Moira
Moe	Tee Shirt	192	Moira
Leilani	Sweatshirt	132	Moira

GivenName	Name	SUM(SalesOrderItems.Quantity)	GivenName
...	...	...	...

Following is a diagram of the tables in this star join. The arrows indicate the directionality (left or right) of the outer joins. As you can see, the complete list of customers is maintained throughout all the joins.



The following standard ANSI/ISO syntax is equivalent to the star join in Example 2.

```
SELECT Customers.GivenName, Products.Name,
  SUM(SalesOrderItems.Quantity), m.GivenName
FROM SalesOrders LEFT OUTER JOIN SalesOrderItems
    ON SalesOrders.ID = SalesOrderItems.ID
  LEFT OUTER JOIN Products
    ON SalesOrderItems.ProductID = Products.ID
  LEFT OUTER JOIN Employees as e
    ON SalesOrders.SalesRepresentative = e.EmployeeID
  LEFT OUTER JOIN Employees as m
    ON e.ManagerID = m.EmployeeID
  RIGHT OUTER JOIN Customers
    ON SalesOrders.CustomerID = Customers.ID
WHERE Customers.State = 'CA'
GROUP BY Customers.GivenName, Products.Name, m.GivenName
ORDER BY SUM(SalesOrderItems.Quantity) DESC,
   Customers.GivenName;
```

Joins involving derived tables

Derived tables allow you to nest queries within a FROM clause. With derived tables, you can perform grouping of groups, or you can construct a join with a group, without having to create a separate view or table and join to it.

In the following example, the inner SELECT statement (enclosed in parentheses) creates a derived table, grouped by customer ID values. The outer SELECT statement assigns this table the correlation name sales_order_counts and joins it to the Customers table using a join condition.


```
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 result is a table of the names of those customers who have placed more than three orders, including the number of orders each has placed.

For an explanation of key joins of derived tables, see "Key joins of views and derived tables" on page 426.

For an explanation of natural joins of derived tables, see "Natural joins of views and derived tables" on page 416.

For an explanation of outer joins of derived tables, see "Outer joins of views and derived tables" on page 402.

Joins resulting from apply expressions

An apply expression is an easy way to specify joins where the right side is dependent upon the left. For example, use an apply expression to evaluate a procedure or derived table once for each row in a table expression. Apply expressions are placed in the FROM clause of a SELECT statement, and do not permit the use of an ON clause.

An APPLY combines rows from multiple sources, similar to a JOIN except that you cannot specify an ON condition for APPLY. The main difference between an APPLY and a JOIN is that the right side of an APPLY can change depending on the current row from the left side. For each row on the left side, the right side is recalculated and the resulting rows are joined with the row on the left. In the case where a row on the left side returns more than one row on the right, the left side is duplicated in the results as many times as there are rows returned from the right.

There are two types of APPLY you can specify: CROSS APPLY and OUTER APPLY. CROSS APPLY returns only rows on the left side that produce results on the right side. OUTER APPLY returns all rows that a CROSS APPLY returns, plus all rows on the left side for which the right side does not return rows (by supplying NULLs for the right side).

The syntax of an apply expression is as follows:

table-expression { **CROSS** | **OUTER** } **APPLY** *table-expression*

Example

The following example creates a procedure, EmployeesWithHighSalary, which takes as input a department ID, and returns the names of all employees in that department with salaries greater than $80,000.

```
CREATE PROCEDURE EmployeesWithHighSalary( IN dept INTEGER )
  RESULT ( Name LONG VARCHAR )
```

```
BEGIN
 SELECT E.GivenName || ' ' || E.Surname
     FROM Employees E
     WHERE E.DepartmentID = dept AND E.Salary > 80000;
END;
```

The following query uses OUTER APPLY to join the Departments table to the results of the EmployeesWithHighSalary procedure, and return the names of all employees with salary greater than $80,000 in each department. The query returns rows with NULL on the right side, indicating that there were no employees with salaries over $80,000 in the respective departments.

```
SELECT D.DepartmentName, HS.Name
    FROM Departments D
    OUTER APPLY EmployeesWithHighSalary( D.DepartmentID ) AS HS;
```

DepartmentName	Name
R & D	Kim Lull
R & D	David Scott
R & D	John Sheffield
Sales	Moira Kelly
Finance	Mary Anne Shea
Marketing	NULL
Shipping	NULL

The next query uses a CROSS APPLY to join the Departments table to the results of the EmployeesWithHighSalary procedure. Note that rows with NULL on the right side are not included.

```
SELECT D.DepartmentName, HS.Name
    FROM Departments D
    CROSS APPLY EmployeesWithHighSalary( D.DepartmentID ) AS HS;
```

DepartmentName	Name
R & D	Kim Lull
R & D	David Scott
R & D	John Sheffield
Sales	Moira Kelly
Finance	Mary Anne Shea

The next query returns the same results as the previous query, but uses a derived table as the right side of the CROSS APPLY.


```
SELECT D.DepartmentName, HS.Name
   FROM Departments D
    CROSS APPLY (
        SELECT E.GivenName || ' ' || E.Surname
            FROM Employees E
            WHERE E.DepartmentID = D.DepartmentID AND E.Salary > 80000
    ) HS( Name );
```

See also

- "FROM clause" [*SQL Anywhere Server - SQL Reference*]
- "Key joins" on page 417
- "Cross joins" on page 397
- "Inner and outer joins" on page 398

Natural joins

When you specify a natural join, SQL Anywhere generates a join condition based on columns with the same name. For this to work in a natural join of base tables, there must be at least one pair of columns with the same name, with one column from each table. If there is no common column name, an error is issued.

If table A and table B have one column name in common, and that column is called x, then

```
SELECT *
FROM A NATURAL JOIN B;
```

is equivalent to the following:

```
SELECT *
FROM A JOIN B
 ON A.x = B.x;
```

If table A and table B have two column names in common, and they are called a and b, then `A NATURAL JOIN B` is equivalent to the following:

```
A JOIN B
 ON A.a = B.a
 AND A.b = B.b;
```

Example 1

For example, you can join the Employees and Departments tables using a natural join because they have a column name in common, the DepartmentID column.

```
SELECT GivenName, Surname, DepartmentName
FROM Employees NATURAL JOIN Departments
ORDER BY DepartmentName, Surname, GivenName;
```

GivenName	Surname	DepartmentName
Janet	Bigelow	Finance

GivenName	Surname	DepartmentName
Kristen	Coe	Finance
James	Coleman	Finance
Jo Ann	Davidson	Finance
...	...	...

The following statement is equivalent. It explicitly specifies the join condition that was generated in the previous example.

```
SELECT GivenName, Surname, DepartmentName
FROM Employees JOIN Departments
  ON (Employees.DepartmentID = Departments.DepartmentID)
ORDER BY DepartmentName, Surname, GivenName;
```

Example 2

In Interactive SQL, execute the following query:

```
SELECT Surname, DepartmentName
FROM Employees NATURAL JOIN Departments;
```

Surname	DepartmentName
Whitney	R & D
Cobb	R & D
Breault	R & D
Shishov	R & D
Driscoll	R & D
...	...

SQL Anywhere looks at the two tables and determines that the only column name they have in common is DepartmentID. The following ON CLAUSE is internally generated and used to perform the join:

```
FROM Employees JOIN Departments
   ON Employees.DepartmentID = Departments.DepartmentID
```

NATURAL JOIN is just a shortcut for entering the ON clause; the two queries are identical.

Errors using NATURAL JOIN

The NATURAL JOIN operator can cause problems by equating columns you may not intend to be equated. For example, the following query generates unwanted results:


```
SELECT *
FROM SalesOrders NATURAL JOIN Customers;
```

The result of this query has no rows. SQL Anywhere internally generates the following ON clause:

```
FROM SalesOrders JOIN Customers
   ON SalesOrders.ID = Customers.ID
```

The ID column in the SalesOrders table is an ID number for the order. The ID column in the Customers table is an ID number for the customer. None of them match. Of course, even if a match were found, it would be a meaningless one.

Natural joins with an ON clause

When you specify a NATURAL JOIN and put a join condition in an ON clause, the result is the conjunction of the two join conditions.

For example, the following two queries are equivalent. In the first query, SQL Anywhere generates the join condition `Employees.DepartmentID = Departments.DepartmentID`. The query also contains an explicit join condition.

```
SELECT GivenName, Surname, DepartmentName
FROM Employees NATURAL JOIN Departments
  ON Employees.ManagerID = Departments.DepartmentHeadID;
```

The next query is equivalent. In it, the natural join condition that was generated in the previous query is specified in the ON clause.

```
SELECT GivenName, Surname, DepartmentName
FROM Employees JOIN Departments
  ON Employees.ManagerID = Departments.DepartmentHeadID
   AND Employees.DepartmentID = Departments.DepartmentID;
```

Natural joins of table expressions

When there is a multiple-table expression on at least one side of a natural join, SQL Anywhere generates a join condition by comparing the set of columns for each side of the join operator, and looking for columns that have the same name.

For example, in the statement

```
SELECT *
FROM (A JOIN B) NATURAL JOIN (C JOIN D);
```

there are two table expressions. The column names in the table expression `A JOIN B` are compared to the column names in the table expression `C JOIN D`, and a join condition is generated for each unambiguous pair of matching column names. An **unambiguous pair of matching columns** means that the column name occurs in both table expressions, but does not occur twice in the same table expression.

If there is a pair of ambiguous column names, an error is issued. However, a column name may occur twice in the same table expression, as long as it doesn't also match the name of a column in the other table expression.

Natural joins of lists

When a list of table expressions is on at least one side of a natural join, a separate join condition is generated for each table expression in the list.

Consider the following tables:

- table A consists of columns called a, b and c
- table B consists of columns called a and d
- table C consists of columns called d and c

In this case, the join `(A,B) NATURAL JOIN C` causes SQL Anywhere to generate two join conditions:

```
ON A.c = C.c
 AND B.d = C.d
```

If there is no common column name for A-C or B-C, an error is issued.

If table C consists of columns a, d, and c, then the join `(A,B) NATURAL JOIN C` is invalid. The reason is that column a appears in all three tables, and so the join is ambiguous.

Example

The following example answers the question: for each sale, provide information about what was sold and who sold it.

```
SELECT *
FROM ( Employees KEY JOIN SalesOrders )
  NATURAL JOIN ( SalesOrderItems KEY JOIN Products );
```

This is equivalent to

```
SELECT *
FROM ( Employees KEY JOIN SalesOrders )
  JOIN ( SalesOrderItems KEY JOIN Products )
    ON SalesOrders.ID = SalesOrderItems.ID;
```

Natural joins of views and derived tables

An extension to the ANSI/ISO SQL standard is that you can specify views or derived tables on either side of a natural join. In the following statement,

```
SELECT *
FROM View1 NATURAL JOIN View2;
```

the columns in View1 are compared to the columns in View2. If, for example, a column called EmployeeID is found to occur in both views, and there are no other columns that have identical names, then the generated join condition is `(View1.EmployeeID = View2.EmployeeID)`.

Example

The following example illustrates that a view used in a natural join can include expressions, and not just columns, and they are treated the same way in the natural join. First, create the view V with a column called x, as follows:

```
CREATE VIEW V(x) AS
SELECT R.y + 1
FROM R;
```

Next, create a natural join of the view to a derived table. The derived table has a correlation name T with a column called x.

```
SELECT *
FROM V NATURAL JOIN (SELECT P.y FROM P) as T(x);
```

This join is equivalent to the following:

```
SELECT *
FROM V JOIN (SELECT P.y FROM P) as T(x) ON (V.x = T.x);
```

Key joins

Many common joins are between two tables related by a foreign key. The most common join restricts foreign key values to be equal to primary key values. The KEY JOIN operator joins two tables based on a foreign key relationship. In other words, SQL Anywhere generates an ON clause that equates the primary key column from one table with the foreign key column of the other. To use a key join, there must be a foreign key relationship between the tables, or an error is issued.

A key join can be considered a shortcut for the ON clause; the two queries are identical. However, you can also use the ON clause with a KEY JOIN. Key join is the default when you specify JOIN but do not specify CROSS, NATURAL, KEY, or use an ON clause. If you look at the diagram of the SQL Anywhere sample database, lines between tables represent foreign keys. You can use the KEY JOIN operator anywhere two tables are joined by a line in the diagram. For more information about the SQL Anywhere sample database, see "Tutorial: Using the sample database" [*SQL Anywhere Server - Database Administration*].

When key join is the default

Key join is the default in SQL Anywhere when all the following apply:

- the keyword JOIN is used.
- the keywords CROSS, NATURAL or KEY are not specified.
- there is no ON clause.

Example

For example, the following query joins the tables Products and SalesOrderItems based on the foreign key relationship in the database.

```
SELECT *
FROM Products KEY JOIN SalesOrderItems;
```

The next query is equivalent. It leaves out the word KEY, but by default a JOIN without an ON clause is a KEY JOIN.

```
SELECT *
FROM Products JOIN SalesOrderItems;
```

The next query is also equivalent because the join condition specified in the ON clause is the same as the join condition that SQL Anywhere generates for these tables based on their foreign key relationship in the SQL Anywhere sample database.

```
SELECT *
FROM Products JOIN SalesOrderItems
ON SalesOrderItems.ProductID = Products.ID;
```

Key joins with an ON clause

When you specify a KEY JOIN and put a join condition in an ON clause, the result is the conjunction of the two join conditions. For example,

```
SELECT *
FROM A KEY JOIN B
ON A.x = B.y;
```

If the join condition generated by the key join of A and B is `A.w = B.z`, then this query is equivalent to

```
SELECT *
FROM A JOIN B
ON A.x = B.y AND A.w = B.z;
```

Key joins when there are multiple foreign key relationships

When SQL Anywhere attempts to generate a join condition based on a foreign key relationship, it sometimes finds more than one relationship. In these cases, SQL Anywhere determines which foreign key relationship to use by matching the role name of the foreign key to the correlation name of the primary key table that the foreign key references.

The following sections describe how SQL Anywhere generates join conditions for key joins. This information is summarized in “Rules describing the operation of key joins” on page 428.

Correlation name and role name

A **correlation name** is the name of a table or view that is used in the FROM clause of the query—either its original name, or an alias that is defined in the FROM clause.

A **role name** is the name of the foreign key. It must be unique for a given foreign (child) table.

If you do not specify a role name for a foreign key, the name is assigned as follows:

- If there is no foreign key with the same name as the primary table name, the primary table name is assigned as the role name.
- If the primary table name is already being used by another foreign key, the role name is the primary table name concatenated with a zero-padded three-digit number unique to the foreign table.

If you don't know the role name of a foreign key, you can find it in Sybase Central by expanding the database container in the left pane. Select the table in left pane, and then click the **Constraints** tab in the right pane. A list of foreign keys for that table appears in the right pane.

See "Sample database schema" on page 387 for a diagram that includes the role names of all foreign keys in the SQL Anywhere sample database.

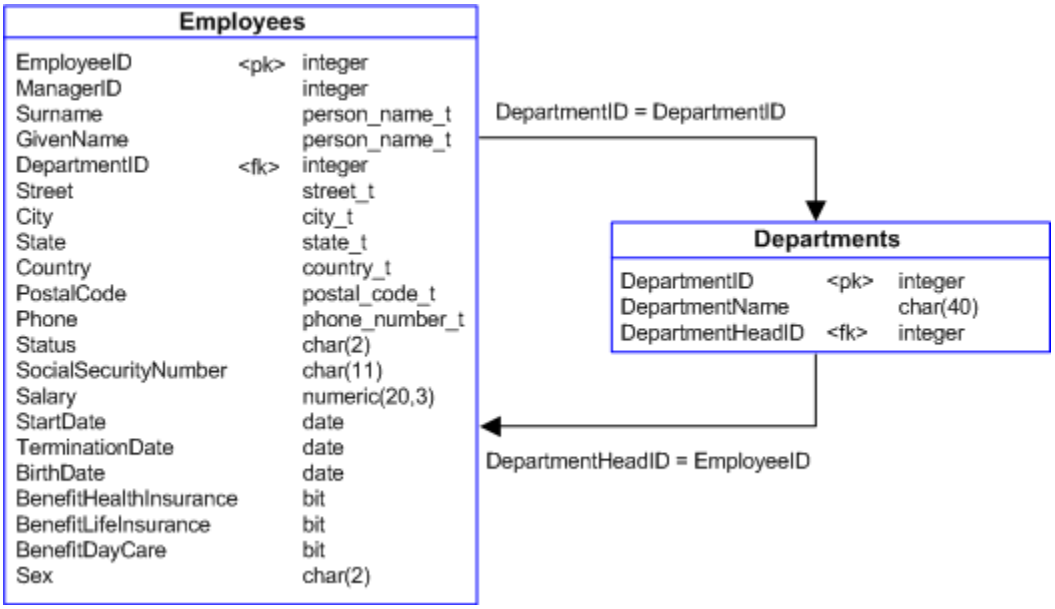
Generating join conditions

SQL Anywhere looks for a foreign key that has the same role name as the correlation name of the primary key table:

- If there is exactly one foreign key with the same name as a table in the join, SQL Anywhere uses it to generate the join condition.
- If there is more than one foreign key with the same name as a table, the join is ambiguous and an error is issued.
- If there is no foreign key with the same name as the table, SQL Anywhere looks for any foreign key relationship, even if the names don't match. If there is more than one foreign key relationship, the join is ambiguous and an error is issued.

Example 1

In the SQL Anywhere sample database, two foreign key relationships are defined between the tables Employees and Departments: the foreign key FK_DepartmentID_DepartmentID in the Employees table references the Departments table; and the foreign key FK_DepartmentHeadID_EmployeeID in the Departments table references the Employees table.



The following query is ambiguous because there are two foreign key relationships and neither has the same role name as the primary key table name. Therefore, attempting this query results in the syntax error `SQLE_AMBIGUOUS_JOIN (-147)`.

```
SELECT Employees.Surname, Departments.DepartmentName
FROM Employees KEY JOIN Departments;
```

Example 2

This query modifies the query in Example 1 by specifying the correlation name FK_DepartmentID_DepartmentID for the Departments table. Now, the foreign key FK_DepartmentID_DepartmentID has the same name as the table it references, and so it is used to define the join condition. The result includes all the employee last names and the departments where they work.

```
SELECT Employees.Surname,
    FK_DepartmentID_DepartmentID.DepartmentName
FROM Employees KEY JOIN Departments
    AS FK_DepartmentID_DepartmentID;
```

The following query is equivalent. It is not necessary to create an alias for the Departments table in this example. The same join condition that was generated above is specified in the ON clause in this query:

```
SELECT Employees.Surname, Departments.DepartmentName
FROM Employees JOIN Departments
   ON Departments.DepartmentID = Employees.DepartmentID;
```

Example 3

If the intent was to list all the employees that are the head of a department, then the foreign key FK_DepartmentHeadID_EmployeeID should be used and Example 1 should be rewritten as follows. This query imposes the use of the foreign key FK_DepartmentHeadID_EmployeeID by specifying the correlation name FK_DepartmentHeadID_EmployeeID for the primary key table Employees.


```
SELECT FK_DepartmentHeadID_EmployeeID.Surname, Departments.DepartmentName
FROM Employees AS FK_DepartmentHeadID_EmployeeID
    KEY JOIN Departments;
```

The following query is equivalent. The join condition that was generated above is specified in the ON clause in this query:

```
SELECT Employees.Surname, Departments.DepartmentName
FROM Employees JOIN Departments
   ON Departments.DepartmentHeadID = Employees.EmployeeID;
```

Example 4

A correlation name is not needed if the foreign key role name is identical to the primary key table name. For example, you can define the foreign key Departments for the Employees table:

```
ALTER TABLE Employees
   ADD FOREIGN KEY Departments (DepartmentID)
   REFERENCES Departments (DepartmentID);
```

Now, this foreign key relationship is the default join condition when a KEY JOIN is specified between the two tables. If the foreign key Departments is defined, then the following query is equivalent to Example 3.

```
SELECT Employees.Surname, Departments.DepartmentName
FROM Employees KEY JOIN Departments;
```

Note

If you try this example in Interactive SQL, you should reverse the change to the SQL Anywhere sample database with the following statement:

```
ALTER TABLE Employees DROP FOREIGN KEY Departments;
```

Key joins of table expressions

SQL Anywhere generates join conditions for the key join of table expressions by examining the foreign key relationship of each pair of tables in the statement.

The following example joins four pairs of tables.

```
SELECT *
FROM (A NATURAL JOIN B) KEY JOIN (C NATURAL JOIN D);
```

The table-pairs are A-C, A-D, B-C and B-D. SQL Anywhere considers the relationship within each pair and then creates a generated join condition for the table expression as a whole. How SQL Anywhere does this depends on whether the table expressions use commas or not. Therefore, the generated join conditions in the following two examples are different. `A JOIN B` is a table expression that does not contain commas, and `(A,B)` is a table expression list.

```
SELECT *
FROM (A JOIN B) KEY JOIN C;
```

is semantically different from

```
SELECT *
FROM (A,B) KEY JOIN C;
```

The two types of join behavior are explained in the following sections:

- "Key joins of table expressions that do not contain commas" on page 422
- "Key joins of table expression lists" on page 423

Key joins of table expressions that do not contain commas

When both of the two table expressions being joined do not contain commas, SQL Anywhere examines the foreign key relationships in the pairs of tables in the statement, and generates a single join condition.

For example, the following join has two table-pairs, A-C and B-C.

```
(A NATURAL JOIN B) KEY JOIN C
```

SQL Anywhere generates a single join condition for joining C with `(A NATURAL JOIN B)` by looking at the foreign key relationships within the table-pairs A-C and B-C. It generates one join condition for the two pairs according to the rules for determining key joins when there are multiple foreign key relationships:

- First, it looks at both A-C and B-C for a single foreign key that has the same role name as the correlation name of one of the primary key tables it references. If there is exactly one foreign key meeting this criterion, it uses it. If there is more than one foreign key with the same role name as the correlation name of a table, the join is considered to be ambiguous and an error is issued.
- If there is no foreign key with the same name as the correlation name of a table, SQL Anywhere looks for any foreign key relationship between the tables. If there is one, it uses it. If there is more than one, the join is considered to be ambiguous and an error is issued.
- If there is no foreign key relationship, an error is issued.

For more information, see "Key joins when there are multiple foreign key relationships" on page 418.

Example

The following query finds all the employees who are sales representatives, and their departments.

```
SELECT Employees.Surname,
        FK_DepartmentID_DepartmentID.DepartmentName
FROM ( Employees KEY JOIN Departments
        AS FK_DepartmentID_DepartmentID )
    KEY JOIN SalesOrders;
```

You can interpret this query as follows.

- SQL Anywhere considers the table expression `( Employees KEY JOIN Departments as FK_DepartmentID_DepartmentID )` and generates the join condition `Employees.DepartmentID = FK_DepartmentID_DepartmentID.DepartmentID` based on the foreign key FK_DepartmentID_DepartmentID.

- SQL Anywhere then considers the table-pairs Employees/SalesOrders and Departments/SalesOrders. Note that only one foreign key can exist between the tables SalesOrders and Employees and between SalesOrders and Departments, or the join is ambiguous. As it happens, there is exactly one foreign key relationship between the tables SalesOrders and Employees (FK_SalesRepresentative_EmployeeID), and no foreign key relationship between SalesOrders and Departments. So, the generated join condition is `SalesOrders.EmployeeID = Employees.SalesRepresentative`.

The following query is therefore equivalent to the previous query:

```
SELECT Employees.Surname, Departments.DepartmentName
FROM ( Employees JOIN Departments
    ON ( Employees.DepartmentID = Departments.DepartmentID ) )
JOIN SalesOrders
    ON ( Employees.EmployeeID = SalesOrders.SalesRepresentative );
```

Key joins of table expression lists

To generate a join condition for the key join of two table expression lists, SQL Anywhere examines the pairs of tables in the statement, and generates a join condition for each pair. The final join condition is the conjunction of the join conditions for each pair. There must be a foreign key relationship between each pair.

The following example joins two table-pairs, A-C and B-C.

```
SELECT *
FROM ( A,B ) KEY JOIN C;
```

SQL Anywhere generates a join condition for joining C with `(A,B)` by generating a join condition for each of the two pairs A-C and B-C. It does so according to the rules for key joins when there are multiple foreign key relationships:

- For each pair, SQL Anywhere looks for a foreign key that has the same role name as the correlation name of the primary key table. If there is exactly one foreign key meeting this criterion, it uses it. If there is more than one, the join is considered to be ambiguous and an error is issued.
- For each pair, if there is no foreign key with the same name as the correlation name of the table, SQL Anywhere looks for any foreign key relationship between the tables. If there is one, it uses it. If there is more than one, the join is considered to be ambiguous and an error is issued.
- For each pair, if there is no foreign key relationship, an error is issued.
- If SQL Anywhere is able to determine exactly one join condition for each pair, it combines the join conditions using AND.

See also "Key joins when there are multiple foreign key relationships" on page 418.

Example

The following query returns the names of all salespeople who have sold at least one order to a specific region.

```
SELECT DISTINCT Employees.Surname,
        FK_DepartmentID_DepartmentID.DepartmentName,
        SalesOrders.Region
```

```
FROM ( SalesOrders, Departments
        AS FK_DepartmentID_DepartmentID )
    KEY JOIN Employees;
```

Surname	DepartmentName	Region
Chin	Sales	Eastern
Chin	Sales	Western
Chin	Sales	Central
...	...	...

This query deals with two pairs of tables: SalesOrders and Employees; and Departments AS FK_DepartmentID_DepartmentID and Employees.

For the pair SalesOrders and Employees, there is no foreign key with the same role name as one of the tables. However, there is a foreign key (FK_SalesRepresentative_EmployeeID) relating the two tables. It is the only foreign key relating the two tables, and so it is used, resulting in the generated join condition `( Employees.EmployeeID = SalesOrders.SalesRepresentative )`.

For the pair Departments AS FK_DepartmentID_DepartmentID and Employees, there is one foreign key that has the same role name as the primary key table. It is FK_DepartmentID_DepartmentID, and it matches the correlation name given to the Departments table in the query. There are no other foreign keys with the same name as the correlation name of the primary key table, so FK_DepartmentID_DepartmentID is used to form the join condition for the table-pair. The join condition that is generated is `(Employees.DepartmentID = FK_DepartmentID_DepartmentID.DepartmentID)`. Note that there is another foreign key relating the two tables, but as it has a different name from either of the tables, it is not a factor.

The final join condition adds together the join condition generated for each table-pair. Therefore, the following query is equivalent:

```
SELECT DISTINCT Employees.Surname,
   Departments.DepartmentName,
   SalesOrders.Region
FROM ( SalesOrders, Departments )
    JOIN Employees
    ON Employees.EmployeeID = SalesOrders.SalesRepresentative
    AND Employees.DepartmentID = Departments.DepartmentID;
```

Key joins of lists and table expressions that do not contain commas

When table expression lists are joined via key join with table expressions that do not contain commas, SQL Anywhere generates a join condition for each table in the table expression list.

For example, the following statement is the key join of a table expression list with a table expression that does not contain commas. This example generates a join condition for table A with table expression `C NATURAL JOIN D`, and for table B with table expression `C NATURAL JOIN D`.


```
SELECT *
FROM (A,B) KEY JOIN (C NATURAL JOIN D);
```

`(A,B)` is a list of table expressions and `C NATURAL JOIN D` is a table expression. SQL Anywhere must therefore generate two join conditions: it generates one join condition for the pairs A-C and A-D, and a second join condition for the pairs B-C and B-D. It does so according to the rules for key joins when there are multiple foreign key relationships:

- For each set of table-pairs, SQL Anywhere looks for a foreign key that has the same role name as the correlation name of one of the primary key tables. If there is exactly one foreign key meeting this criterion, it uses it. If there is more than one, the join is ambiguous and an error is issued.
- For each set of table-pairs, if there is no foreign key with the same name as the correlation name of a table, SQL Anywhere looks for any foreign key relationship between the tables. If there is exactly one relationship, it uses it. If there is more than one, the join is ambiguous and an error is issued.
- For each set of pairs, if there is no foreign key relationship, an error is issued.
- If SQL Anywhere is able to determine exactly one join condition for each set of pairs, it combines the join conditions with the keyword AND.

Example 1

Consider the following join of five tables:

```
((A,B) JOIN (C NATURAL JOIN D) ON A.x = D.y) KEY JOIN E
```

In this case, SQL Anywhere generates a join condition for the key join to E by generating a condition either between `(A,B)` and E or between `C NATURAL JOIN D` and E. This is as described in "Key joins of table expressions that do not contain commas" on page 422.

If SQL Anywhere generates a join condition between `(A,B)` and E, it needs to create two join conditions, one for A-E and one for B-E. It must find a valid foreign key relationship within each table-pair. This is as described in "Key joins of table expression lists" on page 423.

If SQL Anywhere creates a join condition between `C NATURAL JOIN D` and E, it creates only one join condition, and so must find only one foreign key relationship in the pairs C-E and D-E. This is as described in "Key joins of table expressions that do not contain commas" on page 422.

Example 2

The following is an example of a key join of a table expression and a list of table expressions. The example provides the name and department of employees who are sales representatives and also managers.

```
SELECT DISTINCT Employees.Surname,
        FK_DepartmentID_DepartmentID.DepartmentName
FROM ( SalesOrders, Departments
        AS FK_DepartmentID_DepartmentID )
    KEY JOIN ( Employees JOIN Departments AS d
        ON Employees.EmployeeID = d.DepartmentHeadID );
```

SQL Anywhere generates two join conditions:

- There is exactly one foreign key relationship between the table-pairs SalesOrders/Employees and SalesOrders/d: `SalesOrders.SalesRepresentative = Employees.EmployeeID`.
- There is exactly one foreign key relationship between the table-pairs FK_DepartmentID_DepartmentID/Employees and FK_DepartmentID_DepartmentID/d: `FK_DepartmentID_DepartmentID.DepartmentID = Employees.DepartmentID`.

This example is equivalent to the following. In the following version, it is not necessary to create the correlation name `Departments AS FK_DepartmentID_DepartmentID`, because that was only needed to clarify which of two foreign keys should be used to join Employees and Departments.

```
SELECT DISTINCT Employees.Surname,
   Departments.DepartmentName
FROM ( SalesOrders, Departments )
    JOIN ( Employees JOIN Departments AS d
       ON Employees.EmployeeID = d.DepartmentHeadID )
    ON SalesOrders.SalesRepresentative = Employees.EmployeeID
       AND Departments.DepartmentID = Employees.DepartmentID;
```

Key joins of views and derived tables

When you include a view or derived table in a key join, SQL Anywhere follows the same basic procedure as with tables, but with these differences:

- For each key join, SQL Anywhere considers the pairs of tables in the FROM clause of the query and the view, and generates one join condition for the set of all pairs, regardless of whether the FROM clause in the view contains commas or join keywords.
- SQL Anywhere joins the tables based on the foreign key that has the same role name as the correlation name of the view or derived table.
- When you include a view or derived table in a key join, the view or derived table definition cannot contain UNION, INTERSECT, EXCEPT, ORDER BY, DISTINCT, GROUP BY, aggregate functions, window functions, TOP, FIRST, START AT, or FOR XML. If it contains any of these items, an error is returned. In addition, the derived table cannot be defined as a recursive table expression.

  A derived table works identically to a view. The only difference is that instead of referencing a predefined view, the definition for the table is included in the statement.

  For information about recursive table expressions, see "Recursive common table expressions" on page 435, and "RecursiveTable algorithm (RT)" on page 585.

Example 1

For example, in the following statement, View1 is a view.

```
SELECT *
FROM View1 KEY JOIN B;
```

The definition of View1 can be any of the following and result in the same join condition to B. (The result set will differ, but the join conditions will be identical.)


```
SELECT *
FROM C CROSS JOIN D;
```

or

```
SELECT *
FROM C,D;
```

or

```
SELECT *
FROM C JOIN D ON (C.x = D.y);
```

In each case, to generate a join condition for the key join of View1 and B, SQL Anywhere considers the table-pairs C-B and D-B, and generates a single join condition. It generates the join condition based on the rules for multiple foreign key relationships described in "Key joins of table expressions" on page 421, except that it looks for a foreign key with the same name as the correlation name of the view (rather than a table referenced in the view).

Using any of the view definitions above, you can interpret the processing of `View1 KEY JOIN B` as follows:

SQL Anywhere generates a single join condition by considering the table-pairs C-B and D-B. It generates the join condition according to the rules for determining key joins when there are multiple foreign key relationships:

- First, it looks at both C-B and D-B for a single foreign key that has the same role name as the correlation name of the view. If there is exactly one foreign key meeting this criterion, it uses it. If there is more than one foreign key with the same role name as the correlation name of the view, the join is considered to be ambiguous and an error is issued.
- If there is no foreign key with the same name as the correlation name of the view, SQL Anywhere looks for any foreign key relationship between the tables. If there is one, it uses it. If there is more than one, the join is considered to be ambiguous and an error is issued.
- If there is no foreign key relationship, an error is issued.

Assume this generated join condition is `B.y = D.z`. You can now expand the original join. For example, the following two statements are equivalent:

```
SELECT *
FROM View1 KEY JOIN B;

SELECT *
FROM View1 JOIN B ON B.y = View1.z;
```

See "Key joins when there are multiple foreign key relationships" on page 418.

Example 2

The following view contains all the employee information about the manager of each department.

```
CREATE VIEW V AS
SELECT Departments.DepartmentName, Employees.*
```

```
FROM Employees JOIN Departments
  ON Employees.EmployeeID = Departments.DepartmentHeadID;
```

The following query joins the view to a table expression.

```
SELECT *
FROM V KEY JOIN ( SalesOrders,
   Departments FK_DepartmentID_DepartmentID );
```

The following query is equivalent to the previous query:

```
SELECT *
FROM V JOIN ( SalesOrders,
   Departments FK_DepartmentID_DepartmentID )
ON ( V.EmployeeID = SalesOrders.SalesRepresentative
AND V.DepartmentID =
    FK_DepartmentID_DepartmentID.DepartmentID );
```

Rules describing the operation of key joins

The following rules summarize the information provided above.

Rule 1: Key join of two tables

This rule applies to `A KEY JOIN B`, where A and B are base or temporary tables.

1. Find all foreign keys from A referencing B.

   If there exists a foreign key whose role name is the correlation name of table B, then mark it as a preferred foreign key.

2. Find all foreign keys from B referencing A.

   If there exists a foreign key whose role name is the correlation name of table A, then mark it as a preferred foreign key.

3. If there is more than one preferred key, the join is ambiguous. The syntax error `SQLE_AMBIGUOUS_JOIN (-147)` is issued.

4. If there is a single preferred key, then this foreign key is chosen to define the generated join condition for this KEY JOIN expression.

5. If there is no preferred key, then other foreign keys between A and B are used:

   - If there is more than one foreign key between A and B, then the join is ambiguous. The syntax error `SQLE_AMBIGUOUS_JOIN (-147)` is issued.

   - If there is a single foreign key, then this foreign key is chosen to define the generated join condition for this KEY JOIN expression.

   - If there is no foreign key, then the join is invalid and an error is generated.

Rule 2: Key join of table expressions that do not contain commas

This rule applies to `A KEY JOIN B`, where A and B are table expressions that do not contain commas.

1. For each pair of tables; one from expression A and one from expression B, list all foreign keys, and mark all preferred foreign keys between the tables. The rule for determining a preferred foreign key is given in Rule 1, above.
2. If there is more than one preferred key, then the join is ambiguous. The syntax error `SQLE_AMBIGUOUS_JOIN (-147)` is issued.
3. If there is a single preferred key, then this foreign key is chosen to define the generated join condition for this KEY JOIN expression.
4. If there is no preferred key, then other foreign keys between pairs of tables are used:
   - If there is more than one foreign key, then the join is ambiguous. The syntax error `SQLE_AMBIGUOUS_JOIN (-147)` is issued.
   - If there is a single foreign key, then this foreign key is chosen to define the generated join condition for this KEY JOIN expression.
   - If there is no foreign key, then the join is invalid and an error is generated.

Rule 3: Key join of table expression lists

This rule applies to `(A1, A2, ...) KEY JOIN ( B1, B2, ...)` where A1, B1, and so on are table expressions that do not contain commas.

1. For each pair of table expressions Ai and Bj, find a unique generated join condition for the table expression `(Ai KEY JOIN Bj)` by applying Rule 1 or 2. If any KEY JOIN for a pair of table expressions is ambiguous by Rule 1 or 2, a syntax error is generated.
2. The generated join condition for this KEY JOIN expression is the conjunction of the join conditions found in step 1.

Rule 4: Key join of lists and table expressions that do not contain commas

This rule applies to `(A1, A2, ...) KEY JOIN ( B1, B2, ...)` where A1, B1, and so on are table expressions that may contain commas.

1. For each pair of table expressions Ai and Bj, find a unique generated join condition for the table expression `(Ai KEY JOIN Bj)` by applying Rule 1, 2, or 3. If any KEY JOIN for a pair of table expressions is ambiguous by Rule 1, 2, or 3, then a syntax error is generated.
2. The generated join condition for this KEY JOIN expression is the conjunction of the join conditions found in step 1.

Common table expressions

The WITH prefix to the SELECT statement affords you the opportunity to define common table expressions. Common table expressions are temporary views that are known only within the scope of a single SELECT statement. They permit you to write queries more easily, and to write queries that could not otherwise be expressed.

Common table expressions are useful or may be necessary if a query involves multiple aggregate functions or defines a view within a stored procedure that references program variables. Common table expressions also provide a convenient means to temporarily store sets of values.

Using common table expressions

Common table expressions are defined using the WITH clause, which precedes the SELECT keyword in a SELECT statement. The content of the clause defines one or more temporary views that may then be referenced elsewhere in the statement. The syntax of this clause mimics that of the CREATE VIEW statement. Using common table expressions, you can express the previous query as follows.

```
WITH CountEmployees( DepartmentID, n ) AS
  ( SELECT DepartmentID, COUNT( * ) AS n
    FROM Employees GROUP BY DepartmentID )
SELECT DepartmentID, n
FROM CountEmployees
WHERE n = ( SELECT MAX( n )
            FROM CountEmployees );
```

Changing the query to search for the department with the fewest employees demonstrates that such queries may return multiple rows.

```
WITH CountEmployees( DepartmentID, n ) AS
  ( SELECT DepartmentID, COUNT( * ) AS n
    FROM Employees GROUP BY DepartmentID )
SELECT DepartmentID, n
FROM CountEmployees
WHERE n = ( SELECT MIN( n )
            FROM CountEmployees );
```

In the SQL Anywhere sample database, two departments share the minimum number of employees, which is 9.

See also

- “Specifying multiple correlation names” on page 430
- “Using multiple table expressions” on page 431
- “Where common table expressions are permitted” on page 431

Specifying multiple correlation names

Just as when using tables, you can give different correlation names to multiple instances of a common table expression. Doing so permits you to join a common table expression to itself. For example, the query below produces pairs of departments that have the same number of employees, although there are only two departments with the same number of employees in the SQL Anywhere sample database.


```
WITH CountEmployees( DepartmentID, n ) AS
    ( SELECT DepartmentID, COUNT( * ) AS n
      FROM Employees GROUP BY DepartmentID )
SELECT a.DepartmentID, a.n, b.DepartmentID, b.n
FROM CountEmployees AS a JOIN CountEmployees AS b
ON a.n = b.n AND a.DepartmentID < b.DepartmentID;
```

See also

- "Using common table expressions" on page 430
- "Using multiple table expressions" on page 431
- "Where common table expressions are permitted" on page 431

Using multiple table expressions

A single WITH clause may define more than one common table expression. These definitions must be separated by commas. The following example lists the department that has the smallest payroll and the department that has the largest number of employees.

```
WITH
  CountEmployees( DepartmentID, n ) AS
    ( SELECT DepartmentID, COUNT( * ) AS n
      FROM Employees GROUP BY DepartmentID ),
  DepartmentPayroll( DepartmentID, amount ) AS
     ( SELECT DepartmentID, SUM( Salary ) AS amount
       FROM Employees GROUP BY DepartmentID )
SELECT count.DepartmentID, count.n, pay.amount
FROM CountEmployees AS count JOIN DepartmentPayroll AS pay
ON count.DepartmentID = pay.DepartmentID
WHERE count.n = ( SELECT MAX( n ) FROM CountEmployees )
   OR pay.amount = ( SELECT MIN( amount ) FROM DepartmentPayroll );
```

See also

- "Using common table expressions" on page 430
- "Specifying multiple correlation names" on page 430
- "Where common table expressions are permitted" on page 431

Where common table expressions are permitted

Common table expression definitions are permitted in only three places, although they may be referenced throughout the body of the query or in any subqueries.

- **Top-level SELECT statement** Common table expressions are permitted within top-level SELECT statements, but not within subqueries.

  ```
  WITH DepartmentPayroll( DepartmentID, amount ) AS
    ( SELECT DepartmentID, SUM( Salary ) AS amount
      FROM Employees GROUP BY DepartmentID )
  SELECT DepartmentID, amount
  FROM DepartmentPayroll
  WHERE amount = ( SELECT MAX( amount )
                   FROM DepartmentPayroll );
  ```

- **The top-level SELECT statement in a view definition** Common table expressions are permitted within the top-level SELECT statement that defines a view, but not within subqueries within the definition.

```
CREATE VIEW LargestDept ( DepartmentID, Size, pay ) AS
   WITH
     CountEmployees( DepartmentID, n ) AS
       ( SELECT DepartmentID, COUNT( * ) AS n
         FROM Employees GROUP BY DepartmentID ),
     DepartmentPayroll( DepartmentID, amount ) AS
       ( SELECT DepartmentID, SUM( Salary ) AS amount
         FROM Employees GROUP BY DepartmentID )
  SELECT count.DepartmentID, count.n, pay.amount
  FROM CountEmployees count JOIN DepartmentPayroll pay
  ON count.DepartmentID = pay.DepartmentID
  WHERE count.n = ( SELECT MAX( n ) FROM CountEmployees )
     OR pay.amount = ( SELECT MAX( amount ) FROM DepartmentPayroll );
```

- **A top-level SELECT statement in an INSERT statement** Common table expressions are permitted within a top-level SELECT statement in an INSERT statement, but not within subqueries within the INSERT statement.

```
CREATE TABLE LargestPayrolls ( DepartmentID INTEGER, Payroll NUMERIC,
CurrentDate DATE );
INSERT INTO LargestPayrolls( DepartmentID, Payroll, CurrentDate )
  WITH DepartmentPayroll( DepartmentID, amount ) AS
    ( SELECT DepartmentID, SUM( Salary ) AS amount
      FROM Employees
      GROUP BY DepartmentID )
  SELECT DepartmentID, amount, CURRENT TIMESTAMP
  FROM DepartmentPayroll
  WHERE amount = ( SELECT MAX( amount )
                   FROM DepartmentPayroll );
```

See also

- “Using common table expressions” on page 430
- “Specifying multiple correlation names” on page 430
- “Using multiple table expressions” on page 431

Typical applications of common table expressions

In general, common table expressions are useful whenever a table expression must appear multiple times within a single query. The following typical situations are suited to common table expressions.

- Queries that involve multiple aggregate functions.
- Views within a procedure that must contain a reference to a program variable.
- Queries that use temporary views to store a set of values.

This list is not exhaustive; you may encounter many other situations in which common table expressions are useful.

Multiple aggregate functions

Common table expressions are useful whenever multiple levels of aggregation must appear within a single query. This is the case in the example used in the previous section. The task was to retrieve the department ID of the department that has the most employees. To do so, the count aggregate function is used to calculate the number of employees in each department and the MAX function is used to select the largest department.

A similar situation arises when writing a query to determine which department has the largest payroll. The SUM aggregate function is used to calculate each department's payroll and the MAX function to determine which is largest. The presence of both functions in the query is a clue that a common table expression may be helpful.

```
WITH DepartmentPayroll( DepartmentID, amount ) AS
    ( SELECT DepartmentID, SUM( Salary ) AS amount
      FROM Employees GROUP BY DepartmentID )
SELECT DepartmentID, amount
FROM DepartmentPayroll
WHERE amount = ( SELECT MAX( amount )
                 FROM DepartmentPayroll )
```

For more information about aggregate functions, see “Window aggregate functions” on page 462.

Views that reference program variables

Sometimes, it can be convenient to create a view that contains a reference to a program variable. For example, you may define a variable within a procedure that identifies a particular customer. You want to query the customer's purchase history, and as you will be accessing similar information multiple times or perhaps using multiple aggregate functions, you want to create a view that contains information about that specific customer.

You cannot create a view that references a program variable because there is no way to limit the scope of a view to that of your procedure. Once created, a view can be used in other contexts. You can, however, use a common table expressions within the queries in your procedure. As the scope of a common table expression is limited to the statement, the variable reference creates no ambiguity and is permitted.

The following statement selects the gross sales of the various sales representatives in the SQL Anywhere sample database.

```
SELECT GivenName || ' ' || Surname AS sales_rep_name,
       SalesRepresentative AS sales_rep_id,
       SUM( p.UnitPrice * i.Quantity ) AS total_sales
FROM Employees LEFT OUTER JOIN SalesOrders AS o
               INNER JOIN SalesOrderItems AS I
               INNER JOIN Products AS p
WHERE OrderDate BETWEEN '2000-01-01' AND '2001-12-31'
GROUP BY SalesRepresentative, GivenName, Surname;
```

The above query is the basis of the common table expression that appears in the following procedure. The ID number of the sales representative and the year in question are incoming parameters. As the following procedure demonstrates, the procedure parameters and any declared local variables can be referenced within the WITH clause.

```
CREATE PROCEDURE sales_rep_total (
  IN rep  INTEGER,
  IN yyyy INTEGER )
BEGIN
  DECLARE StartDate DATE;
  DECLARE EndDate   DATE;
  SET StartDate = YMD( yyyy,  1,  1 );
  SET EndDate = YMD( yyyy, 12, 31 );
  WITH total_sales_by_rep ( sales_rep_name,
                            sales_rep_id,
                            month,
                            order_year,
                            total_sales ) AS
  ( SELECT GivenName || ' ' || Surname AS sales_rep_name,
           SalesRepresentative AS sales_rep_id,
           month( OrderDate),
           year( OrderDate ),
           SUM( p.UnitPrice * i.Quantity ) AS total_sales
    FROM Employees LEFT OUTER JOIN SalesOrders o
                        INNER JOIN SalesOrderItems I
                        INNER JOIN Products p
    WHERE OrderDate BETWEEN StartDate AND EndDate
          AND SalesRepresentative = rep
    GROUP BY year( OrderDate ), month( OrderDate ),
             GivenName, Surname, SalesRepresentative )
  SELECT sales_rep_name,
         monthname( YMD(yyyy, month, 1) ) AS month_name,
         order_year,
         total_sales
  FROM total_sales_by_rep
  WHERE total_sales =
    ( SELECT MAX( total_sales) FROM total_sales_by_rep )
  ORDER BY order_year ASC, month ASC;
END;
```

The following statement calls the previous procedure.

```
CALL sales_rep_total( 129, 2000 );
```

Views that store values

It can be useful to store a particular set of values within a SELECT statement or within a procedure. For example, suppose a company prefers to analyze the results of its sales staff by thirds of a year, instead of by quarter. Since there is no built-in date part for thirds, as there is for quarters, it is necessary to store the dates within the procedure.

```
WITH thirds ( q_name, q_start, q_end ) AS
( SELECT 'T1', '2000-01-01', '2000-04-30' UNION
  SELECT 'T2', '2000-05-01', '2000-08-31' UNION
  SELECT 'T3', '2000-09-01', '2000-12-31' )
SELECT q_name,
       SalesRepresentative,
       count(*) AS num_orders,
       SUM( p.UnitPrice * i.Quantity ) AS total_sales
FROM thirds LEFT OUTER JOIN SalesOrders AS o
    ON OrderDate BETWEEN q_start and q_end
                  KEY JOIN SalesOrderItems AS I
                  KEY JOIN Products AS p
 GROUP BY q_name, SalesRepresentative
 ORDER BY q_name, SalesRepresentative;
```


This method should be used with care, as the values may need periodic maintenance. For example, the above statement must be modified if it is to be used for any other year.

You can also apply this method within procedures. The following example declares a procedure that takes the year in question as an argument.

```
CREATE PROCEDURE sales_by_third ( IN y INTEGER )
BEGIN
  WITH thirds ( q_name, q_start, q_end ) AS
  ( SELECT 'T1', YMD( y, 01, 01), YMD( y, 04, 30 ) UNION
    SELECT 'T2', YMD( y, 05, 01), YMD( y, 08, 31 ) UNION
    SELECT 'T3', YMD( y, 09, 01), YMD( y, 12, 31 ) )
  SELECT q_name,
         SalesRepresentative,
         count(*) AS num_orders,
         SUM( p.UnitPrice * i.Quantity ) AS total_sales
  FROM thirds LEFT OUTER JOIN SalesOrders AS o
    ON OrderDate BETWEEN q_start and q_end
           KEY JOIN SalesOrderItems AS I
           KEY JOIN Products AS p
  GROUP BY q_name, SalesRepresentative
  ORDER BY q_name, SalesRepresentative;
END;
```

The following statement calls the previous procedure.

```
CALL sales_by_third (2000);
```

Recursive common table expressions

Common table expressions may be recursive. Common table expressions are recursive when the RECURSIVE keyword appears immediately after WITH. A single WITH clause may contain multiple recursive expressions, and may contain both recursive and non-recursive common table expressions.

Recursion provides an easier means of traversing tables that represent tree or tree-like data structures. The only way to traverse such a structure in a single statement without using recursive expressions is to join the table to itself once for each possible level. For example, if a reporting hierarchy contains at most seven levels, you must join the Employees table to itself seven times. If the company reorganizes and a new management level is introduced, you must rewrite the query.

Recursive common table expressions provide a convenient way to write queries that return relationships to an arbitrary depth. For example, given a table that represents the reporting relationships within a company, you can readily write a query that returns all the employees that report to one particular person.

For example, consider the problem of determining which department has the most employees. The Employees table in the SQL Anywhere sample database lists all the employees in a fictional company and specifies in which department each works. The following query lists the department ID codes and the total number of employees in each department.

```
SELECT DepartmentID, COUNT( * ) AS n
FROM Employees
GROUP BY DepartmentID;
```

This query can be used to extract the department with the most employees as follows:

```
SELECT DepartmentID, n
FROM ( SELECT DepartmentID, COUNT( * ) AS n
       FROM Employees GROUP BY DepartmentID ) AS a
WHERE a.n =
  ( SELECT MAX( n )
    FROM ( SELECT DepartmentID, COUNT( * ) AS n
           FROM Employees GROUP BY DepartmentID ) AS b );
```

While this statement provides the correct result, it has some disadvantages. The first disadvantage is that the repeated subquery makes this statement less efficient. The second is that this statement provides no clear link between the subqueries.

One way around these problems is to create a view, then use it to re-express the query. This approach avoids the problems mentioned above.

```
CREATE VIEW CountEmployees( DepartmentID, n ) AS
   SELECT DepartmentID, COUNT( * ) AS n
   FROM Employees GROUP BY DepartmentID;

SELECT DepartmentID, n
   FROM CountEmployees
   WHERE n = ( SELECT MAX( n )
               FROM CountEmployees );
```

The disadvantage of this approach is that some overhead is required, as the database server must update the system tables when creating the view. If the view will be used frequently, this approach is reasonable. However, when the view is used only once within a particular SELECT statement, the preferred method is to instead use a common table expression. For more information about common table expressions, see "Using common table expressions" on page 430.

Recursive common table expressions contain an **initial subquery**, or seed, and a **recursive subquery** that during each iteration appends additional rows to the result set. The two parts can be connected only with the operator UNION ALL. The initial subquery is an ordinary non-recursive query and is processed first. The recursive portion contains a reference to the rows added during the previous iteration. Recursion stops automatically whenever an iteration generates no new rows. There is no way to reference rows selected before the previous iteration.

The select list of the recursive subquery must match that of the initial subquery in number and data type. If automatic translation of data types cannot be performed, explicitly cast the results of one subquery so that they match those in the other subquery.

Selecting hierarchical data

Depending on how you write the query, you may want to limit the number of levels of recursion. Limiting the number of levels permits you to return only the top levels of management, for example, but may exclude some employees if the chains of command are longer than you anticipated. Providing no restriction on the number of levels ensures no employees are excluded, but can introduce infinite recursion should the execution require any cycles; for example, if an employee directly or indirectly reports to himself. This situation could arise within a company's management hierarchy if, for example, an employee within the company also sits on the board of directors.

The following query demonstrates how to list the employees by management level. Level 0 represents employees with no managers. Level 1 represents employees who report directly to one of the level 0 managers, level 2 represents employees who report directly to a level 1 manager, and so on.

```
WITH RECURSIVE
  manager ( EmployeeID, ManagerID,
            GivenName, Surname, mgmt_level ) AS
( ( SELECT EmployeeID, ManagerID,       -- initial subquery
           GivenName, Surname, 0
    FROM Employees AS e
    WHERE ManagerID = EmployeeID )
  UNION ALL
  ( SELECT e.EmployeeID, e.ManagerID,   -- recursive subquery
           e.GivenName, e.Surname, m.mgmt_level + 1
    FROM Employees AS e JOIN manager AS m
     ON   e.ManagerID =  m.EmployeeID
      AND e.ManagerID <> e.EmployeeID
      AND m.mgmt_level < 20 ) )
SELECT * FROM manager
ORDER BY mgmt_level, Surname, GivenName;
```

The condition within the recursive query that restricts the management level to less than 20 is an important precaution. It prevents infinite recursion if the table data contains a cycle.

max_recursive_iterations option

The max_recursive_iterations option is designed to catch runaway recursive queries. The default value of this option is 100. Recursive queries that exceed this number of levels of recursion end, but cause an error.

Although this option may seem to diminish the importance of a stop condition, this is not usually the case. The number of rows selected during each iteration may grow exponentially, seriously impacting database performance before the maximum is reached. Stop conditions within recursive queries provide a means of setting appropriate limits in each situation.

Restrictions on recursive common table expressions

The following restrictions apply to recursive common table expressions.

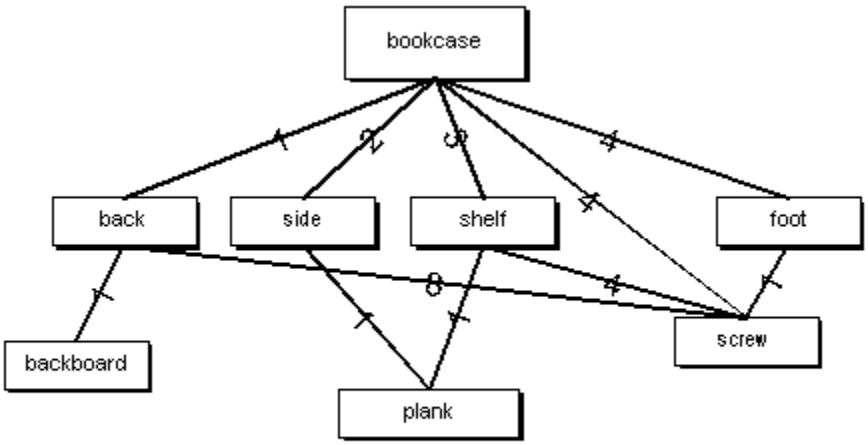
- References to other recursive common table expressions cannot appear within the definition of recursive common table expressions. So, recursive common table expressions cannot be mutually recursive. However, non-recursive common table expressions can contain references to recursive ones, and recursive common table expressions can contain references to non-recursive ones.
- The only set operator permitted between the initial subquery and the recursive subquery is UNION ALL. No other set operators are permitted.
- Within the definition of a recursive subquery, a self-reference to the recursive table expression can appear only within the FROM clause of the recursive subquery.
- When a self-reference appears within the FROM clause of the recursive subquery, the reference to the recursive table cannot appear on the null-supplying side of an outer join.

- The recursive subquery cannot contain DISTINCT, or a GROUP BY or an ORDER BY clause.
- The recursive subquery can not make use of any aggregate function.
- To prevent runaway recursive queries, an error is generated if the number of levels of recursion exceeds the current setting of the max_recursive_iterations option. The default value of this option is 100.

Parts explosion problems

The parts explosion problem is a classic application of recursion. In this problem, the components necessary to assemble a particular object are represented by a graph. The goal is to represent this graph using a database table, then to calculate the total number of the necessary elemental parts.

For example, the following graph represents the components of a simple bookshelf. The bookshelf is made up of three shelves, a back, and four feet that are held on by four screws. Each shelf is a board held on with four screws. The back is another board held on by eight screws.



The information in the table below represents the edges of the bookshelf graph. The first column names a component, the second column names one of the subcomponents of that component, and the third column specifies how many of the subcomponents are required.

component	subcomponent	quantity
bookcase	back	1
bookcase	side	2

component	subcomponent	quantity
bookcase	shelf	3
bookcase	foot	4
bookcase	screw	4
back	backboard	1
back	screw	8
side	plank	1
shelf	plank	1
shelf	screw	4

Execute the following statements to create the bookcase table and insert component and subcomponent data.

```
CREATE TABLE bookcase (
      component      VARCHAR(9),
      subcomponent   VARCHAR(9),
      quantity       INTEGER,
    PRIMARY KEY ( component, subcomponent )
);
INSERT INTO bookcase
  SELECT 'bookcase', 'back',      1 UNION
  SELECT 'bookcase', 'side',      2 UNION
  SELECT 'bookcase', 'shelf',     3 UNION
  SELECT 'bookcase', 'foot',      4 UNION
  SELECT 'bookcase', 'screw',     4 UNION
  SELECT 'back',     'backboard', 1 UNION
  SELECT 'back',     'screw',     8 UNION
  SELECT 'side',     'plank',     1 UNION
  SELECT 'shelf',    'plank',     1 UNION
  SELECT 'shelf',    'screw',     4;
```

Execute the following statement to generate a list of components and subcomponents and the quantity required to assemble the bookcase.

```
SELECT * FROM bookcase
ORDER BY component, subcomponent;
```

Execute the following statement to generate a list of subcomponents and the quantity required to assemble the bookcase.

```
WITH RECURSIVE parts ( component, subcomponent, quantity ) AS
( SELECT component, subcomponent, quantity
  FROM bookcase WHERE component = 'bookcase'
    UNION ALL
  SELECT b.component, b.subcomponent, p.quantity * b.quantity
  FROM parts p JOIN bookcase b ON p.subcomponent = b.component )
SELECT subcomponent, SUM( quantity ) AS quantity
FROM parts
WHERE subcomponent NOT IN ( SELECT component FROM bookcase )
```

```
GROUP BY subcomponent
ORDER BY subcomponent;
```

The results of this query are shown below.

subcomponent	quantity
backboard	1
foot	4
plank	5
screw	24

Alternatively, you can rewrite this query to perform an additional level of recursion, and avoid the need for the subquery in the main SELECT statement. The results of the following query are identical to those of the previous query.

```
WITH RECURSIVE parts ( component, subcomponent, quantity ) AS
( SELECT component, subcomponent, quantity
  FROM bookcase WHERE component = 'bookcase'
    UNION ALL
  SELECT p.subcomponent, b.subcomponent,
    IF b.quantity IS NULL
    THEN p.quantity
    ELSE p.quantity * b.quantity
    ENDIF
  FROM parts p LEFT OUTER JOIN bookcase b
  ON p.subcomponent = b.component
    WHERE p.subcomponent IS NOT NULL
 )
SELECT component, SUM( quantity ) AS quantity
FROM parts
WHERE subcomponent IS NULL
GROUP BY component
ORDER BY component;
```

Data type declarations in recursive common table expressions

The data types of the columns in the temporary view are defined by those of the initial subquery. The data types of the columns from the recursive subquery must match. The database server automatically attempts to convert the values returned by the recursive subquery to match those of the initial query. If this is not possible, or if information may be lost in the conversion, an error is generated.

In general, explicit casts are often required when the initial subquery returns a literal value or NULL. Explicit casts may also be required when the initial subquery selects values from different columns than the recursive subquery.

Casts may be required if the columns of the initial subquery do not have the same domains as those of the recursive subquery. Casts must always be applied to NULL values in the initial subquery.

For example, the bookshelf parts explosion sample works correctly because the initial subquery returns rows from the bookcase table, and inherits the data types of the selected columns. See "Parts explosion problems" on page 438.

If this query is rewritten as follows, explicit casts are required.

```
WITH RECURSIVE parts ( component, subcomponent, quantity ) AS
( SELECT NULL, 'bookcase', 1         -- ERROR! Wrong domains!
    UNION ALL
  SELECT b.component, b.subcomponent,
         p.quantity * b.quantity
  FROM parts p JOIN bookcase b
    ON p.subcomponent = b.component )
SELECT * FROM parts
ORDER BY component, subcomponent;
```

Without casting, errors result for the following reasons:

- The correct data type for component names is VARCHAR, but the first column is NULL.
- The digit 1 is assumed to be a SMALL INT, but the data type of the quantity column is INT.

No cast is required for the second column because this column of the initial query is already a string.

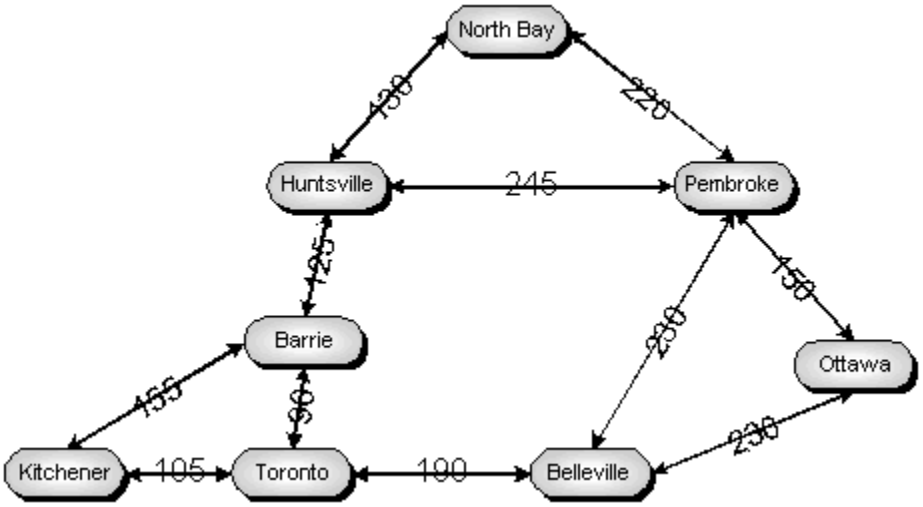
Casting the data types in the initial subquery allows the query to behave as intended:

```
WITH RECURSIVE parts ( component, subcomponent, quantity ) AS
( SELECT CAST( NULL AS VARCHAR ),  -- CASTs must be used
         'bookcase',               -- to declare the
         CAST( 1   AS INT )        -- correct datatypes
      UNION ALL
  SELECT b.component, b.subcomponent,
         p.quantity * b.quantity
  FROM parts p JOIN bookcase b
    ON p.subcomponent = b.component )
SELECT * FROM parts
ORDER BY component, subcomponent;
```

Least distance problem

You can use recursive common table expressions to find desirable paths on a directed graph. Each row in a database table represents a directed edge. Each row specifies an origin, a destination, and a cost of traveling from the origin to the destination. Depending on the problem, the cost may represent distance, travel time, or some other measure. Recursion permits you to explore possible routes through this graph. From the set of possible routes, you can then select the ones that interest you.

For example, consider the problem of finding a desirable way to drive between the cities of Kitchener and Pembroke. There are quite a few possible routes, each of which takes you through a different set of intermediate cities. The goal is to find the shortest routes, and to compare them to reasonable alternatives.



First, define a table to represent the edges of this graph and insert one row for each edge. Since all the edges of this graph are bi-directional, the edges that represent the reverse directions must be inserted also. This is done by selecting the initial set of rows, but interchanging the origin and destination. For example, one row must represent the trip from Kitchener to Toronto, and another row the trip from Toronto back to Kitchener.

```
CREATE TABLE travel (
    origin      VARCHAR(10),
    destination VARCHAR(10),
    distance    INT,
  PRIMARY KEY ( origin, destination )
);
INSERT INTO travel
  SELECT 'Kitchener',  'Toronto',    105 UNION
  SELECT 'Kitchener',  'Barrie',     155 UNION
  SELECT 'North Bay',  'Pembroke',   220 UNION
  SELECT 'Pembroke',   'Ottawa',     150 UNION
  SELECT 'Barrie',     'Toronto',     90 UNION
  SELECT 'Toronto',    'Belleville', 190 UNION
  SELECT 'Belleville', 'Ottawa',     230 UNION
  SELECT 'Belleville', 'Pembroke',   230 UNION
  SELECT 'Barrie',     'Huntsville', 125 UNION
  SELECT 'Huntsville', 'North Bay',  130 UNION
  SELECT 'Huntsville', 'Pembroke',   245;
INSERT INTO travel   -- Insert the return trips
  SELECT destination, origin, distance
  FROM travel;
```

The next task is to write the recursive common table expression. Since the trip starts in Kitchener, the initial subquery begins by selecting all the possible paths out of Kitchener, along with the distance of each.

The recursive subquery extends the paths. For each path, it adds segments that continue along from the destinations of the previous segments, and adds the length of the new segments to maintain a running total cost of each route. For efficiency, routes end if they meet either of the following conditions:

- The path returns to the starting location.

- The path returns to the previous location.
- The path reaches the final destination.

In the current example, no path should return to Kitchener and all paths should end if they reach Pembroke.

When using recursive queries to explore cyclic graphs, it is important to verify that they finish properly. In this case, the above conditions are insufficient, as a route may include an arbitrarily large number of trips back and forth between two intermediate cities. The recursive query below guarantees an end by limiting the maximum number of segments in any given route to seven.

Since the point of the example query is to select a practical route, the main query selects only those routes that are less than 50 percent longer than the shortest route.

```
WITH RECURSIVE
    trip ( route, destination, previous, distance, segments ) AS
( SELECT CAST( origin || ', ' || destination AS VARCHAR(256) ),
         destination, origin, distance, 1
  FROM travel
  WHERE origin = 'Kitchener'
    UNION ALL
  SELECT route || ', ' || v.destination,
         v.destination,             -- current endpoint
         v.origin,                  -- previous endpoint
         t.distance + v.distance,   -- total distance
         segments + 1               -- total number of segments
  FROM trip t JOIN travel v ON t.destination = v.origin
  WHERE v.destination <> 'Kitchener'  -- Don't return to start
    AND v.destination <> t.previous   -- Prevent backtracking
    AND v.origin      <> 'Pembroke'   -- Stop at the end
    AND segments                      -- TERMINATE RECURSION!
          < ( SELECT count(*)/2 FROM travel ) )
SELECT route, distance, segments FROM trip
WHERE destination = 'Pembroke' AND
      distance < 1.5 * ( SELECT MIN( distance )
                         FROM trip
                         WHERE destination = 'Pembroke' )
ORDER BY distance, segments, route;
```

When run with against the above data set, this statement yields the following results.

route	distance	segments
Kitchener, Barrie, Huntsville, Pembroke	525	3
Kitchener, Toronto, Belleville, Pembroke	525	3
Kitchener, Toronto, Barrie, Huntsville, Pembroke	565	4
Kitchener, Barrie, Huntsville, North Bay, Pembroke	630	4
Kitchener, Barrie, Toronto, Belleville, Pembroke	665	4
Kitchener, Toronto, Barrie, Huntsville, North Bay, Pembroke	670	5

route	distance	segments
Kitchener, Toronto, Belleville, Ottawa, Pembroke	675	4

Using multiple recursive common table expressions

A recursive query may include multiple recursive queries, as long as they are disjoint. It may also include a mix of recursive and non-recursive common table expressions. The RECURSIVE keyword must be present if at least one of the common table expressions is recursive.

For example, the following query—which returns the same result as the previous query—uses a second, non-recursive common table expression to select the length of the shortest route. The definition of the second common table expression is separated from the definition of the first by a comma.

```
WITH RECURSIVE
  trip ( route, destination, previous, distance, segments ) AS
    ( SELECT CAST( origin || ', ' || destination AS VARCHAR(256) ),
             destination, origin, distance, 1
      FROM travel
      WHERE origin = 'Kitchener'
      UNION ALL
      SELECT route || ', ' || v.destination,
             v.destination,
             v.origin,
             t.distance + v.distance,
             segments + 1
      FROM trip t JOIN travel v ON t.destination = v.origin
      WHERE v.destination <> 'Kitchener'
        AND v.destination <> t.previous
        AND v.origin      <> 'Pembroke'
        AND segments
              < ( SELECT count(*)/2 FROM travel ) ),
  shortest ( distance ) AS                  -- Additional,
    ( SELECT MIN(distance)                  -- non-recursive
      FROM trip                             -- common table
      WHERE destination = 'Pembroke' )      -- expression
SELECT route, distance, segments FROM trip
WHERE destination = 'Pembroke' AND
      distance < 1.5 * ( SELECT distance FROM shortest )
ORDER BY distance, segments, route;
```

Like non-recursive common table expressions, recursive expressions, when used within stored procedures, may contain references to local variables or procedure parameters. For example, the best_routes procedure, defined below, identifies the shortest routes between the two named cities.

```
CREATE PROCEDURE best_routes (
   IN initial VARCHAR(10),
   IN final   VARCHAR(10)
)
BEGIN
  WITH RECURSIVE
      trip ( route, destination, previous, distance, segments ) AS
  ( SELECT CAST( origin || ', ' || destination AS VARCHAR(256) ),
           destination, origin, distance, 1
    FROM travel
    WHERE origin = initial
      UNION ALL
```



```
    SELECT route || ', ' || v.destination,
           v.destination,            -- current endpoint
           v.origin,                 -- previous endpoint
           t.distance + v.distance,  -- total distance
           segments + 1              -- total number of segments
    FROM trip t JOIN travel v ON t.destination = v.origin
    WHERE v.destination <> initial     -- Don't return to start
      AND v.destination <> t.previous  -- Prevent backtracking
      AND v.origin      <> final       -- Stop at the end
      AND segments                     -- TERMINATE RECURSION!
            < ( SELECT count(*)/2 FROM travel ) )
  SELECT route, distance, segments FROM trip
  WHERE destination = final AND
        distance < 1.4 * ( SELECT MIN( distance )
                           FROM trip
                           WHERE destination = final )
  ORDER BY distance, segments, route;
END;
```

The following statement calls the previous procedure.

```
CALL best_routes ( 'Pembroke', 'Kitchener' );
```

OLAP support

On-Line Analytical Processing (OLAP) offers the ability to perform complex data analysis within a single SQL statement, increasing the value of the results, while improving performance by decreasing the amount of querying on the database. OLAP functionality is made possible in SQL Anywhere through the use of extensions to SQL statements and window functions. These SQL extensions and functions provide the ability, in a concise way, to perform multidimensional data analysis, data mining, time series analyses, trend analysis, cost allocations, goal seeking, and exception alerting, often with a single SQL statement.

- **Extensions to the SELECT statement** Extensions to the SELECT statement allow you to group input rows, analyze the groups, and include the findings in the final result set. These extensions include extensions to the GROUP BY clause (GROUPING SETS, CUBE, and ROLLUP subclauses), and the WINDOW clause.

  The extensions to the GROUP BY clause allow you to partition the input rows in multiple ways, yielding a result set that concatenates the different groups together. You can also create a sparse, multi-dimensional result set for data mining analyses (also known as a **data cube**). Finally, the extensions provide sub-total and grand-total rows to make analysis more convenient. See "GROUP BY clause extensions" on page 446.

  The WINDOW clause is used in conjunction with window functions to provide additional analysis opportunities on groups of input rows. See "Window functions" on page 456.

- **Window aggregate functions** Almost all SQL Anywhere aggregate functions support the concept of a configurable sliding **window** that moves down through the input rows as they are processed. Additional calculations can be performed on data in the window as it moves, allowing further analysis in a manner that is more efficient than using semantically equivalent self-join queries, or correlated subqueries.

For example, window aggregate functions, coupled with the CUBE, ROLLUP, and GROUPING SETS extensions to the GROUP BY clause, provide an efficient mechanism to compute percentiles, moving averages, and cumulative sums in a single SQL statement that would otherwise require self-joins, correlated subqueries, temporary tables, or some combination of all three.

You can use window aggregate functions to obtain such information as the quarterly moving average of the Dow Jones Industrial Average, or all employees and their cumulative salaries for each department. You can also use them to compute variance, standard deviation, correlation, and regression measures. See "Window aggregate functions" on page 462.

- **Window ranking functions** Window ranking functions allow you to form single-statement SQL queries to obtain information such as the top 10 products shipped this year by total sales, or the top 5% of salespersons who sold orders to at least 15 different companies. See "Window ranking functions" on page 481.

Improving OLAP performance

To improve OLAP performance, set the optimization_workload database option to OLAP to instruct the optimizer to consider using the Clustered Group By Hash operator in the possibilities it investigates. You can also tune indexes for OLAP workloads using the FOR OLAP WORKLOAD option when defining the index. Using this option causes the database server to perform certain optimizations which include maintaining a statistic used by the Clustered Group By Hash operator regarding the maximum page distance between two rows within the same key.

See also

- "optimization_workload option" [*SQL Anywhere Server - Database Administration*]
- "ClusteredHashGroupBy algorithm (GrByHClust)" on page 582
- "CREATE INDEX statement" [*SQL Anywhere Server - SQL Reference*]
- "CREATE TABLE statement" [*SQL Anywhere Server - SQL Reference*]
- "ALTER TABLE statement" [*SQL Anywhere Server - SQL Reference*]

GROUP BY clause extensions

The standard GROUP BY clause of a SELECT statement allows you to group rows in the result set according the grouping expressions you supply. For example, if you specify `GROUP BY columnA, columnB`, the rows are grouped by combinations of unique values from columnA and columnB. In the standard GROUP BY clause, the groups reflect the evaluation of the combination of all specified GROUP BY expressions.

However, you may want to specify different groupings or subgroupings of the result set. For example, you may want to your results to show your data grouped by unique values of columnA and columnB, and then regrouped again by unique values of columnC. You can achieve this result using the GROUPING SETS extension to the GROUP BY clause.

GROUP BY GROUPING SETS

The GROUPING SETS clause is an extension to the GROUP BY clause of a SELECT statement. The GROUPING SETS clause allows you to group your results multiple ways, without having to use multiple SELECT statements to do so. This means you can reduce response time and improve performance.

For example, the following two queries statements are semantically equivalent. However, the second query defines the grouping criteria more efficiently using a GROUP BY GROUPING SETS clause.

Multiple groupings using multiple SELECT statements:

```
SELECT NULL, NULL, NULL, COUNT( * ) AS Cnt
FROM Customers
WHERE State IN ( 'MB' , 'KS' )
   UNION ALL
SELECT City, State, NULL, COUNT( * ) AS Cnt
FROM Customers
WHERE State IN ( 'MB' , 'KS' )
GROUP BY City, State
   UNION ALL
SELECT NULL, NULL, CompanyName, COUNT( * ) AS Cnt
FROM Customers
WHERE State IN ( 'MB' , 'KS' )
GROUP BY CompanyName;
```

Multiple groupings using GROUPING SETS:

```
SELECT City, State, CompanyName, COUNT( * ) AS Cnt
FROM Customers
WHERE State IN ( 'MB' , 'KS' )
GROUP BY GROUPING SETS( ( City, State ), ( CompanyName ) , ( ) );
```

Both methods produce the same results, shown below:

	City	State	CompanyName	Cnt
1	(NULL)	(NULL)	(NULL)	8
2	(NULL)	(NULL)	Cooper Inc.	1
3	(NULL)	(NULL)	Westend Dealers	1
4	(NULL)	(NULL)	Toto's Active Wear	1
5	(NULL)	(NULL)	North Land Trading	1
6	(NULL)	(NULL)	The Ultimate	1
7	(NULL)	(NULL)	Molly's	1
8	(NULL)	(NULL)	Overland Army Navy	1
9	(NULL)	(NULL)	Out of Town Sports	1

	City	State	CompanyName	Cnt
10	'Pembroke'	'MB'	(NULL)	4
11	'Petersburg'	'KS'	(NULL)	1
12	'Drayton'	'KS'	(NULL)	3

Rows 2-9 are the rows generated by grouping over CompanyName, rows 10-12 are rows generated by grouping over the combination of City and State, and row 1 is the grand total represented by the empty grouping set, specified using a pair of matched parentheses (). The empty grouping set represents a single partition of all the rows in the input to the GROUP BY.

Notice how NULL values are used as placeholders for any expression that is not used in a grouping set, because the result sets must be combinable. For example, rows 2-9 result from the second grouping set in the query (CompanyName). Since that grouping set did not include City or State as expressions, for rows 2-9 the values for City and State contain the placeholder NULL, while the values in CompanyName contain the distinct values found in CompanyName.

Because NULLs as used as placeholders, it is easy to confuse placeholder NULLs with actual NULLs found in the data. To help distinguish placeholder NULLs from NULL data, use the GROUPING function. See "Detecting placeholder NULLs using the GROUPING function" on page 454.

Example

The following example shows how you can tailor the results that are returned from a query using GROUPING SETS, and an ORDER BY clause to better organize the results. The query returns the total number of orders by Quarter in each Year, and a total for each Year. Ordering by Year and then Quarter makes the results easier to understand:

```
SELECT Year( OrderDate ) AS Year,
        Quarter( OrderDate ) AS Quarter,
        COUNT (*) AS Orders
FROM SalesOrders
GROUP BY GROUPING SETS ( ( Year, Quarter ), ( Year ) )
ORDER BY Year, Quarter;
```

This query returns the following results:

	Year	Quarter	Orders
1	2000	(NULL)	380
2	2000	1	87
3	2000	2	77
4	2000	3	91
5	2000	4	125

	Year	Quarter	Orders
6	2001	(NULL)	268
7	2001	1	139
8	2001	2	119
9	2001	3	10

Rows 1 and 6 are subtotals of orders for Year 2000 and Year 2001, respectively. Rows 2-5 and rows 7-9 are the detail rows for the subtotal rows. That is, they show the total orders per quarter, per year.

There is no grand total for all quarters in all years in the result set. To do that, the query must include the empty grouping specification '()' in the GROUPING SETS specification.

Specifying an empty grouping specification

If you use an empty GROUPING SETS specification '()' in the GROUP BY clause, this results in a grand total row for all things that are being totaled in the results. With a grand total row, all values for all grouping expressions contain placeholder NULLs. You can use the GROUPING function to distinguish placeholder NULLs from actual NULLs resulting from the evaluation of values in the underlying data for the row. See “Detecting placeholder NULLs using the GROUPING function” on page 454.

Specifying duplicate grouping sets

You can specify duplicate grouping specifications in a GROUPING SETS clause. In this case, the result of the SELECT statement contains identical rows.

The following query includes duplicate groupings:

```
SELECT City, COUNT( * ) AS Cnt
FROM Customers
WHERE State IN ( 'MB' , 'KS' )
GROUP BY GROUPING SETS( ( City ), ( City ) );
```

This query returns the following results. Note that as a result of the duplicate groupings, rows 1-3 are identical to rows 4-6:

	City	Cnt
1	'Drayton'	3
2	'Petersburg'	1
3	'Pembroke'	4
4	'Drayton'	3
5	'Petersburg'	1

	City	Cnt
6	'Pembroke'	4

Practicing good form

Grouping syntax is interpreted differently for a GROUP BY GROUPING SETS clause than it is for a simple GROUP BY clause. For example, GROUP BY (X, Y) returns results grouped by distinct combinations of X and Y values. However, GROUP BY GROUPING SETS (X, Y) specifies two individual grouping sets, and the result of the two groupings are UNIONed together. That is, results are grouped by (X), and then unioned to the same results grouped by (Y).

For good form, and to avoid any ambiguity for complex expressions, use parentheses around each individual grouping set in the specification whenever there is a possibility for error. For example, while both of the following statements are correct and semantically equivalent, the second one reflects the recommended form:

```
SELECT * FROM t GROUP BY GROUPING SETS ( X, Y );
SELECT * FROM t GROUP BY GROUPING SETS( ( X ), ( Y ) );
```

Using ROLLUP and CUBE as a shortcut to GROUPING SETS

Using GROUPING SETS is useful when you want to concatenate several different data partitions into a single result set. However, if you have many groupings to specify, and want subtotals included, you may want to use the ROLLUP and CUBE extensions.

The ROLLUP and CUBE clauses can be considered shortcuts for predefined GROUPING SETS specifications.

ROLLUP is equivalent to specifying a series of grouping set specifications starting with the empty grouping set '()' and successively followed by grouping sets where one additional expression is concatenated to the previous one. For example, if you have three grouping expressions, a, b, and c, and you specify ROLLUP, it is as though you specified a GROUPING SETS clause with the sets: (), (a), (a, b), and (a, b, c). This construction is sometimes referred to as hierarchical groupings.

CUBE offers even more groupings. Specifying CUBE is equivalent to specifying all possible GROUPING SETS. For example, if you have the same three grouping expressions, a, b, and c, and you specify CUBE, it is as though you specified a GROUPING SETS clause with the sets: (), (a), (a, b), (a, c), (b), (b, c), (c), and (a, b, c).

When specifying ROLLUP or CUBE, use the GROUPING function to distinguish placeholder NULLs in your results, caused by the subtotal rows that are implicit in a result set formed by ROLLUP or CUBE. See “Detecting placeholder NULLs using the GROUPING function” on page 454.

Using ROLLUP

A common requirement of many applications is to compute subtotals of the grouping attributes from left-to-right, in sequence. This pattern is referred to as a hierarchy because the introduction of additional subtotal calculations produces additional rows with finer granularity of detail. In SQL Anywhere, you can specify a hierarchy of grouping attributes using the ROLLUP keyword to specify a ROLLUP clause.

A query using a ROLLUP clause produces a hierarchical series of grouping sets, as follows. If the ROLLUP clause contains *n* GROUP BY expressions of the form (X_1,X_2, ... , X_n) then the ROLLUP clause generates *n* + 1 grouping sets as:

```
{(), (X1), (X1,X2), (X1,X2,X3), ... , (X1,X2,X3, ... , Xn)}
```

Example

The following query summarizes the sales orders by year and quarter, and returns the result set shown in the table below:

```
SELECT QUARTER( OrderDate ) AS Quarter,
       YEAR( OrderDate ) AS Year,
       COUNT( * ) AS Orders,
       GROUPING( Quarter ) AS GQ,
       GROUPING( Year ) AS GY
FROM SalesOrders
GROUP BY ROLLUP( Year, Quarter )
ORDER BY Year, Quarter;
```

This query returns the following results:

	Quarter	Year	Orders	GQ	GY
1	(NULL)	(NULL)	648	1	1
2	(NULL)	2000	380	1	0
3	1	2000	87	0	0
4	2	2000	77	0	0
5	3	2000	91	0	0
6	4	2000	125	0	0
7	(NULL)	2001	268	1	0
8	1	2001	139	0	0
9	2	2001	119	0	0
10	3	2001	10	0	0

The first row in a result set shows the grand total (648) of all orders, for all quarters, for both years.

Row 2 shows total orders (380) for year 2000, while rows 3-6 show the order subtotals, by quarter, for the same year. Likewise, row 7 shows total Orders (268) for year 2001, while rows 8-10 show the subtotals, by quarter, for the same year.

Note how the values returned by GROUPING function can be used to differentiate subtotal rows from the row that contains the grand total. For rows 2 and 7, the presence of NULL in the quarter column, and the value of 1 in the GQ column (Grouping by Quarter), indicate that the row is a totaling of orders in all quarters (per year).

Likewise, in row 1, the presence of NULL in the Quarter and Year columns, plus the presence of a 1 in the GQ and GY columns, indicate that the row is a totaling of orders for all quarters and for all years.

For more information about the syntax for the ROLLUP clause, see "GROUP BY clause" [*SQL Anywhere Server - SQL Reference*].

Support for Transact-SQL WITH ROLLUP syntax

Alternatively, you can also use the Transact-SQL compatible syntax, WITH ROLLUP, to achieve the same results as GROUP BY ROLLUP. However, the syntax is slightly different and you can only supply a simple GROUP BY expression list in the syntax.

The following query produces an identical result to that of the previous GROUP BY ROLLUP example:

```
SELECT QUARTER( OrderDate ) AS Quarter,
       YEAR( OrderDate ) AS Year,
       COUNT( * ) AS Orders,
       GROUPING( Quarter ) AS GQ,
       GROUPING( Year ) AS GY
FROM SalesOrders
GROUP BY Year, Quarter WITH ROLLUP
ORDER BY Year, Quarter;
```

Using CUBE

As an alternative to the hierarchical grouping pattern provided by the ROLLUP clause, you can also create a data cube, that is, an *n*-dimensional summarization of the input using every possible combination of GROUP BY expressions, using the CUBE clause. The CUBE clause results in a product set of all possible combinations of elements from each set of values. This can be very useful for complex data analysis.

If there are *n* GROUPING expressions of the form (X_1,X_2, ...,X_n) in a CUBE clause, then CUBE generates 2^n grouping sets as:

```
{(), (X1), (X1,X2), (X1,X2,X3), ... , (X1,X2,X3, ...,Xn),
(X2), (X2,X3), (X2,X3,X4), ... , (X2,X3,X4, ... , Xn), ... , (Xn)}.
```

Example

The following query summarizes sales orders by year, by quarter, and quarter within year, and yields the result set shown in the table below:

```
SELECT QUARTER( OrderDate ) AS Quarter,
       YEAR( OrderDate ) AS Year,
       COUNT( * ) AS Orders,
```



```
       GROUPING( Quarter ) AS GQ,
       GROUPING( Year ) AS GY
FROM SalesOrders
GROUP BY CUBE ( Year, Quarter )
ORDER BY Year, Quarter;
```

This query returns the following results:

	Quarter	Year	Orders	GQ	GY
1	(NULL)	(NULL)	648	1	1
2	1	(NULL)	226	0	1
3	2	(NULL)	196	0	1
4	3	(NULL)	101	0	1
5	4	(NULL)	125	0	1
6	(NULL)	2000	380	1	0
7	1	2000	87	0	0
8	2	2000	77	0	0
9	3	2000	91	0	0
10	4	2000	125	0	0
11	(NULL)	2001	268	1	0
12	1	2001	139	0	0
13	2	2001	119	0	0
14	3	2000	10	0	0

The first row in the result set shows the grand total (648) of all orders, for all quarters, for years 2000 and 2001 combined.

Rows 2-5 summarize sales orders by calendar quarter in any year.

Rows 6 and 11 show total Orders for years 2000, and 2001, respectively.

Rows 7-10 and rows 12-14 show the quarterly totals for years 2000, and 2001, respectively.

Note how the values returned by the GROUPING function can be used to differentiate subtotal rows from the row that contains the grand total. For rows 6 and 11, the presence of NULL in the Quarter column, and the value of 1 in the GQ column (Grouping by Quarter), indicate that the row is a totaling of Orders in all quarters for the year.

Note
The result set generated through the use of CUBE can be very large because CUBE generates an exponential number of grouping sets. For this reason, SQL Anywhere does not permit a GROUP BY clause to contain more than 64 GROUP BY expressions. If a statement exceeds this limit, it fails with SQLCODE -944 (SQLSTATE 42WA1).

For more information about the syntax for the CUBE clause, see "GROUP BY clause" [*SQL Anywhere Server - SQL Reference*].

Support for Transact-SQL WITH CUBE syntax

Alternatively, you can also use the Transact-SQL compatible syntax, WITH CUBE, to achieve the same results as GROUP BY CUBE. However, the syntax is slightly different and you can only supply a simple GROUP BY expression list in the syntax.

The following query produces an identical result to that of the previous GROUP BY CUBE example:

```
SELECT QUARTER( OrderDate ) AS Quarter,
       YEAR( OrderDate ) AS Year,
       COUNT( * ) AS Orders,
       GROUPING( Quarter ) AS GQ,
       GROUPING( Year ) AS GY
FROM SalesOrders
GROUP BY Year, Quarter WITH CUBE
ORDER BY Year, Quarter;
```

Detecting placeholder NULLs using the GROUPING function

The total and subtotal rows created by ROLLUP and CUBE contain placeholder NULLs in any column specified in the SELECT list that was not used for the grouping. This means that when you are examining your results, you cannot distinguish whether a NULL in a subtotal row is a placeholder NULL, or a NULL resulting from the evaluation of the underlying data for the row. As a result, it is also difficult to distinguish between a detail row, a subtotal row, and a grand total row.

The GROUPING function allows you to distinguish placeholder NULLs from NULLs caused by underlying data. If you specify a GROUPING function with one *group-by-expression* from the grouping set specification, the function returns a 1 if it is a placeholder NULL, and 0 if it reflects a value (perhaps NULL) present in the underlying data for that row.

For example, the following query returns the result set shown in the table below:

```
SELECT Employees.EmployeeID AS Employee,
   YEAR( OrderDate ) AS Year,
   COUNT( SalesOrders.ID ) AS Orders,
   GROUPING( Employee ) AS GE,
   GROUPING( Year ) AS GY
FROM Employees LEFT OUTER JOIN SalesOrders
   ON Employees.EmployeeID = SalesOrders.SalesRepresentative
WHERE Employees.Sex IN ( 'F' )
   AND Employees.State IN ( 'TX' , 'NY' )
GROUP BY GROUPING SETS ( ( Year, Employee ), ( Year ), ( ) )
ORDER BY Year, Employee;
```


This query returns the following results:

	Employees	Year	Orders	GE	GY
1	(NULL)	(NULL)	54	1	1
2	(NULL)	(NULL)	0	1	0
3	102	(NULL)	0	0	0
4	390	(NULL)	0	0	0
5	1062	(NULL)	0	0	0
6	1090	(NULL)	0	0	0
7	1507	(NULL)	0	0	0
8	(NULL)	2000	34	1	0
9	667	2000	34	0	0
10	(NULL)	2001	20	1	0
11	667	2001	20	0	0

In this example, row 1 represents the grand total of orders (54) because the empty grouping set '()' was specified. Notice that GE and GY both contain a 1 to indicate that the NULLs in the Employees and Year columns are placeholder NULLs for Employees and Year columns, respectively.

Row 2 is a subtotal row. The 1 in the GE column indicates that the NULL in the Employees column is a placeholder NULL. The 0 in the GY column indicates that the NULL in the Year column is the result of evaluating the underlying data, and not a placeholder NULL; in this case, this row represents those employees who have no orders.

Rows 3-7 show the total number of orders, per employee, where the Year was NULL. That is, these are the female employees that live in Texas and New York who have no orders. These are the detail rows for row 2. That is, row 2 is a totaling of rows 3-7.

Row 8 is a subtotal row showing the number of orders for all employees combined, in the year 2000. Row 9 is the single detail row for row 8.

Row 10 is a subtotal row showing the number of orders for all employees combined, in the year 2001. Row 11 is the single detail row for row 10.

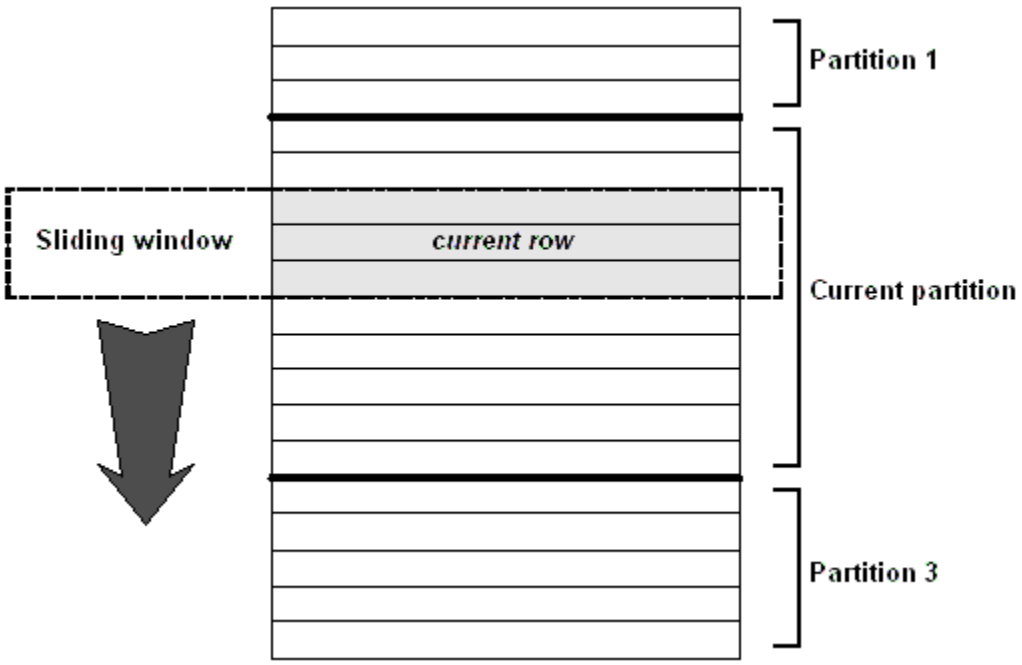
For more information about the syntax of the GROUPING function, see “GROUPING function [Aggregate]” [*SQL Anywhere Server - SQL Reference*].

Window functions

OLAP functionality includes the concept of a sliding **window** that moves down through the input rows as they are processed. Additional calculations can be performed on the data in the window as it moves, allowing further analysis in a manner that is more efficient than using semantically equivalent self-join queries, or correlated subqueries.

You configure the bounds of the window based on the information you are trying to extract from the data. A window can be one, many, or all the rows in the input data, which has been partitioned according to the grouping specifications provided in the window definition. The window moves down through the input data, incorporating the rows needed to perform the requested calculations.

The following diagram illustrates the movement of the window as input rows are processed. The data partitions reflect the grouping of input rows specified in the window definition. If no grouping is specified, all input rows are considered one partition. The length of the window (that is, the number of rows it includes), and the offset of the window compared to the current row, reflect the bounds specified in the window definition.



Defining a window

You can use SQL windowing extensions to configure the bounds of a window, and the partitioning and ordering of the input rows. Logically, as part of the semantics of computing the result of a query specification, partitions are created after the groups defined by the GROUP BY clause are created, but

before the evaluation of the final SELECT list and the query's ORDER BY clause. So, the order of evaluation of the clauses within a SQL statement is:

1. FROM
2. WHERE
3. GROUP BY
4. HAVING
5. WINDOW
6. DISTINCT
7. ORDER BY

When forming your query, the impact of the order of evaluation should be considered. For example, you cannot have a predicate on an expression referencing a window function in the same SELECT query block. However, by putting the query block in a derived table, you can specify a predicate on the derived table. The following query fails with a message indicating that the failure was the result of a predicate being specified on a window function:

```
SELECT DepartmentID, Surname, StartDate, Salary,
       SUM( Salary ) OVER ( PARTITION BY DepartmentID
          ORDER BY StartDate
          RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW ) AS
"Sum_Salary"
   FROM Employees
   WHERE State IN ( 'CA', 'UT', 'NY', 'AZ' )
      AND DepartmentID IN ( '100', '200' )
   GROUP BY DepartmentID, Surname, StartDate, Salary
   HAVING Salary > 0 AND "Sum_Salary" > 200
   ORDER BY DepartmentID, StartDate;
```

Use a derived table (DT) and specify a predicate on it to achieve the results you want:

```
SELECT * FROM ( SELECT DepartmentID, Surname, StartDate, Salary,
                       SUM( Salary ) OVER ( PARTITION BY DepartmentID
                         ORDER BY StartDate
                         RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW )
AS "Sum_Salary"
                  FROM Employees
                  WHERE State IN ( 'CA', 'UT', 'NY', 'AZ' )
                     AND DepartmentID IN ( '100', '200' )
                  GROUP BY DepartmentID, Surname, StartDate, Salary
                  HAVING Salary > 0
                  ORDER BY DepartmentID, StartDate ) AS DT
   WHERE DT.Sum_Salary > 200;
```

Because window partitioning follows a GROUP BY operator, the result of any aggregate function, such as SUM, AVG, or VARIANCE, is available to the computation done for a partition. So, windows provide another opportunity to perform grouping and ordering operations in addition to a query's GROUP BY and ORDER BY clauses.

Defining a window specification

When you define the window over which a window function operates, you specify one or more of the following:

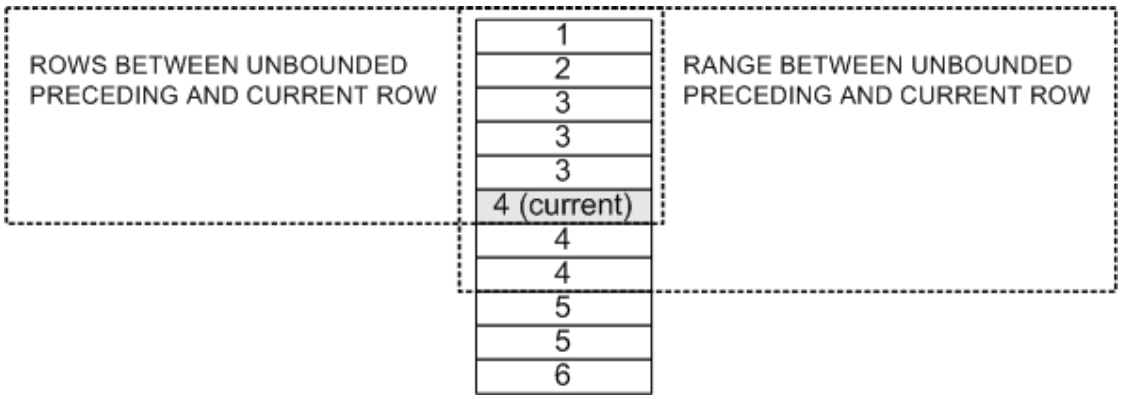
- **Partitioning (PARTITION BY clause)** The PARTITION BY clause defines how the input rows are grouped. If omitted, the entire input is treated as a single partition. A partition can be one, several, or all input rows, depending on what you specify. Data from two partitions is never mixed. That is, when a window reaches the boundary between two partitions, it completes processing the data in one partition, before beginning on the data in the next partition. This means that the window size may vary at the beginning and end of a partition, depending on how the bounds are defined for the window.

- **Ordering (ORDER BY clause)** The ORDER BY clause defines how the input rows are ordered, before being processed by the window function. The ORDER BY clause is required only if you are specifying the bounds using a RANGE clause, or if a ranking function references the window. Otherwise, the ORDER BY clause is optional. If omitted, the database server processes the input rows in the most efficient manner.

- **Bounds (RANGE and ROWS clauses)** The current row provides the reference point for determining the start and end rows of a window. You can use the RANGE and ROWS clauses of the window definition to set these bounds. RANGE defines the window as a *range of data values* offset from the value in the current row. So, if you specify RANGE, you must also specify an ORDER BY clause since range calculations require that the data be ordered.

  ROWS defines the window as *the number of rows* offset from the current row.

  Since RANGE defines a set of rows as a range of data values, the rows included in a RANGE window can include rows beyond the current row. This is different from how ROWS is handled. The following diagram illustrates the difference between the ROWS and RANGE clauses:



  Within the ROWS and RANGE clauses, you can (optionally) specify the start and end rows of the window, relative to the current row. To do this, you use the PRECEDING, BETWEEN, and FOLLOWING clauses. These clauses take expressions, and the keywords UNBOUNDED and CURRENT ROW. If no bounds are defined for a window, the default window bounds are set as follows:

- If the window specification contains an ORDER BY clause, it is equivalent to specifying RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW.
- If the window specification does not contain an ORDER BY clause, it is equivalent to specifying ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING.

The following table contains some example window bounds and description of the rows they contain:

Specification	Meaning
ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW	Start at the beginning of the partition, and end with the current row. Use this when computing cumulative results, such as cumulative sums.
ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING	Use all rows in the partition. Use this when you want the value of an aggregate function to be identical for each row of a partition.
ROWS BETWEEN *x* PRECEDING AND *y* FOLLOWING	Create a fixed-size moving window of rows starting at a distance of *x* from current row and ending at a distance of *y* from current row (inclusive). Use this example when you want to calculate a moving average, or when you want to compute differences in values between adjacent rows. With a moving window of more than one row, NULLs occur when computing the first and last row in the partition. This occurs because when the current row is either the very first or very last row of the partition, there are no preceding or following (respectively) rows to use in the computation. Therefore, NULL values are used instead.
ROWS BETWEEN CURRENT ROW AND CURRENT ROW	A window of one row; the current row.
RANGE BETWEEN 5 PRECEDING AND 5 FOLLOWING	Create a window that is based on values in the rows. For example, suppose that for the current row, the column specified in the ORDER BY clause contains the value 10. If you specify the window size to be `RANGE BETWEEN 5 PRECEDING AND 5 FOLLOWING`, you are specifying the size of the window to be as large as required to ensure that the first row contains a 5 in the column, and the last row in the window contains a 15 in the column. As the window moves down the partition, the size of the window may grow or shrink according to the size required to fulfill the range specification.

Make your window specification as explicit as possible. Otherwise, the defaults may not return the results you expect.

Use the RANGE clause to avoid problems caused by gaps in the input to a window function when the set of values is not continuous. When a window bounds are set using a RANGE clause, the database server automatically handles adjacent rows and rows with duplicate values.

RANGE uses unsigned integer values. Truncation of the range expression can occur depending on the domain of the ORDER BY expression and the domain of the value specified in the RANGE clause.

Do not specify window bounds when using a ranking or a row-numbering function.

Window definition: inlining using the OVER clause and WINDOW clause

There are three ways to define a window:

- inline (within the OVER clause of a window function)
- in a WINDOW clause
- partially inline and partially in a WINDOW clause

However, some approaches have restrictions, as noted in the following sections.

Inline definition (within the OVER clause of a window function)

A window definition can be placed in the OVER clause of a window function. This is referred to as defining the window *inline*.

For example, the following statement queries the SQL Anywhere sample database for all products shipped in July and August 2001, and the cumulative shipped quantity by shipping date. The window is defined inline.

```
SELECT p.ID, p.Description, s.Quantity, s.ShipDate,
   SUM( s.Quantity ) OVER ( PARTITION BY s.ProductID
       ORDER BY s.ShipDate
       ROWS BETWEEN UNBOUNDED PRECEDING
       AND CURRENT ROW ) AS Cumulative_qty
FROM SalesOrderItems s JOIN Products p
   ON ( s.ProductID = p.ID )
WHERE s.ShipDate BETWEEN '2001-07-01' AND '2001-08-31'
ORDER BY p.ID;
```

This query returns the following results:

	ID	Description	Quantity	ShipDate	Cumulative_qty
1	301	V-neck	24	2001-07-16	24
2	302	Crew Neck	60	2001-07-02	60
3	302	Crew Neck	36	2001-07-13	96

	ID	Description	Quantity	ShipDate	Cumulative_qty
4	400	Cotton Cap	48	2001-07-05	48
5	400	Cotton Cap	24	2001-07-19	72
6	401	Wool Cap	48	2001-07-09	48
7	500	Cloth Visor	12	2001-07-22	12
8	501	Plastic Visor	60	2001-07-07	60
9	501	Plastic Visor	12	2001-07-12	72
10	501	Plastic Visor	12	2001-07-22	84
11	601	Zipped Sweatshirt	60	2001-07-19	60
12	700	Cotton Shorts	24	2001-07-26	24

In this example, the computation of the SUM window function occurs after the join of the two tables and the application of the query's WHERE clause. The query is processed as follows:

1. Partition (group) the input rows based on the value ProductID.
2. Within each partition, sort the rows based on the value of ShipDate.
3. For each row in the partition, evaluate the SUM function over the values in Quantity, using a sliding window consisting of the first (sorted) row of each partition, up to and including the current row.

WINDOW clause definition

An alternative construction for the above query is to use a WINDOW clause to specify the window separately from the functions that use it, and then reference the window from within the OVER clause of each function.

In this example, the WINDOW clause creates a window called Cumulative, partitioning data by ProductID, and ordering it by ShipDate. The SUM function references the window in its OVER clause, and defines its size using a ROWS clause.

```
SELECT p.ID, p.Description, s.Quantity, s.ShipDate,
    SUM( s.Quantity ) OVER ( Cumulative
    ROWS BETWEEN UNBOUNDED PRECEDING
    AND CURRENT ROW ) AS cumulative_qty
FROM SalesOrderItems s
JOIN Products p ON ( s.ProductID = p.ID )
WHERE s.ShipDate BETWEEN '2001-07-01' AND '2001-08-31'
WINDOW Cumulative AS ( PARTITION BY s.ProductID ORDER BY s.ShipDate )
ORDER BY p.ID;
```

When using the WINDOW clause syntax, the following restrictions apply:

- If a PARTITION BY clause is specified, it must be placed within the WINDOW clause.
- If a ROWS or RANGE clause is specified, it must be placed in the OVER clause of the referencing function.
- If an ORDER BY clause is specified for the window, it can be placed in either the WINDOW clause or the referencing function's OVER clause, but not both.
- The WINDOW clause must precede the SELECT statement's ORDER BY clause.

Combination inline and WINDOW clause definition

You can inline part of a window definition and then define the rest in the WINDOW clause. For example:

```
AVG() OVER ( windowA
             ORDER BY expression )...
...
WINDOW windowA AS ( PARTITION BY expression )
```

When splitting the window definition in this manner, the following restrictions apply:

- You cannot use a PARTITION BY clause in the window function syntax.
- You can use an ORDER BY clause in either the window function syntax or in the WINDOW clause, but not in both.
- You cannot include a RANGE or ROWS clause in the WINDOW clause.

See also

- "WINDOW clause" [*SQL Anywhere Server - SQL Reference*]
- "Window aggregate functions" on page 462
- "Window ranking functions" on page 481
- "Defining a window" on page 456

Window functions in SQL Anywhere

Functions that allow you to perform analytic operations over a set of input rows are referred to as window functions. For example, all ranking functions, and most aggregate functions, are **window functions**. You can use them to perform additional analysis on your data. This is achieved by partitioning and sorting the input rows before being processed, and then processing the rows in a configurable-sized window that moves through the input.

There are three types of window functions: window aggregate functions, window ranking functions, and row numbering functions.

Window aggregate functions

Window aggregate functions return a value for a specified set of rows in the input. For example, you can use window functions to calculate a moving average of the sales figures for a company over a specified time period.

Window aggregate functions are organized into the following three categories:

- **Basic aggregate functions** Following is the list of supported basic aggregate functions:

  - SUM
  - AVG
  - MAX
  - MIN
  - MEDIAN
  - FIRST_VALUE
  - LAST_VALUE
  - COUNT

  For more information about basic aggregate functions, see "Basic aggregate functions" on page 464.

- **Standard deviation and variance functions** Following is the list of supported standard deviation and variance functions:

  - STDDEV
  - STDDEV_POP
  - STDDEV_SAMP
  - VAR_POP
  - VAR_SAMP
  - VARIANCE

  For more information about standard deviation and variance functions, see "Standard deviation and variance functions" on page 475.

- **Correlation and linear regression functions** Following is the list of supported correlation and linear regression functions:

  - COVAR_POP
  - COVAR_SAMP
  - REGR_AVGX
  - REGR_AVGY
  - REGR_COUNT
  - REGR_INTERCEPT
  - REGR_R2
  - REGR_SLOPE
  - REGR_SXX
  - REGR_SXY
  - REGR_SYY

  For more information about correlation and linear regression functions, see "Correlation and linear regression functions" on page 479.

Basic aggregate functions

Complex data analysis often requires multiple levels of aggregation. Window partitioning and ordering, in addition to, or instead of, a GROUP BY clause, offers you considerable flexibility in the composition of complex SQL queries. For example, by combining a window construct with a simple aggregate function, you can compute values such as moving average, moving sum, moving minimum or maximum, and cumulative sum.

Following are the basic aggregate functions in SQL Anywhere:

- **SUM function** Returns the total of the specified expression for each group of rows.
- **AVG function** Returns the average of a numeric expression or of a set unique values for a set of rows.
- **MAX function** Returns the maximum expression value found in each group of rows.
- **MIN function** Returns the minimum expression value found in each group of rows.
- **MEDIAN function** Returns the median of a numeric expression for a set of rows.
- **FIRST_VALUE function** Returns values from the first row of a window. This function requires a window specification.
- **LAST_VALUE function** Returns values from the last row of a window. This function requires a window specification.
- **COUNT function** Returns the number of rows that qualify for the specified expression.

See also

- "Window functions" on page 456

SUM function example

The following example shows the SUM function used as a window function. The query returns a result set that partitions the data by DepartmentID, and then provides a cumulative summary (Sum_Salary) of employees' salaries, starting with the employee who has been at the company the longest. The result set includes only those employees who reside in California, Utah, New York, or Arizona. The column Sum_Salary provides the cumulative total of employees' salaries.

```
SELECT DepartmentID, Surname, StartDate, Salary,
SUM( Salary ) OVER ( PARTITION BY DepartmentID
    ORDER BY StartDate
    RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW )
AS "Sum_Salary"
FROM Employees
WHERE State IN ( 'CA', 'UT', 'NY', 'AZ' )
   AND DepartmentID IN ( '100', '200' )
ORDER BY DepartmentID, StartDate;
```

The table that follows represents the result set from the query. The result set is partitioned by DepartmentID.

	DepartmentID	Surname	StartDate	Salary	Sum_Salary
1	100	Whitney	1984-08-28	45700.00	45700.00
2	100	Cobb	1985-01-01	62000.00	107700.00
3	100	Shishov	1986-06-07	72995.00	180695.00
4	100	Driscoll	1986-07-01	48023.69	228718.69
5	100	Guevara	1986-10-14	42998.00	271716.69
6	100	Wang	1988-09-29	68400.00	340116.69
7	100	Soo	1990-07-31	39075.00	379191.69
8	100	Diaz	1990-08-19	54900.00	434091.69
9	200	Overbey	1987-02-19	39300.00	39300.00
10	200	Martel	1989-10-16	55700.00	95000.00
11	200	Savarino	1989-11-07	72300.00	167300.00
12	200	Clark	1990-07-21	45000.00	212300.00
13	200	Goggin	1990-08-05	37900.00	250200.00

For DepartmentID 100, the cumulative total of salaries from employees in California, Utah, New York, and Arizona is $434,091.69 and the cumulative total for employees in department 200 is $250,200.00.

For more information about the exact syntax of the SUM function, see “SUM function [Aggregate]” [*SQL Anywhere Server - SQL Reference*].

Computing deltas between adjacent rows

Using two windows—one window over the current row, the other over the previous row—you can compute deltas, or changes, between adjacent rows. For example, the following query computes the delta (Delta) between the salary for one employee and the previous employee in the results:

```
SELECT EmployeeID AS EmployeeNumber,
   Surname AS LastName,
   SUM( Salary ) OVER ( ORDER BY BirthDate
      ROWS BETWEEN CURRENT ROW AND CURRENT ROW )
      AS CurrentRow,
   SUM( Salary ) OVER ( ORDER BY BirthDate
      ROWS BETWEEN 1 PRECEDING AND 1 PRECEDING )
      AS PreviousRow,
   ( CurrentRow - PreviousRow ) AS Delta
FROM Employees
WHERE State IN ( 'NY' );
```

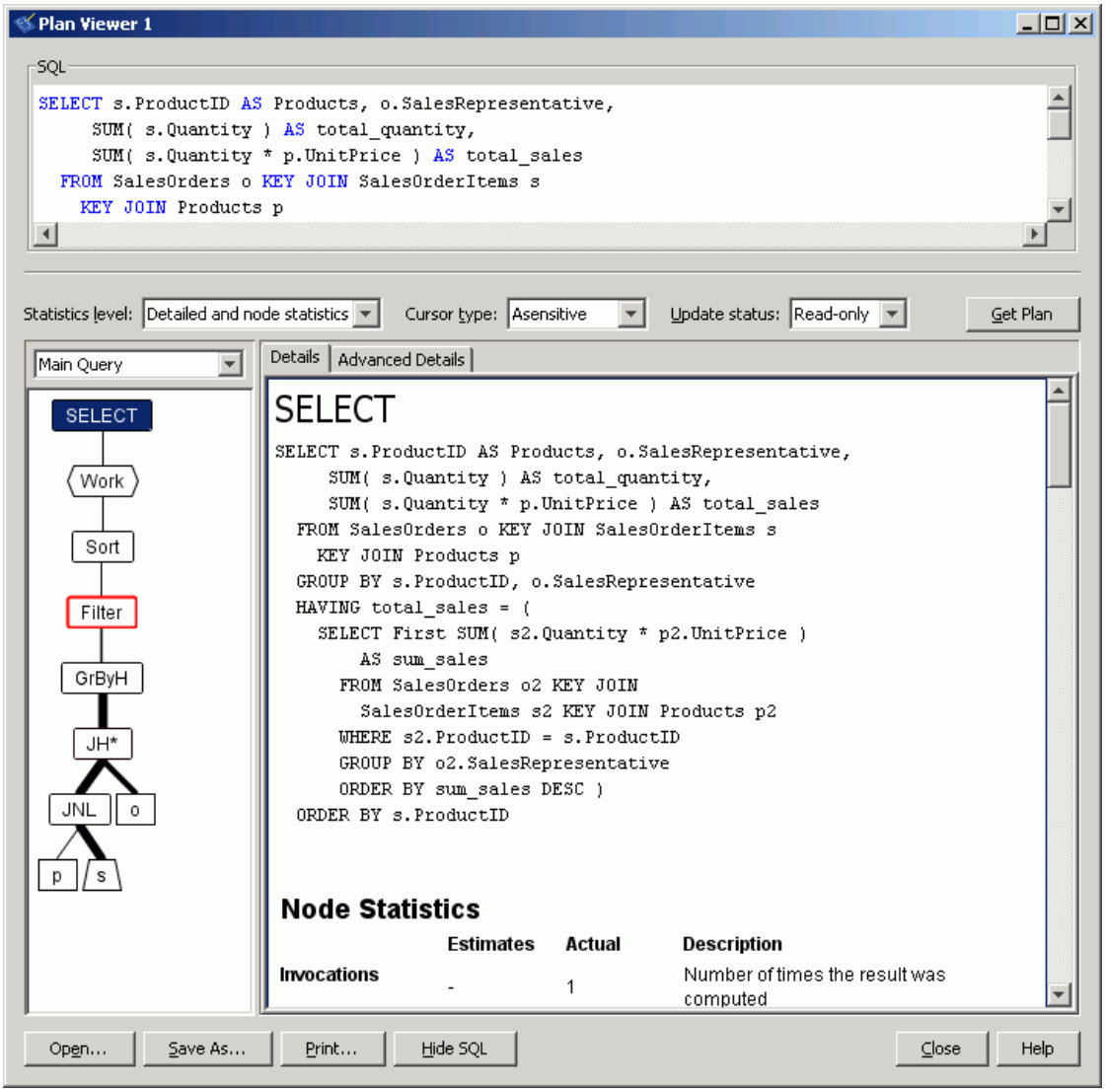
	EmployeeNumber	LastName	CurrentRow	PreviousRow	Delta
1	913	Martel	55700.000	(NULL)	(NULL)
2	1062	Blaikie	54900.000	55700.000	-800.000
3	249	Guevara	42998.000	54900.000	-11902.000
4	390	Davidson	57090.000	42998.000	14092.000
5	102	Whitney	45700.000	57090.000	-11390.000
6	1507	Wetherby	35745.000	45700.000	-9955.000
7	1751	Ahmed	34992.000	35745.000	-753.000
8	1157	Soo	39075.000	34992.000	4083.000

Note that SUM is performed only on the current row for the CurrentRow window because the window size was set to `ROWS BETWEEN CURRENT ROW AND CURRENT ROW`. Likewise, SUM is performed only over the previous row for the PreviousRow window, because the window size was set to `ROWS BETWEEN 1 PRECEDING AND 1 PRECEDING`. The value of PreviousRow is NULL in the first row since it has no predecessor, so the Delta value is also NULL.

Complex analytics

Consider the following query, which lists the top salespeople (defined by total sales) for each product in the database:

```
SELECT s.ProductID AS Products, o.SalesRepresentative,
     SUM( s.Quantity ) AS total_quantity,
     SUM( s.Quantity * p.UnitPrice ) AS total_sales
  FROM SalesOrders o KEY JOIN SalesOrderItems s
    KEY JOIN Products p
  GROUP BY s.ProductID, o.SalesRepresentative
  HAVING total_sales = (
    SELECT First SUM( s2.Quantity * p2.UnitPrice )
        AS sum_sales
      FROM SalesOrders o2 KEY JOIN
        SalesOrderItems s2 KEY JOIN Products p2
      WHERE s2.ProductID = s.ProductID
      GROUP BY o2.SalesRepresentative
      ORDER BY sum_sales DESC )
  ORDER BY s.ProductID;
```

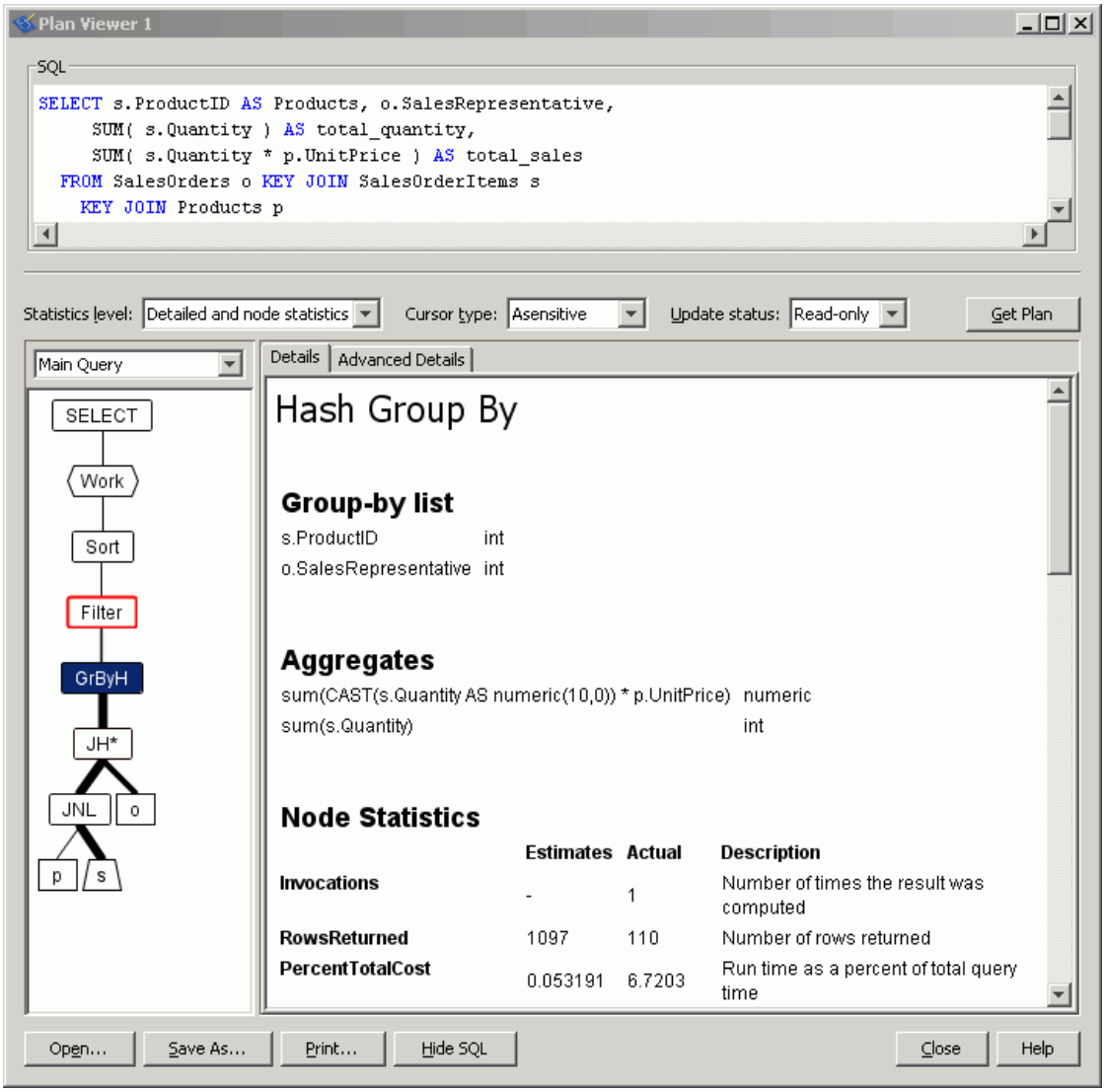



This query returns the result:

	Products	SalesRepresentative	total_quantity	total_sales
1	300	299	660	5940.00
2	301	299	516	7224.00
3	302	299	336	4704.00
4	400	299	458	4122.00

	Products	SalesRepresentative	total_quantity	total_sales
5	401	902	360	3600.00
6	500	949	360	2520.00
7	501	690	360	2520.00
8	501	949	360	2520.00
9	600	299	612	14688.00
10	601	299	636	15264.00
11	700	299	1008	15120.00

The original query is formed using a correlated subquery that determines the highest sales for any particular product, as ProductID is the subquery's correlated outer reference. Using a nested query, however, is often an expensive option, as in this case. This is because the subquery involves not only a GROUP BY clause, but also an ORDER BY clause within the GROUP BY clause. This makes it impossible for the query optimizer to rewrite this nested query as a join while retaining the same semantics. So, during query execution the subquery is evaluated for each derived row computed in the outer block.



Note the expensive Filter predicate in the graphical plan: the optimizer estimates that 99% of the query's execution cost is because of this plan operator. The plan for the subquery clearly illustrates why the filter operator in the main block is so expensive: the subquery involves two nested loops joins, a hashed GROUP BY operation, and a sort.

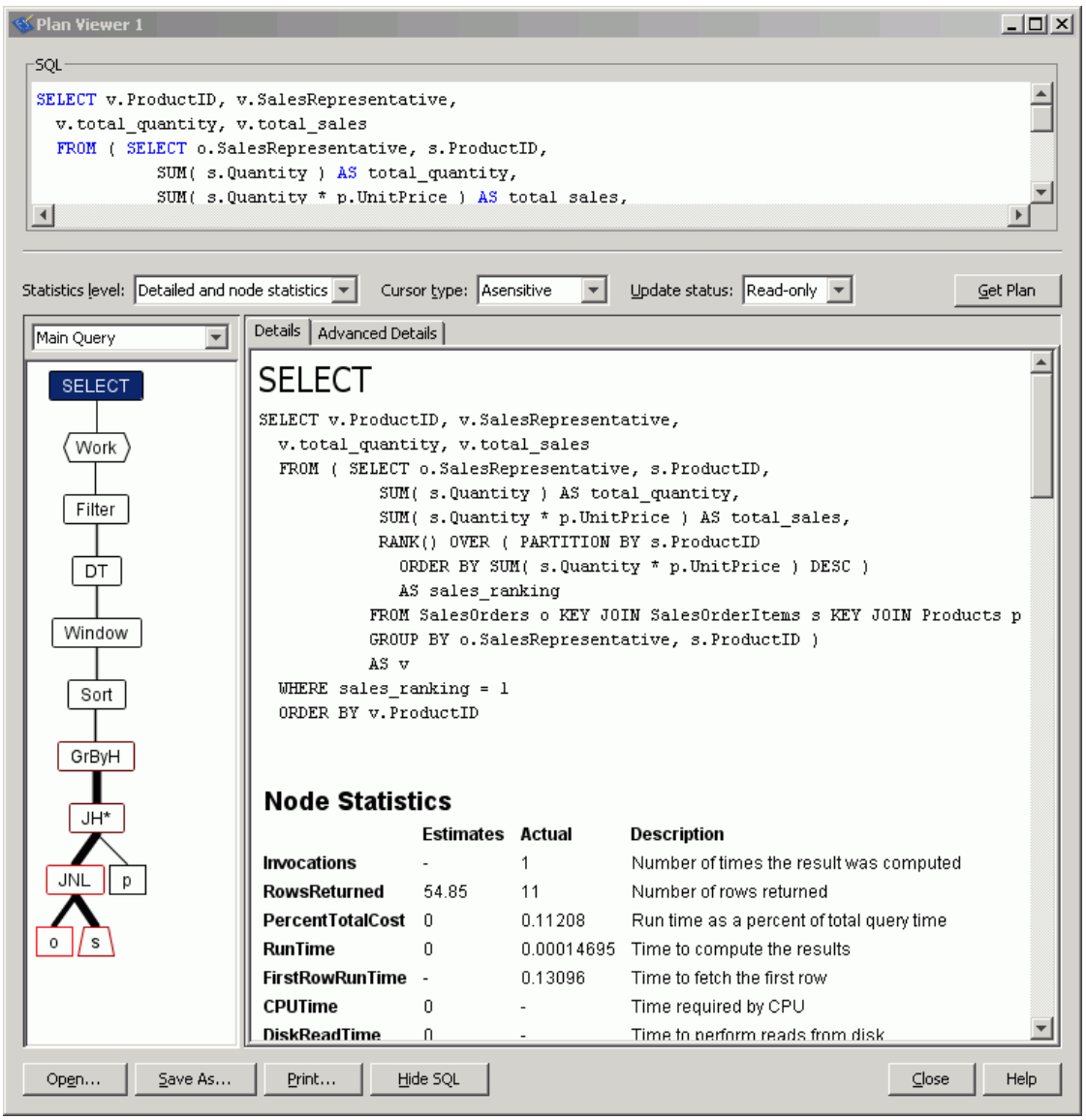
Rewriting using a ranking function

A rewrite of the same query, using a ranking function, computes the identical result much more efficiently:

```
SELECT v.ProductID, v.SalesRepresentative,
  v.total_quantity, v.total_sales
  FROM ( SELECT o.SalesRepresentative, s.ProductID,
           SUM( s.Quantity ) AS total_quantity,
           SUM( s.Quantity * p.UnitPrice ) AS total_sales,
           RANK() OVER ( PARTITION BY s.ProductID
```

```
            ORDER BY SUM( s.Quantity * p.UnitPrice ) DESC )
            AS sales_ranking
         FROM SalesOrders o KEY JOIN SalesOrderItems s KEY JOIN Products p
         GROUP BY o.SalesRepresentative, s.ProductID )
         AS v
WHERE sales_ranking = 1
ORDER BY v.ProductID;
```

This rewritten query results in a simpler plan:



Recall that a window operator is computed after the processing of a GROUP BY clause and before the evaluation of the select list items and the query's ORDER BY clause. As seen in the graphical plan, after the join of the three tables, the joined rows are grouped by the combination of the SalesRepresentative and

ProductID attributes. So, the SUM aggregate functions of total_quantity and total_sales can be computed for each combination of SalesRepresentative and ProductID.

Following the evaluation of the GROUP BY clause, the RANK function is then computed to rank the rows in the intermediate result in descending sequence by total_sales, using a window. Note that the WINDOW specification involves a PARTITION BY clause. By doing so, the result of the GROUP BY clause is repartitioned (or regrouped)—this time by ProductID. So, the RANK function ranks the rows for each product—in descending order of total sales—but for all sales representatives that have sold that product. With this ranking, determining the top salespeople simply requires restricting the derived table's result to reject those rows where the rank is not 1. For ties (rows 7 and 8 in the result set), RANK returns the same value. So, both salespeople 690 and 949 appear in the final result.

See also

- "SUM function [Aggregate]" [*SQL Anywhere Server - SQL Reference*]

AVG function example

In this example, AVG is used as a window function to compute the moving average of all product sales, by month, in the year 2000. Note that the WINDOW specification uses a RANGE clause, which causes the window bounds to be computed based on the month value, and not by the number of adjacent rows as with the ROWS clause. Using ROWS would yield different results if, for example, there were no sales of some or all the products in a particular month.

```
SELECT *
  FROM ( SELECT s.ProductID,
         Month( o.OrderDate ) AS julian_month,
         SUM( s.Quantity ) AS sales,
         AVG( SUM( s.Quantity ) )
         OVER ( PARTITION BY s.ProductID
           ORDER BY Month( o.OrderDate ) ASC
           RANGE BETWEEN 1 PRECEDING AND 1 FOLLOWING )
         AS average_sales
         FROM SalesOrderItems s KEY JOIN SalesOrders o
         WHERE Year( o.OrderDate ) = 2000
         GROUP BY s.ProductID, Month( o.OrderDate ) )
  AS DT
  ORDER BY 1,2;
```

See also

- "AVG function [Aggregate]" [*SQL Anywhere Server - SQL Reference*]

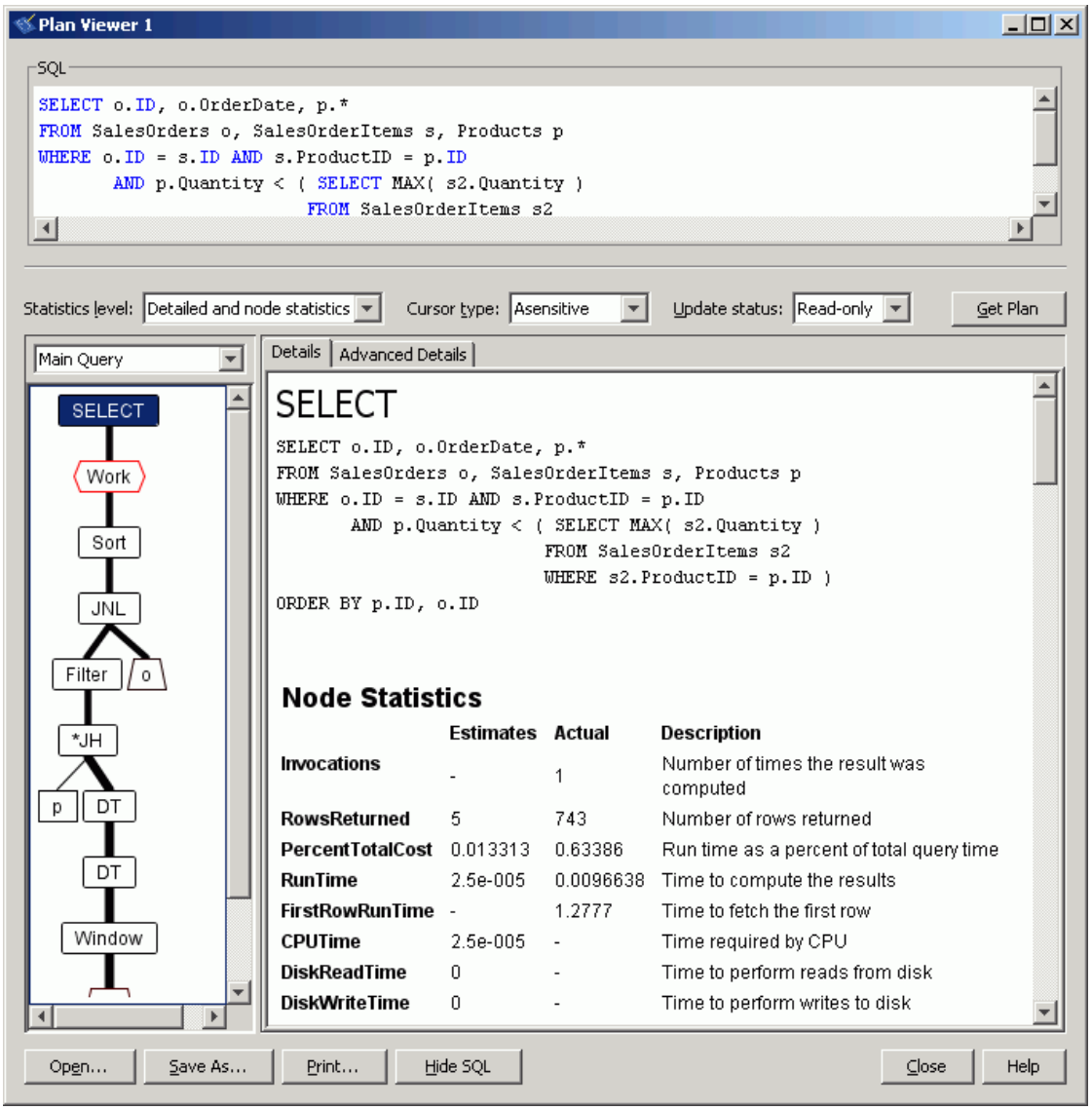
MAX function example

Eliminating correlated subqueries

In some situations, you may need the ability to compare a particular column value with a maximum or minimum value. Often you form these queries as nested queries involving a correlated attribute (also known as an outer reference). As an example, consider the following query, which lists all orders, including product information, where the product quantity-on-hand cannot cover the maximum single order for that product:

```
SELECT o.ID, o.OrderDate, p.*
FROM SalesOrders o, SalesOrderItems s, Products p
WHERE o.ID = s.ID AND s.ProductID = p.ID
       AND p.Quantity < ( SELECT MAX( s2.Quantity )
                          FROM SalesOrderItems s2
                          WHERE s2.ProductID = p.ID )
ORDER BY p.ID, o.ID;
```

The graphical plan for this query is displayed in the Plan Viewer as shown below. Note how the query optimizer has transformed this nested query to a join of the Products and SalesOrders tables with a derived table, denoted by the correlation name DT, which contains a window function.



Rather than relying on the optimizer to transform the correlated subquery into a join with a derived table—which can only be done for straightforward cases due to the complexity of the semantic analysis—you can form such queries using a window function:

```
SELECT order_quantity.ID, o.OrderDate, p.*
  FROM ( SELECT s.ID, s.ProductID,
           MAX( s.Quantity ) OVER (
             PARTITION BY s.ProductID
             ORDER BY s.ProductID )
           AS max_quantity
           FROM SalesOrderItems s )
  AS order_quantity, Products p, SalesOrders o
  WHERE p.ID = ProductID
    AND o.ID = order_quantity.ID
    AND p.Quantity < max_quantity
  ORDER BY p.ID, o.ID;
```

See also

- "MIN function [Aggregate]" [*SQL Anywhere Server - SQL Reference*]
- "MAX function [Aggregate]" [*SQL Anywhere Server - SQL Reference*]

FIRST_VALUE and LAST_VALUE function examples

The FIRST_VALUE and LAST_VALUE functions return values from the first and last rows of a window. This allows a query to access values from multiple rows at once, without the need for a self-join.

These two functions are different from the other window aggregate functions because they must be used with a window. Also, unlike the other window aggregate functions, these functions allow the IGNORE NULLS clause. If IGNORE NULLS is specified, the first or last non-NULL value of the desired expression is returned. Otherwise, the first or last value is returned.

Example 1: First entry in a group

The FIRST_VALUE function can be used to retrieve the first entry in an ordered group of values. The following query returns, for each order, the product identifier of the order's first item; that is, the ProductID of the item with the smallest LineID for each order.

Notice that the query uses the DISTINCT keyword to remove duplicates; without it, duplicate rows are returned for each item in each order.

```
SELECT DISTINCT ID,
FIRST_VALUE( ProductID ) OVER ( PARTITION BY ID ORDER BY LineID )
FROM SalesOrderItems
ORDER BY ID;
```

Example 2: Percentage of highest sales

A common use of the FIRST_VALUE function is to compare a value in each row with the maximum or minimum value within the current group. The following query computes the total sales for each sales representative, and then compares that representative's total sales with the maximum total sales for the same product. The result is expressed as a percentage of the maximum total sales.

```
SELECT s.ProductID AS prod_id, o.SalesRepresentative AS sales_rep,
    SUM( s.Quantity * p.UnitPrice ) AS total_sales,
```

```
  100 * total_sales / ( FIRST_VALUE( SUM( s.Quantity * p.UnitPrice ) )
                        OVER Sales_Window ) AS total_sales_percentage
FROM SalesOrders o KEY JOIN SalesOrderItems s KEY JOIN Products p
GROUP BY o.SalesRepresentative, s.ProductID
  WINDOW Sales_Window AS ( PARTITION BY s.ProductID
                           ORDER BY SUM( s.Quantity * p.UnitPrice ) DESC )
ORDER BY s.ProductID;
```

Example 3: Populating NULL values making data more dense

The FIRST_VALUE and LAST_VALUE functions are useful when you have made your data more dense and you need to populate values instead of having NULLs. For example, suppose the sales representative with the highest total sales each day wins the distinction of Representative of the Day. The following query lists the winning sales representatives for the first week of April, 2001:

```
SELECT v.OrderDate, v.SalesRepresentative AS rep_of_the_day
FROM ( SELECT o.SalesRepresentative, o.OrderDate,
              RANK() OVER ( PARTITION BY o.OrderDate
                            ORDER BY SUM( s.Quantity *
                              p.UnitPrice ) DESC ) AS sales_ranking
       FROM SalesOrders o KEY JOIN SalesOrderItems s KEY JOIN Products p
       GROUP BY o.SalesRepresentative, o.OrderDate ) AS v
WHERE v.sales_ranking = 1
AND v.OrderDate BETWEEN '2001-04-01' AND '2001-04-07'
ORDER BY v.OrderDate;
```

The query returns the following results:

OrderDate	rep_of_the_day
2001-04-01	949
2001-04-02	856
2001-04-05	902
2001-04-06	467
2001-04-07	299

However, note that no results are returned for days in which no sales were made. The following query makes the data more dense, creating records for days in which no sales were made. Additionally, it uses the LAST_VALUE function to populate the NULL values for rep_of_the_day (on non-winning days) with the ID of the last winning representative, until a new winner occurs in the results.

```
SELECT d.dense_order_date,
                   LAST_VALUE( v.SalesRepresentative IGNORE NULLS )
                       OVER ( ORDER BY d.dense_order_date )
                       AS rep_of_the_day
FROM ( SELECT o.SalesRepresentative, o.OrderDate,
              RANK() OVER ( PARTITION BY o.OrderDate
                            ORDER BY SUM( s.Quantity *
                             p.UnitPrice ) DESC ) AS sales_ranking
       FROM SalesOrders o KEY JOIN SalesOrderItems s KEY JOIN Products p
       GROUP BY o.SalesRepresentative, o.OrderDate ) AS v
RIGHT OUTER JOIN ( SELECT DATEADD( day, row_num, '2001-04-01' )
                          AS dense_order_date
```



```
                    FROM sa_rowgenerator( 0, 6 )) AS d
ON v.OrderDate = d.dense_order_date AND sales_ranking = 1
ORDER BY d.dense_order_date;
```

The query returns the following results:

OrderDate	rep_of_the_day
2001-04-01	949
2001-04-02	856
2001-04-03	856
2001-04-04	856
2001-04-05	902
2001-04-06	467
2001-04-07	299

The derived table v from the previous query is joined to a derived table d, which contains all the dates under consideration. This yields a row for each desired day, but this outer join contains NULL in the SalesRepresentative column for dates on which no sales were made. Using the LAST_VALUE function solves this problem by defining rep_of_the_day for a given row to be the last non-NULL value of SalesRepresentative leading up to the corresponding day.

See also

- "FIRST_VALUE function [Aggregate]" [*SQL Anywhere Server - SQL Reference*]
- "LAST_VALUE function [Aggregate]" [*SQL Anywhere Server - SQL Reference*]
- "Window functions" on page 456

Standard deviation and variance functions

SQL Anywhere supports two versions of variance and standard deviation functions: a sampling version, and a population version. Choosing between the two versions depends on the statistical context in which the function is to be used.

All the variance and standard deviation functions are true aggregate functions in that they can compute values for a partition of rows as determined by the query's GROUP BY clause. As with other basic aggregate functions such as MAX or MIN, their computation also ignores NULL values in the input.

For improved performance, SQL Anywhere calculates the mean, and the deviation from mean, in one step. This means that only one pass over the data is required.

Also, regardless of the domain of the expression being analyzed, all variance and standard deviation computation is done using IEEE double-precision floating-point arithmetic. If the input to any variance or

standard deviation function is the empty set, then each function returns NULL as its result. If VAR_SAMP is computed for a single row, then it returns NULL, while VAR_POP returns the value 0.

Following are the standard deviation and variance functions offered in SQL Anywhere:

- STDDEV function
- STDDEV_POP function
- STDDEV_SAMP function
- VARIANCE function
- VAR_POP function
- VAR_SAMP function

To review the mathematical formulas represented by these functions see "Mathematical formulas for the aggregate functions" on page 490.

STDDEV function

This function is an alias for the STDDEV_SAMP function. See "STDDEV_SAMP function [Aggregate]" [*SQL Anywhere Server - SQL Reference*].

STDDEV_POP function

This function computes the standard deviation of a population consisting of a numeric expression, as a DOUBLE.

Example 1

The following query returns a result set that shows the employees whose salary is one standard deviation greater than the average salary of their department. Standard deviation is a measure of how much the data varies from the mean.

```
SELECT *
FROM ( SELECT
    Surname AS Employee,
    DepartmentID AS Department,
    CAST( Salary as DECIMAL( 10, 2 ) )
        AS Salary,
    CAST( AVG( Salary )
        OVER ( PARTITION BY DepartmentID ) AS DECIMAL ( 10, 2 ) )
        AS Average,
    CAST( STDDEV_POP( Salary )
        OVER ( PARTITION BY DepartmentID ) AS DECIMAL ( 10, 2 ) )
        AS StandardDeviation
    FROM Employees
    GROUP BY Department, Employee, Salary )
    AS DerivedTable
WHERE Salary > Average + StandardDeviation
ORDER BY Department, Salary, Employee;
```

The table that follows represents the result set from the query. Every department has at least one employee whose salary significantly deviates from the mean.

	Employee	Department	Salary	Average	StandardDeviation
1	Lull	100	87900.00	58736.28	16829.60

	Employee	Department	Salary	Average	StandardDeviation
2	Scheffield	100	87900.00	58736.28	16829.60
3	Scott	100	96300.00	58736.28	16829.60
4	Sterling	200	64900.00	48390.95	13869.60
5	Savarino	200	72300.00	48390.95	13869.60
6	Kelly	200	87500.00	48390.95	13869.60
7	Shea	300	138948.00	59500.00	30752.40
8	Blaikie	400	54900.00	43640.67	11194.02
9	Morris	400	61300.00	43640.67	11194.02
10	Evans	400	68940.00	43640.67	11194.02
11	Martinez	500	55500.00	33752.20	9084.50

Employee Scott earns $96,300.00, while the departmental average is $58,736.28. The standard deviation for that department is $16,829.00, which means that salaries less than $75,565.88 (`58736.28 + 16829.60 = 75565.88`) fall within one standard deviation of the mean. At $96,300.00, employee Scott is well above that figure.

This example assumes that Surname and Salary are unique for each employee, which isn't necessarily true. To ensure uniqueness, you could add EmployeeID to the GROUP BY clause.

Example 2

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;
```

This query returns the following result:

Year	Quarter	Average	Variance
2000	1	25.775148	14.2794...
2000	2	27.050847	15.0270...

Year	Quarter	Average	Variance
...	...	...	...

For more information about the syntax for this function, see "STDDEV_SAMP function [Aggregate]" [*SQL Anywhere Server - SQL Reference*].

STDDEV_SAMP function

This function computes the standard deviation of a sample consisting of a numeric expression, as a DOUBLE. For example, the following statement returns the average and variance in the number of items per order in different quarters:

```
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;
```

This query returns the following result:

Year	Quarter	Average	Variance
2000	1	25.775148	14.3218...
2000	2	27.050847	15.0696...
...	...	...	...

For more information about the syntax for this function, see "STDDEV_POP function [Aggregate]" [*SQL Anywhere Server - SQL Reference*].

VARIANCE function

This function is an alias for the VAR_SAMP function. See "VAR_SAMP function [Aggregate]" [*SQL Anywhere Server - SQL Reference*].

VAR_POP function

This function computes the statistical variance of a population consisting of a numeric expression, as a DOUBLE. For 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;
```

This query returns the following result:

Year	Quarter	Average	Variance
2000	1	25.775148	203.9021...
2000	2	27.050847	225.8109...
...	...	...	...

If VAR_POP is computed for a single row, then it returns the value 0.

For more information about the syntax for this function, see "VAR_POP function [Aggregate]" [*SQL Anywhere Server - SQL Reference*].

VAR_SAMP function

This function computes the statistical variance of a sample consisting of a numeric expression, as a DOUBLE.

For 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;
```

This query returns the following result:

Year	Quarter	Average	Variance
2000	1	25.775148	205.1158...
2000	2	27.050847	227.0939...
...	...	...	...

If VAR_SAMP is computed for a single row, then it returns NULL.

For more information about the syntax for this function, see "VAR_SAMP function [Aggregate]" [*SQL Anywhere Server - SQL Reference*].

Correlation and linear regression functions

SQL Anywhere supports a variety of statistical functions, the results of which can be used to assist in analyzing the quality of a linear regression.

For more information about the mathematical formulas represented by these functions see "Mathematical formulas for the aggregate functions" on page 490.

The first argument of each function is the dependent expression (designated by Y), and the second argument is the independent expression (designated by X).

- **COVAR_SAMP function** The COVAR_SAMP function returns the sample covariance of a set of (Y, X) pairs.

  For more information about the syntax for this function, see "COVAR_SAMP function [Aggregate]" [*SQL Anywhere Server - SQL Reference*].

- **COVAR_POP function** The COVAR_POP function returns the population covariance of a set of (Y, X) pairs.

  For more information about the syntax for this function, see "COVAR_POP function [Aggregate]" [*SQL Anywhere Server - SQL Reference*].

- **CORR function** The CORR function returns the correlation coefficient of a set of (Y, X) pairs.

  For more information about the syntax for this function, see "CORR function [Aggregate]" [*SQL Anywhere Server - SQL Reference*].

- **REGR_AVGX function** The REGR_AVGX function returns the mean of the x-values from all the non-NULL pairs of (Y, X) values.

  For more information about the syntax for this function, see "REGR_AVGX function [Aggregate]" [*SQL Anywhere Server - SQL Reference*].

- **REGR_AVGY function** The REGR_AVGY function returns the mean of the y-values from all the non-NULL pairs of (Y, X) values.

  For more information about the syntax for this function, see "REGR_AVGY function [Aggregate]" [*SQL Anywhere Server - SQL Reference*].

- **REGR_SLOPE function** The REGR_SLOPE function computes the slope of the linear regression line fitted to non-NULL pairs.

  For more information about the syntax for this function, see "REGR_SLOPE function [Aggregate]" [*SQL Anywhere Server - SQL Reference*].

- **REGR_INTERCEPT function** The REGR_INTERCEPT function computes the y-intercept of the linear regression line that best fits the dependent and independent variables.

  For more information about the syntax for this function, see "REGR_INTERCEPT function [Aggregate]" [*SQL Anywhere Server - SQL Reference*].

- **REGR_R2 function** The REGR_R2 function computes the coefficient of determination (also referred to as **R-squared** or the **goodness of fit** statistic) for the regression line.

  For more information about the syntax for this function, see "REGR_R2 function [Aggregate]" [*SQL Anywhere Server - SQL Reference*].

- **REGR_COUNT function** The REGR_COUNT function returns the number of non-NULL pairs of (Y, X) values in the input. Only if both X and Y in a given pair are non-NULL is that observation be used in any linear regression computation.

  For more information about the syntax for this function, see "REGR_COUNT function [Aggregate]" [*SQL Anywhere Server - SQL Reference*].

- **REGR_SXX function** The function returns the sum of squares of x-values of the (Y, X) pairs.

  The equation for this function is equivalent to the numerator of the sample or population variance formulae. Note, as with the other linear regression functions, that REGR_SXX ignores any pair of (Y, X) values in the input where either X or Y is NULL.

  For more information about the syntax for this function, see "REGR_SXX function [Aggregate]" [*SQL Anywhere Server - SQL Reference*].

- **REGR_SYY function** The function returns the sum of squares of y-values of the (Y, X) pairs.

  For more information about the syntax for this function, see "REGR_SYY function [Aggregate]" [*SQL Anywhere Server - SQL Reference*].

- **REGR_SXY function** The function returns the difference of two sum of products over the set of (Y, X) pairs.

  For more information about the syntax for this function, see "REGR_SXY function [Aggregate]" [*SQL Anywhere Server - SQL Reference*].

Window ranking functions

Window ranking functions return the rank of a row relative to the other rows in a partition. The ranking functions supported by SQL Anywhere are:

- CUME_DIST
- DENSE_RANK
- PERCENT_RANK
- RANK

Ranking functions are not considered aggregate functions because they do not compute a result from multiple input rows in the same manner as, for example, the SUM aggregate function. Rather, each of these functions computes the rank, or relative ordering, of a row within a partition based on the value of a particular expression. Each set of rows within a partition is ranked independently; if the OVER clause does not contain a PARTITION BY clause, the entire input is treated as a single partition. So, you cannot specify a ROWS or RANGE clause for a window used by a ranking function. It is possible to form a query containing multiple ranking functions, each of which partition or sort the input rows differently.

All ranking functions require an ORDER BY clause to specify the sort order of the input rows upon which the ranking functions depend. If the ORDER BY clause includes multiple expressions, the second and subsequent expressions are used to break ties if the first expression has the same value in adjacent rows. NULL values in SQL Anywhere are always sorted before any other value (in ascending sequence).

RANK function

You use the RANK function to return the rank of the value in the current row as compared to the value in other rows. The rank of a value reflects the order in which it would appear if the list of values was sorted.

When using the RANK function, the rank is calculated for the expression specified in the window's ORDER BY clause. If the ORDER BY clause includes multiple expressions, the second and subsequent expressions are used to break ties if the first expression has the same value in adjacent rows. NULL values are sorted before any other value (in ascending sequence).

Example 1

The following query determines the three most expensive products in the database. A descending sort sequence is specified for the window so that the most expensive products have the lowest rank, that is, rankings start at 1.

```
SELECT Top 3 *
       FROM ( SELECT Description, Quantity, UnitPrice,
              RANK() OVER ( ORDER BY UnitPrice DESC ) AS Rank
              FROM Products ) AS DT
ORDER BY Rank;
```

This query returns the following result:

	Description	Quantity	UnitPrice	Rank
1	Zipped Sweatshirt	32	24.00	1
2	Hooded Sweatshirt	39	24.00	1
3	Cotton Shorts	80	15.00	3

Note that rows 1 and 2 have the same value for Unit Price, and therefore also have the same rank. This is called a tie.

With the RANK function, the rank value jumps after a tie. For example, the rank value for row 3 has jumped to three instead of 2. This is different from the DENSE_RANK function, where no jumping occurs after a tie. See “DENSE_RANK function” on page 484.

Example 2

The following SQL query finds the male and female employees from Utah and ranks them in descending order according to salary.

```
SELECT Surname, Salary, Sex,
     RANK() OVER ( ORDER BY Salary DESC ) "Rank"
     FROM Employees
WHERE State IN ( 'UT' );
```

The table that follows represents the result set from the query:

	Surname	Salary	Sex	Rank
1	Shishov	72995.00	F	1
2	Wang	68400.00	M	2
3	Cobb	62000.00	M	3
4	Morris	61300.00	M	4
5	Diaz	54900.00	M	5
6	Driscoll	48023.69	M	6
7	Hildebrand	45829.00	F	7
8	Goggin	37900.00	M	8
9	Rebeiro	34576.00	M	9
10	Bigelow	31200.00	F	10
11	Lynch	24903.00	M	11

Example 3

You can partition your data to provide different results. Using the query from Example 2, you can change the data by partitioning it by gender. The following example ranks employees in descending order by salary and partitions by gender.

```
SELECT Surname, Salary, Sex,
     RANK ( ) OVER ( PARTITION BY Sex
     ORDER BY Salary DESC ) "Rank"
     FROM Employees
WHERE State IN ( 'UT' );
```

The table that follows represents the result set from the query:

	Surname	Salary	Sex	Rank
1	Wang	68400.00	M	1
2	Cobb	62000.00	M	2
3	Morris	61300.00	M	3
4	Diaz	54900.00	M	4
5	Driscoll	48023.69	M	5
6	Goggin	37900.00	M	6

	Surname	Salary	Sex	Rank
7	Rebeiro	34576.00	M	7
8	Lynch	24903.00	M	8
9	Shishov	72995.00	F	1
10	Hildebrand	45829.00	F	2
11	Bigelow	31200.00	F	3

For more information about the syntax for the RANK function, see "RANK function [Ranking]" [*SQL Anywhere Server - SQL Reference*].

DENSE_RANK function

Similar to the RANK function, you use the DENSE_RANK function to return the rank of the value in the current row as compared to the value in other rows. The rank of a value reflects the order in which it would appear if the list of values were sorted. Rank is calculated for the expression specified in the window's ORDER BY clause.

The DENSE_RANK function returns a series of ranks that are monotonically increasing with no gaps, or jumps in rank value. The term dense is used because there are no jumps in rank value (unlike the RANK function).

As the window moves down the input rows, the rank is calculated for the expression specified in the window's ORDER BY clause. If the ORDER BY clause includes multiple expressions, the second and subsequent expressions are used to break ties if the first expression has the same value in adjacent rows. NULL values are sorted before any other value (in ascending sequence).

Example 1

The following query determines the three most expensive products in the database. A descending sort sequence is specified for the window so that the most expensive products have the lowest rank (rankings start at 1).

```
SELECT Top 3 *
  FROM ( SELECT Description, Quantity, UnitPrice,
    DENSE_RANK( ) OVER ( ORDER BY UnitPrice DESC ) AS Rank
    FROM Products ) AS DT
  ORDER BY Rank;
```

This query returns the following result:

	Description	Quantity	UnitPrice	Rank
1	Hooded Sweatshirt	39	24.00	1

	Description	Quantity	UnitPrice	Rank
2	Zipped Sweatshirt	32	24.00	1
3	Cotton Shorts	80	15.00	2

Note that rows 1 and 2 have the same value for Unit Price, and therefore also have the same rank. This is called a tie.

With the DENSE_RANK function, there is no jump in the rank value after a tie. For example, the rank value for row 3 is 2. This is different from the RANK function, where a jump in rank values occurs after a tie. See "RANK function" on page 482.

Example 2

Because windows are evaluated after a query's GROUP BY clause, you can specify complex requests that determine rankings based on the value of an aggregate function.

The following query produces the top three salespeople in each region by their total sales within that region, along with the total sales for each region:

```
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 Products 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;
```

This query returns the following result:

	SalesRepresentative	Region	total_sales	sales_rank
1	299	Canada	9312.00	1
2	(NULL)	Canada	24768.00	1
3	1596	Canada	3564.00	2
4	856	Canada	2724.00	3
5	299	Central	32592.00	1
6	(NULL)	Central	134568.00	1
7	856	Central	14652.00	2

	SalesRepresentative	Region	total_sales	sales_rank
8	467	Central	14352.00	3
9	299	Eastern	21678.00	1
10	(NULL)	Eastern	142038.00	1
11	902	Eastern	15096.00	2
12	690	Eastern	14808.00	3
13	1142	South	6912.00	1
14	(NULL)	South	45262.00	1
15	667	South	6480.00	2
16	949	South	5782.00	3
17	299	Western	5640.00	1
18	(NULL)	Western	37632.00	1
19	1596	Western	5076.00	2
20	667	Western	4068.00	3

This query combines multiple groupings through the use of GROUPING SETS. So, the WINDOW PARTITION clause for the window uses the GROUPING function to distinguish between detail rows that represent particular salespeople and the subtotal rows that list the total sales for an entire region. The subtotal rows by region, which have the value NULL for the sales rep attribute, each have the ranking value of 1 because the result's ranking order is restarted with each partition of the input; this ensures that the detail rows are ranked correctly starting at 1.

Finally, note in this example that the DENSE_RANK function ranks the input over the aggregation of the total sales. An aliased select list item is used as a shorthand in the WINDOW ORDER clause.

For more information about the syntax for the DENSE_RANK function, see “DENSE_RANK function [Ranking]” [*SQL Anywhere Server - SQL Reference*].

CUME_DIST function

The cumulative distribution function, CUME_DIST, is sometimes defined as the inverse of percentile. CUME_DIST computes the normalized position of a specific value relative to the set of values in the window. The range of the function is between 0 and 1.

As the window moves down the input rows, the cumulative distribution is calculated for the expression specified in the window's ORDER BY clause. If the ORDER BY clause includes multiple expressions,

the second and subsequent expressions are used to break ties if the first expression has the same value in adjacent rows. NULL values are sorted before any other value (in ascending sequence).

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' );
```

This query returns the following result:

DepartmentID	Surname	Salary	Rank
200	Savarino	72300.00	0.333333333333333
200	Clark	45000.00	0.666666666666667
200	Overbey	39300.00	1

For more information about the syntax for the CUME_DIST function, see "CUME_DIST function [Ranking]" [*SQL Anywhere Server - SQL Reference*].

PERCENT_RANK function

Similar to the PERCENT function, the PERCENT_RANK function returns the rank for the value in the column specified in the window's ORDER BY clause, but expressed as a fraction between 0 an 1, calculated as (RANK - 1)/(- 1).

As the window moves down the input rows, the rank is calculated for the expression specified in the window's ORDER BY clause. If the ORDER BY clause includes multiple expressions, the second and subsequent expressions are used to break ties if the first expression has the same value in adjacent rows. NULL values are sorted before any other value (in ascending sequence).

Example 1

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 using a decimal percentage, and are partitioned by gender.

```
SELECT DepartmentID, Surname, Salary, Sex,
    PERCENT_RANK( ) OVER ( PARTITION BY Sex
      ORDER BY Salary DESC ) AS PctRank
  FROM Employees
  WHERE State IN ( 'NY' );
```

This query returns the following results:

	DepartmentID	Surname	Salary	Sex	PctRank
1	200	Martel	55700.000	M	0.0
2	100	Guevara	42998.000	M	0.333333333
3	100	Soo	39075.000	M	0.666666667
4	400	Ahmed	34992.000	M	1.0
5	300	Davidson	57090.000	F	0.0
6	400	Blaikie	54900.000	F	0.333333333
7	100	Whitney	45700.000	F	0.666666667
8	400	Wetherby	35745.000	F	1.0

Since the input is partitioned by gender (Sex), PERCENT_RANK is evaluated separately for males and females.

Example 2

The following example returns a list of female employees in Utah and Arizona and ranks them in descending order according to salary. Here, the PERCENT_RANK function is used to provide a cumulative total in descending order.

```
SELECT Surname, Salary,
     PERCENT_RANK ( ) OVER ( ORDER BY Salary DESC ) "Rank"
     FROM Employees
WHERE State IN ( 'UT', 'AZ' ) AND Sex IN ( 'F' );
```

This query returns the following results:

	Surname	Salary	Rank
1	Shishov	72995.00	0
2	Jordan	51432.00	0.25
3	Hildebrand	45829.00	0.5
4	Bigelow	31200.00	0.75
5	Bertrand	29800.00	1

Using PERCENT_RANK to find top and bottom percentiles

You can use PERCENT_RANK to find the top or bottom percentiles in the data set. In the following example, the query returns male employees whose salary is in the top five percent of the data set.


```
SELECT *
FROM ( SELECT Surname, Salary,
       PERCENT_RANK ( ) OVER ( ORDER BY Salary DESC ) "Rank"
       FROM Employees
       WHERE Sex IN ( 'M' )  )
       AS DerivedTable ( Surname, Salary, Percent )
WHERE Percent < 0.05;
```

This query returns the following results:

	Surname	Salary	Percent
1	Scott	96300.00	0
2	Sheffield	87900.00	0.025
3	Lull	87900.00	0.025

For more information about the syntax for the PERCENT_RANK function, see "PERCENT_RANK function [Ranking]" [*SQL Anywhere Server - SQL Reference*].

Row numbering functions

Row numbering functions uniquely number the rows in a partition. SQL Anywhere supports two row numbering functions; NUMBER and ROW_NUMBER. It is recommended that you use the ROW_NUMBER function because it is an ANSI standard-compliant function that provides much of the same functionality as the SQL Anywhere NUMBER(*) function. While both functions perform similar tasks, there are several limitations to the NUMBER function that do not exist for the ROW_NUMBER function.

See also

- "NUMBER function [Miscellaneous]" [*SQL Anywhere Server - SQL Reference*]
- "ROW_NUMBER function" on page 489

ROW_NUMBER function

The ROW_NUMBER function uniquely numbers the rows in its result. It is not a ranking function; however, you can use it in any situation in which you can use a ranking function, and it behaves similarly to a ranking function.

For example, you can use ROW_NUMBER in a derived table so that additional restrictions, even joins, can be made over the ROW_NUMBER values:

```
SELECT *
FROM ( SELECT Description, Quantity,
       ROW_NUMBER( ) OVER ( ORDER BY ID ASC ) AS RowNum
FROM Products ) AS DT
WHERE RowNum <= 3
ORDER BY RowNum;
```

This query returns the following results:

Description	Quantity	RowNum
Tank Top	28	1
V-neck	54	2
Crew Neck	75	3

As with the ranking functions, ROW_NUMBER requires an ORDER BY clause.

As well, ROW_NUMBER can return non-deterministic results when the window's ORDER BY clause is over non-unique expressions; row order is unpredictable for ties.

ROW_NUMBER is designed to work over the entire partition, so a ROWS or RANGE clause cannot be specified with a ROW_NUMBER function.

For more information about the syntax for the ROW_NUMBER function, see "ROW_NUMBER function [Miscellaneous]" [*SQL Anywhere Server - SQL Reference*].

Mathematical formulas for the aggregate functions

For information purposes, the following two tables provide the equivalent mathematical formulas for all the window aggregate functions supported in SQL Anywhere.

Simple aggregate functions

Function	Symbol	Formula
SUM(X)		$\sum_{i=1}^{n} x_i$
MAX(X)		$x_i : x_i \geq x_j, i \neq j \; \forall \, i,j \in n$
MIN(X)		$x_i : x_i \leq x_j, i \neq j \; \forall \, i,j \in n$
AVG(X)	$\bar{x}$	$\frac{\sum x_i}{n}$
COUNT(*)		n
VAR_SAMP(X)	s_x^2	$\frac{\sum (x_i - \bar{x})^2}{(n-1)}$
VAR_POP(X)	σ_x^2	$\frac{\sum (x_i - \bar{x})^2}{n}$
VARIANCE(X)		identical to VAR_SAMP(X)
STDDEV_SAMP(X)	s_x	$\sqrt{\frac{\sum (x_i - \bar{x})^2}{(n-1)}}$
STDDEV_POP(X)	σ_x	$\sqrt{\frac{\sum (x_i - \bar{x})^2}{n}}$
STDDEV(X)		identical to STDDEV_SAMP(X)

Statistical aggregate functions

COVAR_SAMP(Y,X)	*Co-variance*	$s_{xy} = \frac{\sum xy - \frac{(\sum x)(\sum y)}{n}}{(n-1)}$
COVAR_POP(Y,X)	*Co-variance*	$\sigma_{xy} = \frac{\sum xy - \frac{(\sum x)(\sum y)}{n}}{n}$
CORR(Y,X)	*Correlation Coefficient*	$r = \frac{\sum xy - \frac{1}{n}(\sum x)(\sum y)}{(n-1)s_x s_y}$
REGR_AVGX(Y,X)	*Independent mean*	$\bar{x}$
REGR_AVGY(Y,X)	*Dependent mean*	$\bar{y}$
REGR_SLOPE(Y,X)	*Regression Slope*	$b = r\frac{s_y}{s_x}$
REGR_INTERCEPT(Y,X)	*Regression Intercept*	$a = \bar{y} - b\bar{x}$
REGR_R2(Y,X)	*'Goodness-of-fit'*	r^2
REGR_COUNT(Y,X)	*Sample size*	n (non-null (Y, X) pairs)
REGR_SXX(Y,X)	*Sum of squares* (x)	$\sum x^2 - \frac{(\sum x)^2}{n}$
REGR_SYY(Y,X)	*Sum of squares* (y)	$\sum y^2 - \frac{(\sum y)^2}{n}$
REGR_SXY(Y,X)	*Sum of products*	$\sum xy - \frac{(\sum y)(\sum x)}{n}$

Using subqueries

With a relational database, you can store related data in more than one table. In addition to being able to extract data from related tables using a join, you can also extract it using a **subquery**. A subquery is a SELECT statement nested within the SELECT, WHERE, or HAVING clause of a parent SQL statement.

Subqueries make some queries easier to write than joins, and there are queries that cannot be written without using subqueries.

Subqueries can be categorized in different ways:

- whether they can return one or more rows (single-row vs. multiple-row subqueries)
- whether they are correlated or uncorrelated
- whether they are nested within another subquery

Single-row and multiple-row subqueries

Subqueries that can return only one or zero rows to the outer statement are called **single-row subqueries**. Single-row subqueries are subqueries used with a comparison operator in a WHERE, or HAVING clause.

Subqueries that can return more than one row (but only one column) to the outer statement are called **multiple-row subqueries**. Multiple-row subqueries are subqueries used with an IN, ANY, or ALL clause.

Example 1: Single-row subquery

You store information particular to products in one table, Products, and information that pertains to sales orders in another table, SalesOrdersItems. The Products table contains the information about the various products. The SalesOrdersItems table contains information about customers' orders. If a company reorders products when there are fewer than 50 of them in stock, then it is possible to answer the question "Which products are nearly out of stock?" with this query:

```
SELECT ID, Name, Description, Quantity
FROM Products
WHERE Quantity < 50;
```

However, a more helpful result would take into consideration how frequently a product is ordered, since having few of a product that is frequently purchased is more of a concern than having few product that is rarely ordered.

You can use a subquery to determine the average number of items that a customer orders, and then use that average in the main query to find products that are nearly out of stock. The following query finds the names and descriptions of the products which number less than twice the average number of items of each type that a customer orders.

```
SELECT Name, Description
FROM Products WHERE Quantity <  2 * (
   SELECT AVG( Quantity )
   FROM SalesOrderItems
   );
```

In the WHERE clause, subqueries help select the rows from the tables listed in the FROM clause that appear in the query results. In the HAVING clause, they help select the row groups, as specified by the main query's GROUP BY clause, that appear in the query results.

Example 2: Single-row subquery

The following example of a single-row subquery calculates the average price of the products in the Products table. The average is then passed to the WHERE clause of the outer query. The outer query returns the ID, Name, and UnitPrice of all products that are less expensive than the average:

```
SELECT ID, Name, UnitPrice
FROM Products
WHERE UnitPrice <
  ( SELECT AVG( UnitPrice ) FROM Products )
ORDER BY UnitPrice DESC;
```

ID	Name	UnitPrice
401	Baseball Cap	10.00
300	Tee Shirt	9.00
400	Baseball Cap	9.00

ID	Name	UnitPrice
500	Visor	7.00
501	Visor	7.00

Example 3: Simple multiple-row subquery using IN

Suppose you want to identify items that are low in stock, while also identifying orders for those items. You could execute a SELECT statement containing a subquery in the WHERE clause, similar to the following:

```
SELECT *
FROM SalesOrderItems
WHERE ProductID IN
   (  SELECT ID
       FROM Products
       WHERE Quantity < 20 )
ORDER BY ShipDate DESC;
```

In this example, the subquery makes a list of all values in the ID column in the Products table, satisfying the WHERE clause search condition. The subquery then returns a set of rows, but only a single column. The IN keyword treats each value as a member of a set and tests whether each row in the main query is a member of the set.

Example 4: Multiple-row subqueries comparing use of IN, ANY, and ALL

Two tables in the SQL Anywhere sample database contain financial results data. The FinancialCodes table is a table holding the different codes for financial data and their meaning. To list the revenue items from the FinancialData table, execute the following query:

```
SELECT *
FROM FinancialData
WHERE Code IN
    ( SELECT Code
        FROM FinancialCodes
        WHERE type = 'revenue' );
```

Year	Quarter	Code	Amount
1999	Q1	r1	1023
1999	Q2	r1	2033
1999	Q3	r1	2998
1999	Q4	r1	3014
2000	Q1	r1	3114
...	...	...	...

The ANY and ALL keywords can be used in a similar manner. For example, the following query returns the same results as the previous query, but uses the ANY keyword:

```
SELECT *
FROM FinancialData
WHERE FinancialData.Code = ANY
   (  SELECT FinancialCodes.Code
       FROM FinancialCodes
       WHERE type = 'revenue' );
```

While the =ANY condition is identical to the IN condition, ANY can also be used with inequalities such as < or > to give more flexible use of subqueries.

The ALL keyword is similar to the word ANY. For example, the following query lists financial data that is not revenue:

```
SELECT *
FROM FinancialData
WHERE FinancialData.Code <> ALL
   (  SELECT FinancialCodes.Code
       FROM FinancialCodes
      WHERE type = 'revenue' );
```

This is equivalent to the following command using NOT IN:

```
SELECT *
FROM FinancialData
WHERE FinancialData.Code NOT IN
   (  SELECT FinancialCodes.Code
      FROM FinancialCodes
      WHERE type = 'revenue' );
```

Correlated and uncorrelated subqueries

A subquery can contain a reference to an object defined in a parent statement. This is called an **outer reference**. A subquery that contains an outer reference is called a **correlated subquery**. Correlated subqueries cannot be evaluated independently of the outer query because the subquery uses the values of the parent statement. That is, the subquery is performed for each row in the parent statement. So, results of the subquery are dependent upon the active row being evaluated in the parent statement.

For example, the subquery in the statement below returns a value dependent upon the active row in the Products table:

```
SELECT Name, Description
FROM Products
WHERE Quantity < 2 * (
   SELECT AVG( Quantity )
   FROM SalesOrderItems
   WHERE Products.ID=SalesOrderItems.ProductID );
```

In this example, the Products.ID column in this subquery is the outer reference. The query extracts the names and descriptions of the products whose in-stock quantities are less than double the average ordered quantity of that product—specifically, the product being tested by the WHERE clause in the main query. The subquery does this by scanning the SalesOrderItems table. But the Products.ID column in the

WHERE clause of the subquery refers to a column in the table named in the FROM clause of the *main* query —not the subquery. As the database server moves through each row of the Products table, it uses the ID value of the current row when it evaluates the WHERE clause of the subquery.

A query executes without error when a column referenced in a subquery does not exist in the table referenced by the subquery's FROM clause, but exists in a table referenced by the outer query's FROM clause. SQL Anywhere implicitly qualifies the column in the subquery with the table name in the outer query.

A subquery that does not contain references to objects in a parent statement is called an **uncorrelated subquery**. In the example below, the subquery calculates exactly one value: the average quantity from the SalesOrderItems table. In evaluating the query, the database server computes this value once, and compares each value in the Quantity field of the Products table to it to determine whether to select the corresponding row.

```
SELECT Name, Description
FROM Products
WHERE Quantity <  2 * (
   SELECT AVG( Quantity )
   FROM SalesOrderItems );
```

Nested subqueries

A **nested subquery** is a subquery nested within another subquery. There is no limit to the level of subquery nesting you can define, however, queries with three or more levels take considerably longer to run than do smaller queries.

The following example uses nested subqueries to determine the order IDs and line IDs of those orders shipped on the same day when any item in the fees department was ordered.

```
SELECT ID, LineID
FROM SalesOrderItems
WHERE ShipDate = ANY (
   SELECT OrderDate
   FROM SalesOrders
   WHERE FinancialCode IN (
      SELECT Code
      FROM FinancialCodes
      WHERE ( Description = 'Fees' ) ) );
```

ID	LineID
2001	1
2001	2
2001	3
2002	1
...	...

In this example, the innermost subquery produces a column of financial codes whose descriptions are "Fees":

```
SELECT Code
FROM FinancialCodes
WHERE ( Description = 'Fees' );
```

The next subquery finds the order dates of the items whose codes match one of the codes selected in the innermost subquery:

```
SELECT OrderDate
FROM SalesOrders
WHERE FinancialCode
IN ( subquery-expression );
```

Finally, the outermost query finds the order IDs and line IDs of the orders shipped on one of the dates found in the subquery.

```
SELECT ID, LineID
FROM SalesOrderItems
WHERE ShipDate = ANY ( subquery-expression );
```

Using subqueries instead of joins

Suppose you need a chronological list of orders and the company that placed them, but would like the company name instead of their Customers ID. You can get this result using a join.

Using a join

To list the order ID, date, and company name for each order since the beginning of 2001, execute the following query:

```
SELECT SalesOrders.ID,
            SalesOrders.OrderDate,
            Customers.CompanyName
FROM SalesOrders
   KEY JOIN Customers
WHERE OrderDate > '2001/01/01'
ORDER BY OrderDate;
```

Using a subquery

The following statement obtains the same results using a subquery instead of a join:

```
SELECT SalesOrders.ID,
   SalesOrders.OrderDate,
   (  SELECT CompanyName FROM Customers
       WHERE Customers.ID = SalesOrders.CustomerID )
FROM SalesOrders
WHERE OrderDate > '2001/01/01'
ORDER BY OrderDate;
```

The subquery refers to the CustomerID column in the SalesOrders table even though the SalesOrders table is not part of the subquery. Instead, the SalesOrders.CustomerID column refers to the SalesOrders table in the main body of the statement.

A subquery can be used instead of a join whenever only one column is required from the other table. (Recall that subqueries can only return one column.) In this example, you only needed the CompanyName column, so the join could be changed into a subquery.

Using an outer join

To list all customers in Washington state, together with their most recent order ID, execute the following query:

```
SELECT  CompanyName, State,
   ( SELECT MAX( ID )
       FROM SalesOrders
      WHERE SalesOrders.CustomerID = Customers.ID )
FROM Customers
WHERE State = 'WA';
```

CompanyName	State	MAX(SalesOrders.ID)
Custom Designs	WA	2547
It's a Hit!	WA	(NULL)

The It's a Hit! company placed no orders, and the subquery returns NULL for this customer. Companies who have not placed an order are not listed when inner joins are used.

You could also specify an outer join explicitly. In this case, a GROUP BY clause is also required.

```
SELECT CompanyName, State,
   MAX( SalesOrders.ID )
FROM Customers
   KEY LEFT OUTER JOIN SalesOrders
WHERE State = 'WA'
GROUP BY CompanyName, State;
```

Subqueries in the WHERE clause

Subqueries in the WHERE clause work as part of the row selection process. You use a subquery in the WHERE clause when the criteria you use to select rows depend on the results of another table.

Example

Find the products whose in-stock quantities are less than double the average ordered quantity.

```
SELECT Name, Description
FROM Products WHERE Quantity <  2 * (
   SELECT AVG( Quantity )
   FROM SalesOrderItems );
```

This is a two-step query: first, find the average number of items requested per order; and then find which products in stock number less than double that quantity.

The query in two steps

The Quantity column of the SalesOrderItems table stores the *number* of items requested per item type, customer, and order. The subquery is

```
SELECT AVG( Quantity )
FROM SalesOrderItems;
```

It returns the average quantity of items in the SalesOrderItems table, which is 25.851413.

The next query returns the names and descriptions of the items whose in-stock quantities are less than twice the previously-extracted value.

```
SELECT Name, Description
FROM Products
WHERE Quantity < 2*25.851413;
```

Using a subquery combines the two steps into a single operation.

Purpose of a subquery in the WHERE clause

A subquery in the WHERE clause is part of a search condition. The section “Querying data” on page 255 describes simple search conditions you can use in the WHERE clause.

Subqueries in the HAVING clause

Although you usually use subqueries as search conditions in the WHERE clause, sometimes you can also use them in the HAVING clause of a query. When a subquery appears in the HAVING clause, like any expression in the HAVING clause, it is used as part of the row group selection.

Here is a request that lends itself naturally to a query with a subquery in the HAVING clause: "Which products' average in-stock quantity is more than double the average number of each item ordered per customer?"

Example

```
SELECT Name, AVG( Quantity )
FROM Products
GROUP BY Name
HAVING AVG( Quantity ) > 2* (
   SELECT AVG( Quantity )
   FROM SalesOrderItems
 );
```

name	AVG(Products.Quantity)
Baseball Cap	62.000000
Shorts	80.000000
Tee Shirt	52.333333

The query executes as follows:

- The subquery calculates the average quantity of items in the SalesOrderItems table.
- The main query then goes through the Products table, calculating the average quantity per product, grouping by product name.

- The HAVING clause then checks if each average quantity is more than double the quantity found by the subquery. If so, the main query returns that row group; otherwise, it doesn't.
- The SELECT clause produces one summary row for each group, displaying the name of each product and its in-stock average quantity.

You can also use outer references in a HAVING clause, as shown in the following example, a slight variation on the one above.

Example

This example finds the product ID numbers and line ID numbers of those products whose average ordered quantities is more than half the in-stock quantities of those products.

```
SELECT ProductID, LineID
FROM SalesOrderItems
GROUP BY ProductID, LineID
HAVING 2* AVG( Quantity ) > (
   SELECT Quantity
   FROM Products
   WHERE Products.ID = SalesOrderItems.ProductID );
```

ProductID	LineID
601	3
601	2
601	1
600	2
...	...

In this example, the subquery must produce the in-stock quantity of the product corresponding to the row group being tested by the HAVING clause. The subquery selects records for that particular product, using the outer reference SalesOrderItems.ProductID**.**

A subquery with a comparison returns a single value

This query uses the comparison >, suggesting that the subquery must return exactly one value. In this case, it does. Since the ID field of the Products table is a primary key, there is only one record in the Products table corresponding to any particular product ID.

Testing subqueries

The section "Querying data" on page 255 describes simple search conditions you can use in the HAVING clause. Since a subquery is just an expression that appears in the WHERE or HAVING clauses, the search conditions on subqueries may look familiar.

They include:

- **Subquery comparison test** Compares the value of an expression to a single value produced by the subquery for each record in the table(s) in the main query. Comparison tests use the operators (=, <>, <. <=, >, >=) provided with the subquery.
- **Quantified comparison test** Compares the value of an expression to each of the set of values produced by a subquery.
- **Subquery set membership test** Checks if the value of an expression matches one of the set of values produced by a subquery.
- **Existence test** Checks if the subquery produces any rows.

Subquery comparison test

The subquery comparison test (=, <>, <. <=, >, >=) is a modified version of the simple comparison test. The only difference between the two is that in the former, the expression following the operator is a subquery. This test is used to compare a value from a row in the main query to a *single* value produced by the subquery.

Example

This query contains an example of a subquery comparison test:

```
SELECT Name, Description, Quantity
FROM Products
WHERE Quantity <  2 * (
   SELECT AVG( Quantity )
   FROM SalesOrderItems );
```

name	Description	Quantity
Tee Shirt	Tank Top	28
Baseball Cap	Wool cap	12
Visor	Cloth Visor	36
Visor	Plastic Visor	28
...	...	...

The following subquery retrieves a single value—the average quantity of items of each type per customer's order—from the SalesOrderItems table.

```
SELECT AVG( Quantity )
FROM SalesOrderItems;
```

Then the main query compares the quantity of each in-stock item to that value.

A subquery in a comparison test returns one value

A subquery in a comparison test must return exactly one value. Consider this query, whose subquery extracts two columns from the SalesOrderItems table:

```
SELECT Name, Description, Quantity
FROM Products
WHERE Quantity <  2 * (
   SELECT AVG( Quantity ), MAX( Quantity )
   FROM SalesOrderItems);
```

It returns the error `Subquery allowed only one select list item.`

Subqueries and the IN test

You can use the subquery set membership test to compare a value from the main query to more than one value in the subquery.

The subquery set membership test compares a single data value for each row in the main query to the single column of data values produced by the subquery. If the data value from the main query matches *one* of the data values in the column, the subquery returns TRUE.

Example

Select the names of the employees who head the Shipping or Finance departments:

```
SELECT GivenName, Surname
FROM Employees
WHERE EmployeeID IN (
   SELECT DepartmentHeadID
   FROM Departments
   WHERE ( DepartmentName='Finance' OR
          DepartmentName = 'Shipping' ) );
```

GivenName	Surname
Mary Anne	Shea
Jose	Martinez

The subquery in this example extracts from the Departments table the ID numbers that correspond to the heads of the Shipping and Finance departments. The main query then returns the names of the employees whose ID numbers match one of the two found by the subquery.

```
SELECT DepartmentHeadID
FROM Departments
WHERE ( DepartmentName='Finance' OR
       DepartmentName = 'Shipping' );
```

Set membership test is equivalent to =ANY test

The subquery set membership test is equivalent to the =ANY test. The following query is equivalent to the query from the above example.

```
SELECT GivenName, Surname
FROM Employees
WHERE EmployeeID = ANY (
   SELECT DepartmentHeadID
   FROM Departments
   WHERE ( DepartmentName='Finance' OR
          DepartmentName = 'Shipping' ) );
```

Negation of the set membership test

You can also use the subquery set membership test to extract those rows whose column values are not equal to any of those produced by a subquery. To negate a set membership test, insert the word NOT in front of the keyword IN.

Example

The subquery in this query returns the first and last names of the employees that are not heads of the Finance or Shipping departments.

```
SELECT GivenName, Surname
FROM Employees
WHERE EmployeeID NOT IN (
   SELECT DepartmentHeadID
   FROM Departments
   WHERE ( DepartmentName='Finance' OR
          DepartmentName = 'Shipping' ) );
```

Subqueries and the ANY test

The ANY test, used in conjunction with one of the SQL comparison operators (=, >, <, >=, <=, !=, <>, !>, !<), compares a single value to the column of data values produced by the subquery. To perform the test, SQL uses the specified comparison operator to compare the test value to each data value in the column. If *any* of the comparisons yields a TRUE result, the ANY test returns TRUE.

A subquery used with ANY must return a single column.

Example

Find the order and customer IDs of those orders placed after the first product of the order #2005 was shipped.

```
SELECT ID, CustomerID
FROM SalesOrders
WHERE OrderDate > ANY (
   SELECT ShipDate
   FROM SalesOrderItems
   WHERE ID=2005 );
```

ID	CustomerID
2006	105
2007	106
2008	107

ID	CustomerID
2009	108
...	...

In executing this query, the main query tests the order dates for each order against the shipping dates of *every* product of the order #2005. If an order date is greater than the shipping date for *one* shipment of order #2005, then that ID and customer ID from the SalesOrders table are part of the result set. The ANY test is analogous to the OR operator: the above query can be read, "Was this sales order placed after the first product of the order #2005 was shipped, or after the second product of order #2005 was shipped, or..."

Understanding the ANY operator

The ANY operator can be a bit confusing. It is tempting to read the query as "Return those orders placed after any products of order #2005 were shipped." But this means the query will return the order IDs and customer IDs for the orders placed after *all* products of order #2005 were shipped—which is not what the query does.

Instead, try reading the query like this: "Return the order and customer IDs for those orders placed after *at least one* product of order #2005 was shipped." Using the keyword SOME may provide a more intuitive way to phrase the query. The following query is equivalent to the previous query.

```
SELECT ID, CustomerID
FROM SalesOrders
WHERE OrderDate > SOME (
   SELECT ShipDate
   FROM SalesOrderItems
   WHERE ID=2005 );
```

The keyword SOME is equivalent to the keyword ANY.

Notes about the ANY operator

There are two additional important characteristics of the ANY test:

- **Empty subquery result set** If the subquery produces an empty result set, the ANY test returns FALSE. This makes sense, since if there are no results, then it is not true that at least one result satisfies the comparison test.

- **NULL values in subquery result set** Assume that there is at least one NULL value in the subquery result set. If the comparison test is FALSE for all non-NULL data values in the result set, the ANY search returns UNKNOWN. This is because in this situation, you cannot conclusively state whether there is a value for the subquery for which the comparison test holds. There may or may not be a value, depending on the *correct* values for the NULL data in the result set. For more information about the ANY search condition, see “ANY and SOME search conditions” [*SQL Anywhere Server - SQL Reference*].

Subqueries and the ALL test

The ALL test is used with one of the SQL comparison operators (=, >, <, >=, <=, !=, <>, !>, !<) to compare a single value to the data values produced by the subquery. To perform the test, SQL uses the specified comparison operator to compare the test value to each data value in the result set. If all the comparisons yield TRUE results, the ALL test returns TRUE.

Example

This example finds the order and customer IDs of orders placed after all products of order #2001 were shipped.

```
SELECT ID, CustomerID
FROM SalesOrders
WHERE OrderDate > ALL (
   SELECT ShipDate
   FROM SalesOrderItems
   WHERE ID=2001 );
```

ID	CustomerID
2002	102
2003	103
2004	104
2005	101
...	...

In executing this query, the main query tests the order dates for each order against the shipping dates of *every* product of order #2001. If an order date is greater than the shipping date for *every* shipment of order #2001, then the ID and customer ID from the SalesOrders table are part of the result set. The ALL test is analogous to the AND operator: the above query can be read, "Was this sales order placed before the first product of order #2001 was shipped, and before the second product of order #2001 was shipped, and..."

Notes about the ALL operator

There are three additional important characteristics of the ALL test:

- **Empty subquery result set** If the subquery produces an empty result set, the ALL test returns TRUE. This makes sense, since if there are no results, then it is true that the comparison test holds for every value in the result set.

- **NULL values in subquery result set** If the comparison test is false for any values in the result set, the ALL search returns FALSE. It returns TRUE if all values are true. Otherwise, it returns UNKNOWN—for example, this can occur if there is a NULL value in the subquery result set but the search condition is TRUE for all non-NULL values.

- **Negating the ALL test** The following expressions are *not* equivalent.

  ```
  NOT a = ALL (subquery)
  a <> ALL (subquery)
  ```


For more information about this test, see "Subquery that follows ANY, ALL or SOME" on page 508.

Subqueries and the EXISTS test

Subqueries used in the subquery comparison test and set membership test both return data values from the subquery table. Sometimes, however, you may be more concerned with whether the subquery returns *any* results, rather than *which* results. The existence test (EXISTS) checks whether a subquery produces any rows of query results. If the subquery produces one or more rows of results, the EXISTS test returns TRUE. Otherwise, it returns FALSE.

Example

Here is an example of a request expressed using a subquery: "Which customers placed orders after July 13, 2001?"

```
SELECT GivenName, Surname
FROM Customers
WHERE EXISTS (
   SELECT *
   FROM SalesOrders
   WHERE ( OrderDate > '2001-07-13' ) AND
         ( Customers.ID = SalesOrders.CustomerID ) );
```

GivenName	Surname
Almen	de Joie
Grover	Pendelton
Ling Ling	Andrews
Bubba	Murphy

Explanation of the existence test

Here, for each row in the Customers table, the subquery checks if that customer ID corresponds to one that has placed an order after July 13, 2001. If it does, the query extracts the first and last names of that customer from the main table.

The EXISTS test does not use the results of the subquery; it just checks if the subquery produces any rows. So the existence test applied to the following two subqueries return the same results. These are subqueries and cannot be processed on their own, because they refer to the Customers table which is part of the main query, but not part of the subquery.

For more information, see "Correlated and uncorrelated subqueries" on page 494.

```
SELECT *
FROM Customers, SalesOrders
WHERE ( OrderDate > '2001-07-13' ) AND
      ( Customers.ID = SalesOrders.CustomerID )

SELECT OrderDate
```

```
FROM Customers, SalesOrders
WHERE ( OrderDate > '2001-07-13' ) AND
      ( Customers.ID = SalesOrders.CustomerID );
```

It does not matter which columns from the SalesOrders table appear in the SELECT statement, though by convention, the "SELECT *" notation is used.

Negating the existence test

You can reverse the logic of the EXISTS test using the NOT EXISTS form. In this case, the test returns TRUE if the subquery produces no rows, and FALSE otherwise.

Correlated subqueries

You may have noticed that the subquery contains a reference to the ID column from the Customers table. A reference to columns or expressions in the main table(s) is called an **outer reference** and the subquery is said to be **correlated**. Conceptually, SQL processes the above query by going through the Customers table, and performing the subquery for each customer. If the order date in the SalesOrders table is after July 13, 2001, and the customer ID in the Customers and SalesOrders tables match, then the first and last names from the Customers table appear. Since the subquery references the main query, the subquery in this section, unlike those from previous sections, returns an error if you attempt to run it by itself.

Optimizer automatic conversion of subqueries to joins

The query optimizer automatically rewrites as joins many of the queries that make use of subqueries. The conversion is performed without any user action. This section describes which subqueries can be converted to joins so you can understand the performance of queries in your database.

The criteria that must be satisfied in order for a multi-level query to be able to be rewritten with joins differ for the various types of operators, and the structures of the query and of the subquery. Recall that when a subquery appears in the query's WHERE clause, it is of the form

SELECT *select-list*
FROM *table*
WHERE
[**NOT**] *expression comparison-operator* **(** *subquery-expression* **)**
| [**NOT**] *expression comparison-operator* { **ANY** | **SOME** } **(** *subquery-expression* **)**
| [**NOT**] *expression comparison-operator* **ALL (** *subquery-expression* **)**
| [**NOT**] *expression* **IN (** *subquery-expression* **)**
| [**NOT**] **EXISTS (** *subquery-expression* **)**
GROUP BY *group-by-expression*
HAVING *search-condition*

For example, consider the request, "When did Mrs. Clarke and Suresh place their orders, and by which sales representatives?" It can be answered with the following query:

```
SELECT OrderDate, SalesRepresentative
FROM SalesOrders
WHERE CustomerID IN (
   SELECT ID
   FROM Customers
   WHERE Surname = 'Clarke' OR GivenName = 'Suresh' );
```


OrderDate	SalesRepresentative
2001-01-05	1596
2000-01-27	667
2000-11-11	467
2001-02-04	195
...	...

The subquery yields a list of customer IDs that correspond to the two customers whose names are listed in the WHERE clause, and the main query finds the order dates and sales representatives corresponding to those two people's orders.

The same question can be answered using joins. Here is an alternative form of the query, using a two-table join:

```
SELECT OrderDate, SalesRepresentative
FROM SalesOrders, Customers
WHERE CustomerID=Customers.ID AND
  ( Surname = 'Clarke' OR GivenName = 'Suresh' );
```

This form of the query joins the SalesOrders table to the Customers table to find the orders for each customer, and then returns only those records for Suresh and Clarke.

Case where a subquery works, but a join does not

There are cases where a subquery works but a join does not. For example:

```
SELECT Name, Description, Quantity
FROM Products
WHERE Quantity <  2 * (
   SELECT AVG( Quantity )
   FROM SalesOrderItems );
```

name	Description	Quantity
Tee Shirt	Tank Top	28
Baseball Cap	Wool cap	12
Visor	Cloth Visor	36
...	...	...

In this case, the inner query is a summary query and the outer query is not, so there is no way to combine the two queries by a simple join.

See also

- “Joins: Retrieving data from several tables” on page 386

Subquery that follows a comparison operator

A subquery that follows a comparison operator (=, >, <, >=, <=, !=, <>, !>, !<) is called a comparison. The optimizer converts these subqueries to joins if the subquery:

- returns exactly one value for each row of the main query.
- does not contain a GROUP BY clause
- does not contain the keyword DISTINCT
- is not a UNION query
- is not an aggregate query

Example

Suppose the request "When were Suresh's products ordered, and by which sales representative?" were phrased as the subquery

```
SELECT OrderDate, SalesRepresentative
FROM SalesOrders
WHERE CustomerID = (
   SELECT ID
   FROM Customers
   WHERE GivenName = 'Suresh' );
```

This query satisfies the criteria, and therefore, it would be converted to a query using a join:

```
SELECT OrderDate, SalesRepresentative
FROM SalesOrders, Customers
WHERE CustomerID=Customers.ID AND
  ( Surname = 'Clarke' OR GivenName = 'Suresh' );
```

However, the request, "Find the products whose in-stock quantities are less than double the average ordered quantity" cannot be converted to a join, as the subquery contains the AVG aggregate function:

```
SELECT Name, Description
FROM Products
WHERE Quantity <  2 * (
   SELECT AVG( Quantity )
   FROM SalesOrderItems );
```

Subquery that follows ANY, ALL or SOME

A subquery that follows the keywords ALL, ANY, or SOME is called a quantified comparison. The optimizer converts these subqueries to joins if:

- The main query does not contain a GROUP BY clause, and is not an aggregate query, or the subquery returns exactly one value.
- The subquery does not contain a GROUP BY clause.
- The subquery does not contain the keyword DISTINCT.
- The subquery is not a UNION query.
- The subquery is not an aggregate query.
- The conjunct '*expression comparison-operator* { **ANY** | **SOME** } **(** *subquery-expression* **)**' must not be negated.
- The conjunct '*expression comparison-operator* **ALL (** *subquery-expression* **)**' must be negated.

The first four of these conditions are relatively straightforward.

Example

The request "When did Ms. Clarke and Suresh place their orders, and by which sales representatives?" can be handled in subquery form:

```
SELECT OrderDate, SalesRepresentative
FROM SalesOrders
WHERE CustomerID = ANY (
   SELECT ID
   FROM Customers
   WHERE Surname = 'Clarke' OR GivenName = 'Suresh' );
```

Alternately, it can be phrased in join form

```
SELECT OrderDate, SalesRepresentative
FROM SalesOrders, Customers
WHERE CustomerID=Customers.ID AND
  ( Surname = 'Clarke' OR GivenName = 'Suresh' );
```

However, the request, "When did Ms. Clarke, Suresh, and any employee who is also a customer, place their orders?" would be phrased as a union query, and cannot be converted to a join:

```
SELECT OrderDate, SalesRepresentative
FROM SalesOrders
WHERE CustomerID = ANY (
   SELECT ID
   FROM Customers
   WHERE Surname = 'Clarke' OR GivenName = 'Suresh'
   UNION
   SELECT EmployeeID
   FROM Employees );
```

Similarly, the request "Find the order IDs and customer IDs of those orders not shipped after the first shipping dates of all the products" would be phrased as the aggregate query, and therefore cannot be converted to a join:

```
SELECT ID, CustomerID
FROM SalesOrders
WHERE NOT OrderDate > ALL (
```

```
SELECT FIRST ( ShipDate )
FROM SalesOrderItems
ORDER BY ShipDate );
```

Negating subqueries with the ANY and ALL operators

The fifth criterion is a little more puzzling. Queries taking the following form are converted to joins:

SELECT *select-list*
FROM *table*
WHERE NOT *expression comparison-operator* **ALL (** *subquery-expression* **)**

SELECT *select-list*
FROM *table*
WHERE *expression comparison-operator* **ANY (** *subquery-expression* **)**

However, the following queries are not converted to joins:

SELECT *select-list*
FROM *table*
WHERE *expression comparison-operator* **ALL (** *subquery-expression* **)**

SELECT *select-list*
FROM *table*
WHERE NOT *expression comparison-operator* **ANY (** *subquery-expression* **)**

The first two queries are equivalent, as are the last two. Recall that the ANY operator is analogous to the OR operator, but with a variable number of arguments; and that the ALL operator is similarly analogous to the AND operator. For example, the following two expressions are equivalent:

```
NOT ( ( X > A ) AND ( X > B ) )
( X <= A ) OR ( X <= B )
```

The following two expressions are also equivalent:

```
WHERE NOT OrderDate > ALL (
   SELECT FIRST ( ShipDate )
   FROM SalesOrderItems
   ORDER BY ShipDate )

WHERE OrderDate <= ANY (
   SELECT FIRST ( ShipDate )
   FROM SalesOrderItems
   ORDER BY ShipDate )
```

Negating the ANY and ALL expressions

In general, the following expressions are equivalent:

NOT *column-name operator* **ANY (** *subquery-expression* **)**

column-name inverse-operator **ALL (** *subquery-expression* **)**

These expressions are generally equivalent as well:

NOT *column-name operator* **ALL (** *subquery-expression* **)**

column-name inverse-operator **ANY (** *subquery-expression* **)**

where *inverse-operator* is obtained by negating *operator*, as shown in the table below:

operator	inverse-operator
=	<>
<	=>
>	=<
=<	>
=>	<
<>	=

Subquery that follows IN

The optimizer converts a subquery that follows an IN keyword only if:

- The main query does not contain a GROUP BY clause, and is not an aggregate query, or the subquery returns exactly one value.
- The subquery does not contain a GROUP BY clause.
- The subquery does not contain the keyword DISTINCT.
- The subquery is not a UNION query.
- The subquery is not an aggregate query.
- The conjunct '*expression* IN (*subquery-expression*)' must not be negated.

Example

So, the request "Find the names of the employees who are also department heads", expressed by the following query, would be converted to a joined query, as it satisfies the conditions.

```
SELECT GivenName, Surname
FROM Employees
WHERE EmployeeID IN (
   SELECT DepartmentHeadID
   FROM Departments
   WHERE ( DepartmentName ='Finance' OR
          DepartmentName = 'Shipping' ) );
```

However, the request, "Find the names of the employees who are either department heads or customers" would not be converted to a join if it were expressed by the UNION query.

A UNION query following the IN operator cannot be converted

```
SELECT GivenName, Surname
FROM Employees
```

```
WHERE EmployeeID IN (
   SELECT DepartmentHeadID
   FROM Departments
   WHERE ( DepartmentName='Finance' OR
          DepartmentName = 'Shipping' )
   UNION
   SELECT CustomerID
   FROM SalesOrders);
```

Similarly, the request "Find the names of employees who are not department heads" is formulated as the negated subquery shown below, and would not be converted

```
SELECT GivenName, Surname
FROM Employees
  WHERE NOT EmployeeID IN (
   SELECT DepartmentHeadID
   FROM Departments
   WHERE ( DepartmentName='Finance' OR
          DepartmentName = 'Shipping' ) );
```

The conditions necessary for an IN or ANY subquery to be converted to a join are identical. This is because the two expressions are logically equivalent.

Query with IN operator converted to a query with an ANY operator

Sometimes SQL Anywhere converts a query with the IN operator to one with an ANY operator, and decides whether to convert the subquery to a join. For example, the following two expressions are equivalent:

WHERE *column-name* **IN(** *subquery-expression* **)**

WHERE *column-name* = **ANY(** *subquery-expression* **)**

Likewise, the following two queries are equivalent:

```
SELECT GivenName, Surname
FROM Employees
WHERE EmployeeID IN (
   SELECT DepartmentHeadID
   FROM Departments
   WHERE ( DepartmentName='Finance' OR
          DepartmentName = 'Shipping' ) );

SELECT GivenName, Surname
FROM Employees
WHERE EmployeeID = ANY (
   SELECT DepartmentHeadID
   FROM Departments
   WHERE ( DepartmentName='Finance' OR
          DepartmentName = 'Shipping' ) );
```

Subquery that follows EXISTS

The optimizer converts a subquery that follows the EXISTS keyword only if:

- The main query does not contain a GROUP BY clause, and is not an aggregate query, or the subquery returns exactly one value.

- The conjunct 'EXISTS (subquery)' is not negated.
- The subquery is correlated; that is, it contains an outer reference.

Example

The request, "Which customers placed orders after July 13, 2001?", which can be formulated by a query whose non-negated subquery contains the outer reference **Customers.ID** = **SalesOrders.CustomerID**, can be represented with the following join:

```
SELECT GivenName, Surname
FROM Customers
WHERE EXISTS (
   SELECT *
   FROM SalesOrders
   WHERE ( OrderDate > '2001-07-13' ) AND
         ( Customers.ID = SalesOrders.CustomerID ) );
```

The EXISTS keyword tells the database server to check for empty result sets. When using inner joins, the database server automatically displays only the rows where there is data from all the tables in the FROM clause. So, this query returns the same rows as does the one with the subquery:

```
SELECT DISTINCT GivenName, Surname
FROM Customers, SalesOrders
WHERE ( SalesOrders.OrderDate > '2001-07-13' ) AND
      ( Customers.ID = SalesOrders.CustomerID );
```

Adding, changing, and deleting data

Data modification statements

The statements you use to add, change, or delete data are called data modification statements which are a subset of the data manipulation language (DML) statements part of ANSI SQL. The main DML statements are:

- **INSERT statement** Adds new rows to a table or view. See “INSERT statement” [*SQL Anywhere Server - SQL Reference*].
- **UPDATE statement** The UPDATE statement changes rows in a set of tables or views. See “UPDATE statement” [*SQL Anywhere Server - SQL Reference*].
- **DELETE statement** The DELETE statement removes rows from a set of tables or views. “DELETE statement” [*SQL Anywhere Server - SQL Reference*].
- **MERGE statement** The MERGE statement adds, changes, and removes specific rows from a table or view. See “MERGE statement” [*SQL Anywhere Server - SQL Reference*].

In addition to the statements above, the LOAD TABLE and TRUNCATE TABLE statements are especially useful for bulk loading and deleting of data.

Permissions for data modification

You can only execute data modification statements if you have the proper permissions on the database tables you want to modify. The database administrator and the owners of database objects use the GRANT and REVOKE statements to decide who has access to which data modification functions.

Permissions can be granted to individual users, groups, or the PUBLIC group. For more information on permissions, see "Managing user IDs, authorities, and permissions" [*SQL Anywhere Server - Database Administration*].

Transactions and data modification

When you modify data, the rollback log stores a copy of the old and new state of each row affected by each data modification statement. This means that if you begin a transaction, realize you have made a mistake, and roll the transaction back, you restore the database to its previous condition. See "Using transactions and isolation levels" on page 89.

Making changes permanent

The COMMIT statement makes all changes permanent.

You should use the COMMIT statement after groups of statements that make sense together. For example, if you want to transfer money from one customer's account to another, you should add money to one customer's account, then delete it from the other's, and then commit, since in this case it does not make sense to leave your database with less or more money than it started with.

You can instruct Interactive SQL to commit your changes automatically by setting the auto_commit option to On. This is an Interactive SQL option. When auto_commit is set to On, Interactive SQL issues a COMMIT statement after every insert, update, and delete statement you make. This can slow down performance considerably. Therefore, it is a good idea to leave the auto_commit option set to Off.

Use COMMIT with care
When trying the examples in this tutorial, be careful not to commit changes until you are sure that you want to change the database permanently. See "COMMIT statement" [*SQL Anywhere Server - SQL Reference*].

See also

- "Interactive SQL options" [*SQL Anywhere Server - Database Administration*]

Canceling changes

Any uncommitted change you make can be canceled. SQL allows you to undo all the changes you made since your last commit with the ROLLBACK statement. This statement undoes all changes you have

made to the database since the last time you made changes permanent. See "ROLLBACK statement" [*SQL Anywhere Server - SQL Reference*].

Transactions and data recovery

SQL Anywhere protects the integrity of your database in the event of a system failure or power outage. You have several different options for restoring your database server. For example, the log file that SQL Anywhere stores on a separate drive can be used to restore your data. When using a log file for recovery, SQL Anywhere does not need to update your database as frequently, and the performance of your database server is improved.

Transaction processing allows the database server to identify situations in which your data is in a consistent state. Transaction processing ensures that if, for any reason, a transaction is not successfully completed, then the entire transaction is undone, or rolled back. The database is left entirely unaffected by failed transactions.

The transaction processing in SQL Anywhere ensures that the contents of a transaction are processed securely, even in the event of a system failure in the middle of a transaction.

See also

- "Backup and data recovery" [*SQL Anywhere Server - Database Administration*]

Referential integrity

SQL Anywhere automatically checks for some common errors in your data when inserting, updating, and deleting data. This kind of validity checking is called **enforcing referential integrity** as it checks the integrity of data within and between tables in the database. See "Enforcing entity and referential integrity" on page 81.

Adding data using INSERT

You add rows to the database using the INSERT statement. The INSERT statement has two forms: you can use the VALUES keyword or a SELECT statement:

INSERT using values

The VALUES keyword specifies values for some or all the columns in a new row. A simplified version of the syntax for the INSERT statement using the VALUES keyword is:

INSERT [**INTO**] *table-name* [(*column-name*, ...)]
VALUES (*expression*, ...)

You can omit the list of column names if you provide a value for each column in the table, in the order in which they appear when you execute a query using SELECT *.

INSERT from SELECT

You can use SELECT within an INSERT statement to pull values from one or more tables. If the table you are inserting data into has a large number of columns, you can also use WITH AUTO NAME to simplify the syntax. Using WITH AUTO NAME, you only need to specify the column names in the SELECT statement, rather than in both the INSERT and the SELECT statements. The names in the SELECT statement must be column references or aliased expressions.

A simplified version of the syntax for the INSERT statement using a select statement is:

INSERT [**INTO**] *table-name*
[**WITH AUTO NAME**] *select-statement*

For more information about the INSERT statement, see "INSERT statement" [*SQL Anywhere Server - SQL Reference*].

Inserting values into all columns of a row

The following INSERT statement adds a new row to the Departments table, giving a value for every column in the row:

```
INSERT INTO Departments
VALUES ( 702, 'Eastern Sales', 902 );
```

Notes

- Type the values in the same order as the column names in the original CREATE TABLE statement, that is, first the ID number, then the name, then the department head ID.
- Surround the values by parentheses.
- Enclose all character data in single quotes.
- Use a separate insert statement for each row you add.

Inserting values into specific columns

You can add data to some columns in a row by specifying only those columns and their values. Define all other columns not included in the column list to allow NULL or have defaults. If you skip a column that has a default value, the default appears in that column.

Adding data in only two columns, for example, DepartmentID and DepartmentName, requires a statement like this:

```
INSERT INTO Departments ( DepartmentID, DepartmentName )
VALUES ( 703, 'Western Sales' );
```

DepartmentHeadID does not have a default value but accepts NULL. therefore a NULL is automatically assigned to that column.

Cancel these changes to the database by entering a ROLLBACK statement:

```
ROLLBACK;
```

While the column order you specify does not need to match the order of columns in the table, it must match the order in which you specify the values you are inserting.

Inserted values for specified and unspecified columns

Values are inserted in a row according to what is specified in the INSERT statement. If no value is specified for a column, the inserted value depends on column settings such as whether to allow NULLs, whether to use a DEFAULT, and so on. Sometimes the insert operation fails and an error is returned. The following table shows the possible outcomes depending on the value being inserted (if any) and the column settings:

Value being inserted	Nullable	Not nullable	Nullable, with DEFAULT	Not nullable, with DEFAULT	Not nullable, with DEFAULT AUTOINCREMENT or DEFAULT [UTC] TIMESTAMP
<none>	NULL	SQL error	DEFAULT value	DEFAULT value	DEFAULT value
NULL	NULL	SQL error	NULL	SQL error	DEFAULT value
specified value	specified value	specified value	specified value	specified value	specified value

By default, columns allow NULL values unless you explicitly state NOT NULL in the column definition when creating a table. You can alter this default using the allow_nulls_by_default option. You can also alter whether a specific column allows NULLs using the ALTER TABLE statement. See “allow_nulls_by_default option” [*SQL Anywhere Server - Database Administration*] and “ALTER TABLE statement” [*SQL Anywhere Server - SQL Reference*].

Restricting column data using constraints

You can create constraints for a column or domain. Constraints govern the kind of data you can or cannot add. See “Using table and column constraints” on page 74.

Explicitly inserting NULL

You can explicitly insert NULL into a column by entering NULL. Do not enclose this in quotes, or it will be taken as a string. For example, the following statement explicitly inserts NULL into the DepartmentHeadID column:

```
INSERT INTO Departments
VALUES ( 703, 'Western Sales', NULL );
```

Using defaults to supply values

You can define a column so that, even though the column receives no value, a default value automatically appears whenever a row is inserted. You do this by supplying a default for the column. See "Using column defaults" on page 67.

Adding new rows with SELECT

To pull values into a table from one or more other tables, you can use a SELECT clause in the INSERT statement. The select clause can insert values into some or all the columns in a row.

Inserting values for only some columns can come in handy when you want to take some values from an existing table. Then, you can use update to add the values for the other columns.

Before inserting values for some, but not all, columns in a table, make sure that either a default exists, or you specify NULL for the columns for which you are not inserting values. Otherwise, an error appears.

When you insert rows from one table into another, the two tables must have compatible structures—that is, the matching columns must be either the same data types or data types between which SQL Anywhere automatically converts.

Example

If the columns are in the same order in both tables, you do not need to specify column names in either table. For example, suppose you have a table named NewProducts that has the same schema as the Products table and contains some rows of product information that you want to add to the Products table. You could execute the following statement:

```
INSERT Products
SELECT *
FROM NewProducts;
```

Inserting data into some columns

You can use the SELECT statement to add data to some, but not all, columns in a row just as you do with the VALUES clause. Simply specify the columns to which you want to add data in the INSERT clause.

Inserting data from the same table

You can insert data into a table based on other data in the same table. Essentially, this means copying all or part of a row.

For example, you can insert new products, based on existing products, into the Products table. The following statement adds new Extra Large Tee Shirts (of Tank Top, V-neck, and Crew Neck varieties) into the Products table. The identification number is 30 greater than the existing sized shirt:

```
INSERT INTO Products
SELECT ID + 30, Name, Description,
    'Extra large', Color, 50, UnitPrice, NULL
FROM Products
WHERE Name = 'Tee Shirt';
```


Inserting documents and images

If you want to store documents or images in your database, you can write an application that reads the contents of the file into a variable, and supplies that variable as a value for an INSERT statement. See "How to use prepared statements" [*SQL Anywhere Server - Programming*], and "SET statement" [*SQL Anywhere Server - SQL Reference*].

You can also use the xp_read_file system function to insert file contents into a table. This function is useful if you want to insert file contents from Interactive SQL, or some other environment that does not provide a full programming language.

DBA authority is required to use this function.

Example

In this example, you create a table, and insert an image into a column of the table. You can perform these steps from Interactive SQL.

1. Create a table to hold images.

```
CREATE TABLE Pictures
( C1 INT DEFAULT AUTOINCREMENT PRIMARY KEY,
   Filename VARCHAR(254),
   Picture LONG BINARY );
```

2. Insert the contents of *portrait.gif*, in the current working directory of the database server, into the table.

```
INSERT INTO Pictures ( Filename, Picture )
VALUES ( 'portrait.gif',
   xp_read_file( 'portrait.gif' ) );
```

See also

- "xp_read_file system procedure" [*SQL Anywhere Server - SQL Reference*]
- "Using openxml with xp_read_file" on page 669
- "Storing BLOBs" [*SQL Anywhere Server - Database Administration*]
- "CREATE TABLE statement" [*SQL Anywhere Server - SQL Reference*]
- "INSERT statement" [*SQL Anywhere Server - SQL Reference*]

Changing data using UPDATE

You can use the UPDATE statement, followed by the name of the table or view, to change single rows, groups of rows, or all rows in a table. As in all data modification statements, you can change the data in only one table or view at a time.

The UPDATE statement specifies the row or rows you want changed and the new data. The new data can be a constant or an expression that you specify or data pulled from other tables.

If an UPDATE statement violates an integrity constraint, the update does not take place and an error message appears. For example, if one of the values being added is the wrong data type, or if it violates a constraint defined for one of the columns or data types involved, the update does not take place.

UPDATE syntax

A simplified version of the UPDATE syntax is:

UPDATE *table-name*
SET *column_name = expression*
WHERE *search-condition*

If the company Newton Ent. (in the Customers table of the SQL Anywhere sample database) is taken over by Einstein, Inc., you can update the name of the company using a statement such as the following:

```
UPDATE Customers
SET CompanyName = 'Einstein, Inc.'
WHERE CompanyName = 'Newton Ent.';
```

You can use any expression in the WHERE clause. If you are not sure how the company name was spelled, you could try updating any company called Newton, with a statement such as the following:

```
UPDATE Customers
SET CompanyName = 'Einstein, Inc.'
WHERE CompanyName LIKE 'Newton%';
```

The search condition need not refer to the column being updated. The company ID for Newton Entertainments is 109. As the ID value is the primary key for the table, you could be sure of updating the correct row using the following statement:

```
UPDATE Customers
SET CompanyName = 'Einstein, Inc.'
WHERE ID = 109;
```

Tip
You can also modify rows from the result set in Interactive SQL. See "Editing result sets in Interactive SQL" [*SQL Anywhere Server - Database Administration*].

SET clause

The SET clause specifies the columns to be updated, and their new values. The WHERE clause determines the row or rows to be updated. If you do not have a WHERE clause, the specified columns of all rows are updated with the values given in the SET clause.

You can provide any expression of the correct data type in the SET clause.

WHERE clause

The WHERE clause specifies the rows to be updated. For example, the following statement replaces the One Size Fits All Tee Shirt with an Extra Large Tee Shirt

```
UPDATE Products
SET Size  = 'Extra Large'
WHERE Name = 'Tee Shirt'
   AND Size = 'One Size Fits All';
```

FROM clause

You can use a FROM clause to pull data from one or more tables into the table you are updating.

Changing data using INSERT

You can use the ON EXISTING clause of the INSERT statement to update existing rows in a table (based on primary key lookup) with new values. This clause can only be used on tables that have a primary key. Attempting to use this clause on tables without primary keys or on proxy tables generates a syntax error.

Specifying the ON EXISTING clause causes the server to do a primary key lookup for each input row. If the corresponding row does not exist, it inserts the new row. For rows already existing in the table, you can choose to:

- generate an error for duplicate key values. This is the default behavior if the ON EXISTING clause is not specified.
- silently ignore the input row, without generating any errors.
- update the existing row with the values in the input row.

For more information, see "INSERT statement" [*SQL Anywhere Server - SQL Reference*].

Deleting data using DELETE

Simple DELETE statements have the following form:

DELETE [**FROM**] *table-name*
WHERE *column-name = expression*

You can also use a more complex form, as follows

DELETE [**FROM**] *table-name*
FROM *table-list*
WHERE *search-condition*

WHERE clause

Use the WHERE clause to specify which rows to remove. If no WHERE clause appears, the DELETE statement remove all rows in the table.

FROM clause

The FROM clause in the second position of a DELETE statement is a special feature allowing you to select data from a table or tables and delete corresponding data from the first-named table. The rows you select in the FROM clause specify the conditions for the delete. See "DELETE statement" [*SQL Anywhere Server - SQL Reference*].

Example

This example uses the SQL Anywhere sample database. To execute the statements in the example, you should set the option wait_for_commit to On. The following statement does this for the current connection only:

```
SET TEMPORARY OPTION wait_for_commit = 'On';
```

This allows you to delete rows even if they contain primary keys referenced by a foreign key, but does not permit a COMMIT unless the corresponding foreign key is deleted also.

The following view displays products and the value of that product that has been sold:

```
CREATE VIEW ProductPopularity as
SELECT  Products.ID,
   SUM( Products.UnitPrice * SalesOrderItems.Quantity )
   AS "Value Sold"
FROM  Products JOIN SalesOrderItems
ON Products.ID = SalesOrderItems.ProductID
GROUP BY Products.ID;
```

Using this view, you can delete those products which have sold less than $20,000 from the Products table.

```
DELETE
FROM Products
FROM Products NATURAL JOIN ProductPopularity
WHERE "Value Sold" < 20000;
```

Cancel these changes to the database by entering a ROLLBACK statement:

```
ROLLBACK;
```

Tip
You can also delete rows from database tables from the Interactive SQL result set. See "Editing result sets in Interactive SQL" [*SQL Anywhere Server - Database Administration*].

Deleting all rows from a table

You can use the TRUNCATE TABLE statement as a fast method of deleting all the rows in a table. It is faster than a DELETE statement with no conditions, because the DELETE logs each change, while TRUNCATE does not record individual rows deleted.

The table definition for a table emptied with the TRUNCATE TABLE statement remains in the database, along with its indexes and other associated objects, unless you execute a DROP TABLE statement.

You cannot use TRUNCATE TABLE if another table has rows that reference it through a referential integrity constraint. Delete the rows from the foreign table, or truncate the foreign table and then truncate the primary table.

Truncating base tables or performing bulk loading operations causes data in indexes (regular or text) and dependent materialized views to become stale. You should first truncate the data in the indexes and dependent materialized views, execute the INPUT statement, and then rebuild or refresh the indexes and materialized views. See "TRUNCATE statement" [*SQL Anywhere Server - SQL Reference*], and "TRUNCATE TEXT INDEX statement" [*SQL Anywhere Server - SQL Reference*].

TRUNCATE TABLE syntax

The syntax of TRUNCATE TABLE is:

TRUNCATE TABLE *table-name*

For example, to remove all the data in the SalesOrders table, enter the following:

```
TRUNCATE TABLE SalesOrders;
```

A TRUNCATE TABLE statement does not fire triggers defined on the table.

Cancel these changes to the database by entering a ROLLBACK statement:

```
ROLLBACK;
```

Query optimization and execution

Optimization is essential in generating a suitable access plan for a query. Once each query is parsed, the optimizer analyzes it and decides on an access plan that computes the result using as few resources as possible. Optimization begins just before execution. If you are using cursors in your application, optimization commences when the cursor is opened. Unlike many other commercial database systems, SQL Anywhere usually optimizes each statement just before executing it. Because SQL Anywhere performs just-in-time optimization of each statement, the optimizer has access to the values of host and stored procedure variables, which allows for better selectivity estimation analysis. In addition, just-in-time optimization allows the optimizer to adjust its choices based on the statistics saved after previous query executions.

Query processing phases

This section describes the phases that a statement goes through starting with the annotation phase and ending with its execution. It also describes the assumptions that underlie the design of the optimizer, and discusses selectivity estimation, cost estimation, and the steps of query processing.

Statements that have no result sets, such as UPDATE or DELETE statements, go through the query processing phases.

- **Annotation phase** When the database server receives a query, it uses a parser to parse the statement and transform it into an algebraic representation of the query, also known as a parse tree. At this stage the **parse tree** is used for semantic and syntactic checking (for example, validating that objects referenced in the query exist in the catalog), permission checking, KEY JOINs and NATURAL JOINs transformation using defined referential constraints, and non-materialized view expansion. The output of this phase is a rewritten query, in the form of a parse tree, which contains annotation to all the objects referenced in the original query.

- **Semantic transformation phase** During this phase, the query undergoes iterative semantic transformations. While the query is still represented as an annotated parse tree, rewrite optimizations, such as join elimination, DISTINCT elimination, and predicate normalization, are applied in this phase. The semantic transformations in this phase are performed based on semantic transformation rules that are applied heuristically to the parse tree representation. See “Semantic query transformations” on page 528.

  Queries with plans already cached by the database server skip this phase of query processing. Simple statements may also skip this phase of query processing. For example, many statements that use heuristic plan selection in the optimizer bypass are not processed by the semantic transformation phase. The complexity of the SQL statement determines if this phase is applied to a statement. See “Plan caching” on page 554, and “Eligibility to skip query processing phases” on page 526.

- **Optimization phase** The optimization phase uses a different internal representation of the query, the query optimization structure, which is built from the parse tree. See “How the optimizer works” on page 540.

Queries with plans already cached by the database server skip this phase of query processing. As well, simple statements may also skip this phase of query processing. See "Plan caching" on page 554, and "Eligibility to skip query processing phases" on page 526.

This phase is broken into two sub-phases:

- **Pre-optimization phase** The pre-optimization phase completes the optimization structure with the information needed later in the enumeration phase. During this phase the query is analyzed to find all relevant indexes and materialized views that may be used in the query access plan. For example, in this phase, the View Matching algorithm determines all the materialized views that may be used to satisfy all, or part of the query. In addition, based on query predicate analysis, the optimizer builds alternative join methods that may be used in the enumeration phase to join the query's tables. During this phase, no decision is made regarding the best access plan for the query; the goal of this phase is to prepare for the enumeration phase.

- **Enumeration phase** During this phase, the optimizer enumerates possible access plans for the query using the building blocks generated in the pre-optimization phase. The search space is very large and the optimizer uses a proprietary enumeration algorithm to generate and prune the generated access plans. For each plan, cost estimation is computed, which is used to compare the current plan with the best plan found so far. Expensive plans are discarded during these comparisons. Cost estimation takes into account resource utilization such as disk and CPU operations, the estimated number of rows of the intermediate results, optimization goal, cache size, and so on. The output of the enumeration phase is the best access plan for the query.

- **Plan building phase** The plan building phase takes the best access plan and builds the corresponding final representation of the query execution plan used to execute the query. You can see a graphical version of the plan in the Plan Viewer in Interactive SQL. The graphical plan has a tree structure where each node is a physical operator implementing a specific relational algebraic operation, for example, Hash Join and Ordered Group By are physical operators implementing a join and a group by operation, respectively. See "Reading graphical plans" on page 595.

  Queries with plans already cached by the database server skip this phase of query processing. See "Plan caching" on page 554, and "Eligibility to skip query processing phases" on page 526.

- **Execution phase** The result of the query is computed using the query execution plan built in the plan building phase.

Eligibility to skip query processing phases

Almost all statements pass through all query processing phases. However, there are two main exceptions: queries that benefit from **plan caching** (queries whose plans are already cached by the database server), and **bypass queries**.

- **Plan caching** For queries contained inside stored procedures and user-defined functions, the database server may cache the execution plans so that they can be reused. For this class of queries, the query execution plan is cached after execution. The next time the query is executed, the plan is retrieved and all the phases up to the execution phase are skipped. See "Plan caching" on page 554.

- **Bypass queries** Bypass queries are a subclass of simple queries that have certain characteristics that the database server recognizes as making them eligible for bypassing the optimizer. Bypassing optimization can reduce the time needed to build an execution plan.

  If a query is recognized as a bypass query, a heuristic rather than cost-based optimization is used—that is, the semantic transformation and optimization phases may be skipped and the query execution plan is built directly from the parse tree representation of the query.

Simple queries

A simple query is a SELECT, INSERT, DELETE, or UPDATE statement with a single query block and the following characteristics:

- The query block does not contain subqueries or additional query blocks such as those for UNION, INTERSECT, EXCEPT, and common table expressions.
- The query block references a single base table or materialized view.
- The query block may include the TOP N, FIRST, ORDER BY, or DISTINCT clauses.
- The query block may include aggregate functions without GROUP BY or HAVING clauses.
- The query block does not include window functions.
- The query block expressions do not include NUMBER, IDENTITY, or subqueries.
- The constraints defined on the base table are simple expressions.

A complex statement may be transformed into a simple statement after the semantic transformation phase. When this occurs, the query can be processed by the optimizer bypass or have its plan cached by the SQL Anywhere Server.

Forcing optimization, and forcing no optimization

You can force queries that qualify for plan caching, or for bypassing the optimizer, to be processed by the SQL Anywhere optimizer. To do so, use the FORCE OPTIMIZATION clause with any SQL statement.

You can also try to force a statement to bypass the optimizer. To do so, use the FORCE NO OPTIMIZATION clause of the statement. If the statement is too complex to bypass the optimizer - possibly due to database option settings or characteristics of the schema or query - the query fails and an error is returned.

The FORCE OPTIMIZATION and FORCE NO OPTIMIZATION clauses are permitted in the OPTION clause of the following statements:

- "SELECT statement" [*SQL Anywhere Server - SQL Reference*]
- "UPDATE statement" [*SQL Anywhere Server - SQL Reference*]
- "INSERT statement" [*SQL Anywhere Server - SQL Reference*]
- "DELETE statement" [*SQL Anywhere Server - SQL Reference*]

Semantic query transformations

To operate efficiently, SQL Anywhere rewrites your queries into semantically equivalent, but syntactically different, forms. SQL Anywhere performs many different rewrite operations.

If you read the access plans, you frequently find that they do not correspond to a literal interpretation of your original statement. For example, to make your SQL statements more efficient, the optimizer tries as much as possible to rewrite subqueries with joins.

In the Query Rewrite phase, SQL Anywhere performs several transformations in search of more efficient and convenient representations of the query. Because the query may be rewritten into a semantically equivalent query, the plan may look quite different from a literal interpretation of your original query. Common manipulations include:

- eliminating of unnecessary DISTINCT conditions
- un-nesting subqueries
- performing a predicate push-down in UNION or GROUPed views and derived tables
- optimizing of OR and IN-list predicates
- optimizing of LIKE predicates
- converting outer joins to inner joins
- eliminating of outer joins and inner joins
- discovering exploitable conditions through predicate inference
- eliminating of unnecessary case translation
- rewriting subqueries as EXISTS predicates

Note
Some query rewrite optimizations cannot be performed on the main query block if the cursor is updatable. Declare the cursor as read-only to take advantage of the optimizations. See "Choosing cursor types" [*SQL Anywhere Server - Programming*], and "DECLARE CURSOR statement [ESQL] [SP]" [*SQL Anywhere Server - SQL Reference*].

For an example of an optimization that cannot be performed if the main query block is an updatable cursor, see "Elimination of unnecessary inner and outer joins" on page 534.

Some of the rewrite optimizations performed during the Query Rewrite phase can be observed in the results returned by the REWRITE function. See "REWRITE function [Miscellaneous]" [*SQL Anywhere Server - SQL Reference*].

Example

Unlike the SQL language definition, some languages mandate strict behavior for AND and OR operations. Some guarantee that the condition on the left-hand side will be evaluated first. If the truth of

the entire condition can then be determined, the compiler guarantees that the condition on the right-hand side will not be evaluated.

This arrangement lets you combine conditions that would otherwise require two nested IF statements into one. For example, in C you can test whether a pointer is NULL before you use it as follows. The nested conditions in the first statement can be replaced using the syntax shown in the second statement below:

```
if ( X != NULL ) {
   if ( X->var != 0 ) {
      ... statements ...
   }
}

if ( X != NULL && X->var != 0 ) {
      ... statements ...
}
```

Unlike C, SQL has no such rules concerning execution order. SQL Anywhere is free to rearrange the order of such conditions as it sees fit. The original and reordered forms are semantically equivalent because the SQL language specification makes no distinction between one order or another. In particular, a query optimizer is completely free to reorder predicates in a WHERE, HAVING, or ON clause.

Elimination of unnecessary DISTINCT conditions

Sometimes a DISTINCT condition is unnecessary. For example, the properties of one or more column in your result may contain a UNIQUE condition, either explicitly or implicitly, because it is a primary key.

Examples

The DISTINCT keyword in the following command is unnecessary because the Products table contains the primary key p.ID, which is part of the result set.

```
SELECT DISTINCT p.ID, p.Quantity
FROM Products p;
```

```
Products<seq>
```

The database server executes the semantically-equivalent query:

```
SELECT p.ID, p.Quantity
FROM Products p;
```

Similarly, the result of the following query contains the primary keys of both tables, so each row in the result must be distinct. So, the database server executes this query without performing DISTINCT on the result set.

```
SELECT DISTINCT *
FROM SalesOrders o JOIN Customers c
   ON o.CustomerID = c.ID
WHERE c.State = 'NY';
```

```
Work[ HF[ c<seq> ] *JH o<seq> ]
```

Un-nesting subqueries

You can express statements as nested queries, given the convenient syntax provided in the SQL language. However, rewriting nested queries as joins often leads to more efficient execution and more effective optimization, since SQL Anywhere can take better advantage of highly selective conditions in a subquery WHERE clause. In general, subquery un-nesting is always done for correlated subqueries with, at most, one table in the FROM clause, which are used in ANY, ALL, and EXISTS predicates. A uncorrelated subquery, or a subquery with more than one table in the FROM clause, is flattened if it can be decided, based on the query semantics, that the subquery returns at most one row.

Examples

The subquery in the following example can match at most one row for each row in the outer block. Because it can match at most one row, SQL Anywhere recognizes that it can convert it to an inner join.

```
SELECT s.*
FROM SalesOrderItems s
WHERE EXISTS
    ( SELECT *
      FROM Products p
      WHERE s.ProductID = p.ID
         AND p.ID = 300 AND p.Quantity > 20);
```

Following conversion, this same statement is expressed internally using join syntax:

```
SELECT s.*
FROM Products p JOIN SalesOrderItems s
   ON p.ID = s.ProductID
WHERE p.ID = 300 AND p.Quantity > 20;
```

```
p<Products> JNL s<FK_ProductID_ID>
```

Similarly, the following query contains a conjunctive EXISTS predicate in the subquery. This subquery can match more than one row.

```
SELECT p.*
FROM Products p
WHERE EXISTS
    ( SELECT *
      FROM SalesOrderItems s
      WHERE s.ProductID = p.ID
         AND s.ID = 2001);
```

SQL Anywhere converts this query to an inner join, with a DISTINCT in the SELECT-list.

```
SELECT DISTINCT p.*
FROM Products p JOIN SalesOrderItems s
   ON p.ID = s.ProductID
WHERE s.ID = 2001;
```

```
Work[ DistH[ s<FK_ID_ID> JNL p<Products> ] ]
```

SQL Anywhere can also eliminate subqueries in comparisons when the subquery matches at most one row for each row in the outer block. Such is the case in the following query.

```
SELECT *
FROM Products p
```



```
WHERE p.ID =
    ( SELECT s.ProductID
      FROM SalesOrderItems s
      WHERE s.ID = 2001
         AND s.LineID = 1 );
```

SQL Anywhere rewrites this query as follows:

```
SELECT p.*
FROM Products p, SalesOrderItems s
WHERE p.ID = s.ProductID
   AND s.ID = 2001
   AND s.LineID = 1;
```

```
s<SalesOrderItems> JNL p<Products>
```

The DUMMY table is treated as a special table when subquery un-nesting rewrite optimizations are performed. Subquery flattening is always done on subqueries of the form `SELECT` *`expression`* `FROM DUMMY`, even if the subquery is not correlated.

Predicate push-down in UNION or GROUPed views and derived tables

It is common for queries to restrict the result of a view so that only a few of the records are returned. When the view contains GROUP BY or UNION, it is preferable for the database server to only compute the result for the desired rows. Predicate push-down is performed for a predicate if, and only if, the predicate refers exclusively to the columns of a single view or derived table. A join predicate, for example, is not pushed down into the view.

Example

Suppose you have the view ProductSummary defined as follows:

```
CREATE VIEW ProductSummary( ID,
        NumberOfOrders,
        TotalQuantity) AS
SELECT ProductID, COUNT( * ), sum( Quantity )
FROM SalesOrderItems
GROUP BY ProductID;
```

For each product ordered, the ProductSummary view returns a count of the number of orders that include it, and the sum of the quantities ordered over all the orders. Now consider the following query over this view:

```
SELECT *
FROM ProductSummary
WHERE ID = 300;
```

The query restricts the output to only the row for which the value in the ID column is 300. This query, and the query in the definition of the view could be combined into the following, semantically-equivalent, SELECT statement:

```
SELECT ProductID, COUNT( * ), SUM( Quantity )
FROM SalesOrderItems
GROUP BY ProductID
HAVING ProductID = 300;
```

An unsophisticated execution plan for this query would involve computing the aggregates for each product, and then restricting the result to only the single row for product ID 300. However, the HAVING predicate on the ProductID column can be pushed into the query's WHERE clause since it is a grouping column, yielding the following:

```
SELECT ProductID, COUNT( * ), SUM( Quantity )
FROM SalesOrderItems
WHERE ProductID = 300
GROUP BY ProductID;
```

This SELECT statement significantly reduces the computation required. If this predicate is sufficiently selective, the optimizer could now use an index on ProductID to retrieve only those rows for product 300, rather than sequentially scanning the SalesOrderItems table.

The same optimization is also used for views involving UNION or UNION ALL.

Optimization of OR and IN-list predicates

The optimizer supports a special optimization for exploiting IN predicates on indexed columns. This optimization also applies equally to multiple predicates on the same indexed column that are OR'ed together, since the two are semantically equivalent. To enable the optimization, the IN-list must contain only constants, or values that are constant during one execution of the query block, such as outer references.

When the optimizer encounters a qualifying IN-list predicate, and the IN-list predicate is sufficiently selective to consider indexed retrieval, the optimizer converts the IN-list predicate into a nested loops join. The following example illustrates how the optimization works.

Suppose you have the following query, which lists all the orders for two sales reps:

```
SELECT *
FROM SalesOrders
WHERE SalesRepresentative = 902 OR SalesRepresentative = 195;
```

This query is semantically equivalent to:

```
SELECT *
FROM SalesOrders
WHERE SalesRepresentative IN (195, 902);
```

The optimizer estimates the combined selectivity of the IN-list predicate to be low enough to warrant indexed retrieval. So, the optimizer treats the IN-list as a virtual table, and joins this virtual table to the SalesOrders table on the SalesRepresentative attribute. While the net effect of the optimization is to include an additional join in the access plan, the join degree of the query is not increased, so optimization time should not be affected.

There are two main advantages of this optimization. First, the IN-list predicate can be treated as a sargable predicate and exploited for indexed retrieval. Second, the optimizer can sort the IN-list to match the sort sequence of the index, leading to more efficient retrieval.

The short form of the access plan for the above query is:

```
SalesOrders<FK_SalesRepresentative_EmployeeID>
```


See also

- "InList algorithm (IN)" on page 590

Optimization of LIKE predicates

LIKE predicates involving patterns that are either literal constants or host variables are very common. Depending on the pattern, the optimizer may rewrite the LIKE predicate entirely, or augment it with additional conditions that could be exploited to perform indexed retrieval on the corresponding table. Additional conditions for LIKE predicates utilize the LIKE_PREFIX predicate, which cannot be specified directly in a query but appear in long and graphical plans when the query optimizer can apply the optimization.

Examples

In each of the following examples, assume that the pattern in the LIKE predicate is a literal constant or host variable, and X is a column in a base table:

- `X LIKE '%'` is rewritten as `X IS NOT NULL`.
- `X LIKE 'abc%'` is augmented with a LIKE_PREFIX predicate that is a sargable predicate (it can be used for index retrieval) and enforces the condition that any value of X must begin with the characters abc. The LIKE_PREFIX predicate enforces the correct semantics with multi-byte character sets and blank-padded databases.

Conversion of outer joins to inner joins

The optimizer generates a left-deep processing tree for its access plans. The only exception to this rule is the existence of a right-deep nested outer join expression. The query execution engine's algorithms for computing LEFT or RIGHT OUTER JOINs require that preserved tables must precede null-supplying tables in any join strategy. So, the optimizer looks for opportunities to convert LEFT or RIGHT outer joins to INNER JOINs whenever possible, since INNER JOINs are commutable and give the optimizer greater degrees of freedom when performing join enumeration.

A LEFT or RIGHT OUTER JOIN is converted to an INNER JOIN if one of the following conditions is true:

- a null-intolerant predicate referencing columns of the null-supplying tables is present in the query WHERE clause. Since this predicate is null-intolerant, any all-NULL row that would be produced by the OUTER join is eliminated from the result, making the query semantically equivalent to an inner join.
- the null-supplying side of an OUTER JOIN returns exactly one row for each row from the preserved side. If this condition is true, there are no null-supplied rows and the OUTER JOIN is equivalent to an INNER join.

This rewrite optimization can apply to an outer join query when the query refers to one or more views that are written using OUTER JOINs. The query WHERE clause may include conditions that restrict the output such that all null-supplying rows from one or more table expressions would be eliminated, making this optimization applicable.

Example 1

For the query below, for each row of the SalesOrderItems table there is exactly one row that matches the Products table because the ProductID column is declared not NULL and the SalesOrderItems table has the following foreign key: `"FK_ProductID_ID" ("ProductID") REFERENCING "Products" ("ID")`.

The following SELECT statements show how the query is rewritten after a rewrite optimization:

```
SELECT * FROM SalesOrderItems s LEFT OUTER JOIN Products p ON (p.ID =
s.ProductID);
SELECT * FROM SalesOrderItems s JOIN Products p ON (p.ID = s.ProductID);
```

Example 2

The following query lists products and their corresponding orders for larger quantities; the LEFT OUTER JOIN ensures that all products are listed, even if they have no orders:

```
SELECT *
FROM Products p KEY LEFT OUTER JOIN SalesOrderItems s
WHERE s.Quantity > 15;
```

The problem with this query is that the predicate in the WHERE clause eliminates any product with no orders from the result because the predicate `s.Quantity > 15` is interpreted as FALSE if `s.Quantity` is NULL. The query is semantically equivalent to:

```
SELECT *
FROM Products p KEY JOIN SalesOrderItems s
WHERE s.Quantity > 15;
```

This rewritten form is the query that the database server optimizes.

In this example, the query is almost certainly written incorrectly; it should instead be:

```
SELECT *
FROM Products p
    KEY LEFT OUTER JOIN SalesOrderItems s
          ON s.Quantity > 15;
```

In this way, the test of Quantity is part of the outer join condition. You can demonstrate the difference in the two queries by inserting some new products into the Products table for which there are no orders and then executing the queries again.

```
INSERT INTO Products
SELECT ID + 10, Name, Description,
        'Extra large', Color, 50, UnitPrice, Photo
FROM Products
WHERE Name = 'Tee Shirt';
```

Elimination of unnecessary inner and outer joins

The join elimination rewrite optimization reduces the join degree of the query by eliminating tables from the query when it is safe to do so. Typically, this optimization is applied for inner joins defined as primary key to foreign key joins, or primary key to primary key joins. The join elimination optimization can also

be applied to tables used in outer joins, although the conditions for which the optimization is valid are much more complex.

This optimization does not eliminate tables that are updatable using UPDATE or DELETE WHERE CURRENT, even when it is correct to do so. This can negatively impact performance of the query. However, if your query is for read only, you can specify FOR READ ONLY in the SELECT statement, to ensure that the join eliminations are performed. Note that the tables appearing in subqueries or nested derived tables are inherently non-updatable, even though the tables in the main query block are updatable.

To summarize, there are three main categories of joins for which this rewrite optimization applies:

- The join is a primary key to foreign key join, and only primary key columns from the primary table are referenced in the query. In this case, the primary key table is eliminated if it is not updatable.
- The join is a primary key to primary key join between two instances of the same table. In this case, one of the tables is eliminated if it is not updatable.
- The join is an outer join and the null-supplying table expression has the following properties:
    - the null-supplying table expression returns at most one row for each row of the preserved side of the outer join
    - no expression produced by the null-supplying table expression is needed in the rest of the query beyond the outer join.

Example

For example, in the query below, the join is a primary key to foreign key join and the primary key table, Products, can be eliminated:

```
SELECT s.ID, s.LineID, p.ID
FROM SalesOrderItems s KEY JOIN Products p
FOR READ ONLY;
```

The query would be rewritten as:

```
SELECT s.ID, s.LineID, s.ProductID
FROM SalesOrderItems s
WHERE s.ProductID IS NOT NULL
FOR READ ONLY;
```

The second query is semantically equivalent to the first because any row from the SalesOrderItems table that has a NULL foreign key to Products does not appear in the result.

In the following query, the OUTER JOIN can be eliminated given that the null-supplying table expression cannot produce more than one row for any row of the preserved side and none of the columns from Products is used above the LEFT OUTER JOIN.

```
SELECT s.ID, s.LineID
FROM SalesOrderItems s LEFT OUTER JOIN Products p ON p.ID = s.ProductID
WHERE s.Quantity > 5
FOR READ ONLY;
```

The query is rewritten as:

```
SELECT s.ID, s.LineID
FROM SalesOrderItems s
WHERE s.Quantity > 5
FOR READ ONLY;
```

Discovery of exploitable conditions through predicate inference

An efficient access strategy for virtually any query relies on the presence of sargable conditions in the WHERE, ON, and HAVING clauses. Indexed retrieval is possible only by exploiting sargable conditions as matching predicates. In addition, hash, merge, and block-nested loops join can only be used when an equijoin condition is present. For these reasons, SQL Anywhere does detailed analysis of the search conditions in the original query text to discover simplified or implied conditions that can be exploited by the optimizer.

As a preprocessing step, several simplifications are made to predicates in the original statement once view expansion and merging have taken place. For example:

- `X = X` is rewritten as `X IS NOT NULL` if X is nullable; otherwise, the predicate is eliminated.
- `ISNULL(X,X)` is rewritten as `X`.
- `X+0` is rewritten as `X` if X is a numeric column.
- `AND 1=1` is eliminated.
- `OR 1=0` is eliminated.
- IN-list predicates that consist of a single element are converted to simple equality conditions.

After this preprocessing step, SQL Anywhere attempts to normalize the original search condition into conjunctive normal form (CNF). For an expression to be in CNF, each term in the expression must be AND'ed together. Each term is either made up of a single atomic condition, or a set of conditions OR'ed together.

Converting an arbitrary condition into CNF may yield an expression of similar complexity, but with a much larger set of conditions. SQL Anywhere recognizes this situation, and refrains from naively converting the condition into CNF. Instead, SQL Anywhere analyzes the original expression for exploitable predicates that are implied by the original search condition, and ANDs these inferred conditions to the query. Complete normalization is also avoided if this requires duplication of an expensive predicate (for example, a quantified subquery predicate). However, the algorithm merges IN-list predicates together whenever feasible.

Once the search condition has either been completely normalized or the exploitable conditions have been found, the optimizer performs transitivity analysis to discover transitive equality conditions, primarily transitive join conditions and conditions with a constant. In doing so, the optimizer increases its degrees of freedom when performing join enumeration during its cost-based optimization phase, since these transitive conditions may permit additional alternative join orders.

Example

Suppose the original query is as follows:

```
SELECT e.Surname, s.ID, s.OrderDate
FROM SalesOrders s, Employees e
WHERE
  ( e.EmployeeID = s.SalesRepresentative AND
    ( s.SalesRepresentative = 142 OR
      s.SalesRepresentative = 1596 )
  ) OR (
     e.EmployeeID = s.SalesRepresentative AND
     s.CustomerID = 667 );
```

This query has no conjunctive equijoin condition, and without detailed predicate analysis the optimizer would fail to discover an efficient access plan. Fortunately, SQL Anywhere is able to convert the entire expression to CNF, yielding the equivalent query:

```
SELECT e.Surname, s.ID, s.OrderDate
FROM SalesOrders s, Employees e
WHERE
   e.EmployeeID = s.SalesRepresentative AND
   ( s.SalesRepresentative = 142 OR
     s.SalesRepresentative = 1596 OR
     s.CustomerID = 667 );
```

This query can now be efficiently optimized as an inner join query.

Elimination of unnecessary case translation

By default, SQL Anywhere databases support case-insensitive string comparisons. Occasionally the optimizer may encounter queries where the user is explicitly forcing text conversion through the use of the UPPER, UCASE, LOWER, or LCASE built-in functions when such conversion is unnecessary. SQL Anywhere automatically eliminates this unnecessary conversion when the database's collation sequence permits it. An extra benefit of eliminating the case translations in the predicates is the transformation of some of these predicates into sargable predicates, which can be used for indexed retrieval of the corresponding table.

Example

Consider the following query:

```
SELECT *
FROM Customers
WHERE UPPER(Surname) = 'SMITH';
```

On a case insensitive database, this query is rewritten internally as follows, so that the optimizer can consider using an index on Customers.Surname:

```
SELECT *
FROM Customers
WHERE Surname = 'SMITH';
```

Rewriting subqueries as EXISTS predicates

The assumptions that underlie the design of SQL Anywhere require that it conserves memory and that by default it returns the first few results of a cursor as quickly as possible. In keeping with these objectives, SQL Anywhere rewrites all set-operation subqueries, such as IN, ANY, or SOME predicates, as EXISTS or NOT EXISTS predicates, if such rewriting is semantically correct. By doing so, SQL Anywhere avoids creating unnecessary work tables and may more easily identify a suitable index through which to access a table.

Uncorrelated and correlated subqueries

Uncorrelated subqueries are subqueries that contain no explicit reference to the table or tables contained in the rest of the higher-level portions of the query.

The following is an ordinary query that contains a uncorrelated subquery. It selects information about all the customers who did not place an order on January 1, 2001.

```
SELECT *
FROM Customers c
WHERE c.ID NOT IN
   (  SELECT o.CustomerID
      FROM SalesOrders o
      WHERE o.OrderDate = '2001-01-01' );
```

One possible way to evaluate this query is to create a work table of all customers in the SalesOrder table who placed orders on January 1, 2001, and then query the Customers table and extract one row for each customer listed in the work table.

However, SQL Anywhere avoids materializing results as work tables. It also gives preference to plans that return the first few rows of a result most quickly. So, the optimizer rewrites such queries using NOT EXISTS predicates. In this form, the subquery becomes **correlated**: the subquery now contains an explicit outside reference to the ID column of the Customers table.

```
SELECT *
FROM Customers c
WHERE NOT EXISTS
   (  SELECT *
      FROM SalesOrders o
      WHERE o.OrderDate = '2000-01-01'
         AND o.CustomerID = c.ID );
```

This query is semantically equivalent to the one above, but when expressed in this new syntax, several advantages become clear:

1. The optimizer can choose to use either the index on the CustomerID attribute or the OrderDate attribute of the SalesOrders table. However, in the SQL Anywhere sample database, only the ID and CustomerID columns are indexed.

2. The optimizer has the option of choosing to evaluate the subquery without materializing intermediate results as work tables.

3. The database server can cache the results of a correlated subquery during execution. This allows the re-use of previously-computed values of this predicate for the same values of the outside reference c.ID.

For the previous query, caching does not help because customer identification numbers are unique in the Customers table. So, the subquery is always computed with different values for the outside reference c.ID.

Further information about subquery caching is located in "Subquery and function caching" on page 587.

See also

- "Correlated and uncorrelated subqueries" on page 494

Inlining user-defined functions

Simple user-defined functions are sometimes inlined when called as part of a query. That is, the query is rewritten to be equivalent to the original query but without the function definition. Temporary functions, recursive functions, and functions with the NOT DETERMINISTIC clause are never inlined. Also, a function is never inlined if it is called with a subquery as an argument, or when it is called from inside a temporary procedure.

User-defined functions can be inlined if they take one of the following forms:

- A function with a single RETURN statement. For example:

```
CREATE FUNCTION F1( arg1 INT, arg2 INT )
RETURNS INT
BEGIN
 RETURN arg1 * arg2
END;
```

- A function that declares a single variable, assigns the variable, and returns a single value. For example:

```
CREATE FUNCTION F2( arg1 INT )
RETURNS INT
BEGIN
 DECLARE result INT;
 SET result = ( SELECT ManagerID FROM Employees WHERE EmployeeID=arg1 );
 RETURN result;
END;
```

- A function that declares a single variable, selects into that variable, and returns a single value. For example:

```
CREATE FUNCTION F3( arg1 INT )
RETURNS INT
BEGIN
 DECLARE result INT;
 SELECT ManagerID INTO result FROM Employees e1 WHERE EmployeeID=arg1;
 RETURN result;
END;
```

A user-defined function is inlined by copying the body of the user-defined function, inserting the arguments from the call, and inserting appropriate CAST functions to ensure that the rewritten form of the query is equivalent to the original. For example, suppose you created a function similar to the function F1 defined previously, and then you call the procedure in a FROM clause of a query as follows:

```
SELECT F1( e.EmployeeID, 2.5 ) FROM Employees e;
```

The database server may rewrite the query as follows:

```
SELECT CAST( e.EmployeeID AS INT ) * CAST( 2.5 AS INT ) FROM Employees e;
```

See also

- "Introduction to user-defined functions" on page 799
- "CREATE FUNCTION statement" [*SQL Anywhere Server - SQL Reference*]
- "CAST function [Data type conversion]" [*SQL Anywhere Server - SQL Reference*]
- "SELECT statement" [*SQL Anywhere Server - SQL Reference*]

Inlining simple system procedures

A system procedure defined only as a single SELECT statement in the body is sometimes inlined when called in the FROM clause of a query. That is, the query is rewritten to be equivalent to the original query but without the procedure definition. When a procedure is inlined, it is rewritten as a derived table. A procedure is never inlined if it uses default arguments, or if it contains anything other than a single SELECT statement in the body.

For example, suppose you create the following procedure:

```
CREATE PROCEDURE Test1( arg1 INT )
 BEGIN
  SELECT * FROM Employees WHERE EmployeeID=arg1
 END;
```

Now suppose you call the procedure in a FROM clause of a query as follows:

```
SELECT * FROM Test1( 200 );
```

The database server may rewrite the query as follows:

```
SELECT * FROM ( SELECT * FROM Employees WHERE EmployeeID=CAST( 200 AS INT ) )
AS Test1;
```

See also

- "Introduction to procedures" on page 794
- "CREATE PROCEDURE statement" [*SQL Anywhere Server - SQL Reference*]
- "SELECT statement" [*SQL Anywhere Server - SQL Reference*]

How the optimizer works

The role of the optimizer is to devise an efficient way to execute SQL statements. To do this, the optimizer must determine an execution plan for a query. This includes decisions about the access order for tables referenced in the query, the join operators and access methods used for each table, and whether materialized views that are not referenced in the query can be used to compute parts of the query. The optimizer attempts to pick the best plan for executing the query during the join enumeration phase, when

possible access plans for a query are generated and costed. The best access plan is the one that the optimizer estimates will return the desired result set in the shortest period of time, with the least cost. The optimizer determines the cost of each enumerated strategy by estimating the number of disk reads and writes required.

In Interactive SQL, you can view the best access plan used to execute a query by clicking **Tools** » **Plan Viewer**. See "Reading graphical plans" on page 595, and "Reading execution plans" on page 592.

Minimizing the cost of returning the first row

The optimizer uses a generic disk access cost model to differentiate the relative performance differences between random and sequential retrieval on the database file. It is possible to calibrate a database for a particular hardware configuration using an ALTER DATABASE statement. See "ALTER DATABASE statement" [*SQL Anywhere Server - SQL Reference*].

By default, query processing is optimized towards returning the complete result set. You can change the default behavior using the optimization_goal option, to minimize the cost of returning the first row quickly. Note that when the option is set to First-row, the optimizer favors an access plan that is intended to reduce the time to fetch the first row of the query's result, likely at the expense of total retrieval time. See "optimization_goal option" [*SQL Anywhere Server - Database Administration*].

Using semantically equivalent syntax

Most commands can be expressed in many different ways using the SQL language. These expressions are semantically equivalent in that they do the same task, but may differ substantially in syntax. With few exceptions, the optimizer devises a suitable access plan based only on the semantics of each statement.

Syntactic differences, although they may appear to be substantial, usually have no effect. For example, differences in the order of predicates, tables, and attributes in the query syntax have no effect on the choice of access plan. Neither is the optimizer affected by whether a query contains a non-materialized view.

Reducing the cost of optimizing queries

Ideally, the optimizer would identify the most efficient access plan possible, but this goal is often impractical. Given a complicated query, a great number of possibilities may exist.

However efficient the optimizer, analyzing each option takes time and resources. The optimizer compares the cost of further optimization with the cost of executing the best plan it has found so far. If a plan has been devised that has a relatively low cost, the optimizer stops and allows execution of that plan to proceed. Further optimization might consume more resources than would execution of an access plan already found. You can control the amount of effort made by the optimizer by setting a high value for the optimization_level option. See "optimization_level option" [*SQL Anywhere Server - Database Administration*].

The optimizer works longer for expensive and complex queries, or when the optimization level is set high. For very expensive queries, it may run long enough to cause a discernible delay.

Optimizer estimates and column statistics

The optimizer chooses a strategy for processing a statement based on **column statistics** stored in the database and on **heuristics** (educated guesses). For each access plan considered by the optimizer, an estimated result size (number of rows) must be computed. For example, for each join method or index access based on the selectivity estimations of the predicates used in the query, an estimated result size is calculated. The estimated result sizes are used to compute the estimated disk access and CPU cost for each operator such as a join method, a group by method, or a sequential scan, used in the plan. Column statistics are the primary data used by the optimizer to compute selectivity estimation of predicates. Therefore, they are vital to estimating correctly the cost of an access plan.

If column statistics become stale, or are missing, performance can degrade since inaccurate statistics may result in an inefficient execution plan. If you suspect that poor performance is due to inaccurate column statistics, you should recreate them. See "Updating column statistics to improve optimizer performance" on page 545.

Selectivity estimate sources

For any predicate, the optimizer can use any of the following source for selectivity estimates. The chosen source is indicated in the graphical and long plan for the query.

- **Statistics** The optimizer can use stored column statistics to calculate selectivity estimates. If constants are used in the predicate, the stored statistics are available only when the selectivity of a constant is a significant enough number that it is stored in the statistics.

  For example, the predicate `EmployeeID > 100` can use column statistics as the selectivity estimate source if the statistics for the EmployeeID column exists.

- **Join** The optimizer can use referential integrity constraints, unique constraints, or join histograms to compute selectivity estimates. Join histograms are computed for a predicate of the form `T.X=R.X` from the available statistics of the T.X and R.X columns.

- **Column-column** In the case of a join where there are no referential integrity constraints, unique constraints, or join histograms available to use as selectivity sources, the optimizer can use, as a selectivity source, the estimated number of rows in the joined result set divided by the number of rows in the Cartesian product of the two tables.

- **Column** The optimizer can use the average of all values that have been stored in the column statistics.

  For example, the selectivity of the predicate `DepartmentName = expression` can be computed using the average if *expression* is not a constant.

- **Index** The optimizer can probe indexes to compute selectivity estimates. In general, an index is used for selectivity estimates if no other sources of selectivity estimates, for example column statistics, can be used.

  For example, for the predicate `DepartmentName = 'Sales'`, the optimizer can use an index defined on the column DepartmentName to estimate the number of rows having the value Sales.

- **User** The optimizer can use user-supplied selectivity estimates, provided the user_estimates database option is not set to Disabled. See "user_estimates option" [*SQL Anywhere Server - Database Administration*].
- **Guess** The optimizer can resort to best guessing to calculate selectivity estimates when there is no relevant index to use, no statistics have been collected for the referenced columns, or the predicate is a complex predicate. In this case, built-in guesses are defined for each type of predicate.
- **Computed** For example, a very complex predicate may have the selectivity estimate set to 100% and the selectivity source set to Computed if the selectivity estimate was computed, for example, by multiplying or adding the selectivities.
- **Always** If a predicate is always true, the selectivity source is 'Always'. For example, the predicate `1=1` is always true.
- **Combined** If the selectivity estimate is computed by combining more than one of the sources above, the selectivity source is 'Combined'.
- **Bounded** When SQL Anywhere has placed an upper and/or lower bound on the selectivity estimate, the selectivity source is 'Bounded'. For example, bounds are sets to ensure that an estimate is not greater than 100%, or that the selectivity is not less than 0%.

See also

- "ESTIMATE function [Miscellaneous]" [*SQL Anywhere Server - SQL Reference*]
- "ESTIMATE_SOURCE function [Miscellaneous]" [*SQL Anywhere Server - SQL Reference*]
- "Viewing selectivity in the graphical plan" on page 601
- "Supply explicit selectivity estimates sparingly" on page 222
- "sa_get_histogram system procedure" [*SQL Anywhere Server - SQL Reference*]
- "INDEX_ESTIMATE function [Miscellaneous]" [*SQL Anywhere Server - SQL Reference*]
- "EXPERIENCE_ESTIMATE function [Miscellaneous]" [*SQL Anywhere Server - SQL Reference*]
- "Supply explicit selectivity estimates sparingly" on page 222
- "user_estimates option" [*SQL Anywhere Server - Database Administration*]

How the optimizer uses column statistics

The most important component of the column statistics used by the optimizer are **histograms**. Histograms store information about the distribution of values in a column. In SQL Anywhere, a histogram represents the data distribution for a column by dividing the domain of the column into a set of consecutive value ranges (also called **buckets**) and by remembering, for each value range (or bucket), the number of rows in the table for which the column value falls in the bucket.

SQL Anywhere pays particular attention to single column values that are present in a large number of rows in the table. Significant single value selectivities are maintained in singleton histogram buckets (for example, buckets that encompass a single value in the column domain). SQL Anywhere tries to maintain a minimum number of singleton buckets in each histogram, usually between 10 and 100 depending upon the size of the table. Additionally, all single values with selectivities greater than 1% are kept as singleton buckets. As a result, a histogram for a given column remembers the top *N* single value selectivities for the

column where the value of *N* is dependent upon the size of the table and the number of single value selectivities that are greater than 1%.

Once the minimum number of value ranges has been met, low-selectivity frequencies are replaced by large-selectivity frequencies as they come along. The histogram will only have more than the minimum number of singleton value ranges after it has seen enough values with a selectivity of greater than 1%.

For more information about column statistics, see "SYSCOLSTAT system view" [*SQL Anywhere Server - SQL Reference*].

How the optimizer uses heuristics

For each table in a potential execution plan, the optimizer estimates the number of rows that will form part of the results. The number of rows depends on the size of the table and the restrictions in the WHERE clause or the ON clause of the query.

Given the histogram on a column, SQL Anywhere estimates the number of rows satisfying a given query predicate on the column by adding up the number of rows in all value ranges that overlap the values satisfying the specified predicate. For value ranges in the histograms that are partially contained in the query result set, SQL Anywhere uses interpolation within the value range.

Often, the optimizer uses more sophisticated heuristics. For example, the optimizer only uses default estimates when better statistics are unavailable. As well, the optimizer makes use of indexes and keys to improve its guess of the number of rows. The following are a few single-column examples:

- Equating a column to a value: estimate one row when the column has a unique index or is the primary key.
- A comparison of an indexed column to a constant: probe the index to estimate the percentage of rows that satisfy the comparison.
- Equating a foreign key to a primary key (key join): use relative table sizes in determining an estimate. For example, if a 5000 row table has a foreign key to a 1000 row table, the optimizer guesses that there are five foreign key rows for each primary key row.

See also

For information about the distribution of column values, see:

- "Selectivity estimate sources" on page 542
- "ESTIMATE function [Miscellaneous]" [*SQL Anywhere Server - SQL Reference*]
- "ESTIMATE_SOURCE function [Miscellaneous]" [*SQL Anywhere Server - SQL Reference*]

How the optimizer uses procedure statistics

Unlike base tables, procedure calls executed in the FROM clause do not have column statistics. Therefore, the optimizer uses defaults or guesses for all selectivity estimates on data coming from a procedure call.

The execution time of a procedure call, and the total number of rows in its result set, are estimated using statistics collected from previous calls. These statistics are maintained in the stats column of the ISYSPROCEDURE system table by the ProCall algorithm. See "SYSPROCEDURE system view" [*SQL Anywhere Server - SQL Reference*], and "ProcCall algorithm (PC)" on page 591.

See also

For information about obtaining the selectivities of predicates, see:

- "sa_get_histogram system procedure" [*SQL Anywhere Server - SQL Reference*]
- "Histogram utility (dbhist)" [*SQL Anywhere Server - Database Administration*]

Updating column statistics to improve optimizer performance

Column statistics are stored permanently in the database in the ISYSCOLSTAT system table. To continually improve the optimizer's performance, the database server automatically updates column statistics during the processing of any SELECT, INSERT, UPDATE, or DELETE statement, including statements on a single row. It does so by monitoring the number of rows that satisfy any predicate that references a table or column, comparing that number to the number of rows estimated, and then, if necessary, updating existing statistics.

With more accurate column statistics available to it, the optimizer can compute better estimates and improve the performance of subsequent queries.

You can set whether to update column statistics using database options. The update_statistics database option controls whether to update column statistics during execution of queries, while the collect_statistics_on_dml_updates database option controls whether to update the statistics during the execution of data-altering DML statements such as LOAD, INSERT, DELETE, and UPDATE.

If you suspect that performance is suffering because your statistics inaccurately reflect the current column values, you may want to execute the statements CREATE STATISTICS or DROP STATISTICS. CREATE STATISTICS deletes old statistics and creates new ones, while DROP STATISTICS only deletes old statistics.

When you execute the CREATE INDEX statement, statistics are automatically created for the index.

When you execute the LOAD TABLE statement, statistics are automatically created for the table.

See also

- "SYSCOLSTAT system view" [*SQL Anywhere Server - SQL Reference*]
- "CREATE STATISTICS statement" [*SQL Anywhere Server - SQL Reference*]
- "ALTER STATISTICS statement" [*SQL Anywhere Server - SQL Reference*]
- "DROP STATISTICS statement" [*SQL Anywhere Server - SQL Reference*]
- "update_statistics option" [*SQL Anywhere Server - Database Administration*]
- "collect_statistics_on_dml_updates option" [*SQL Anywhere Server - Database Administration*]

How the statistics governor maintains statistics

In addition to the automatic adjustment of column statistics that is performed when a query is executed, the statistics governor also monitors the health and usage of optimizer statistics. The statistics governor automatically evaluates the health and usefulness of each statistic in the database and performs required maintenance so that the statistics are self-monitored and self-healing. Statistics maintenance is performed in the background and does not create a significant load on database server performance.

The statistics governor performs the following tasks:

- Records statistics usage and estimation errors from query feedback
- Fixes or remakes statistics that have low accuracy
- Stops automatic maintenance for statistics that cannot be maintained efficiently
- Creates potentially useful statistics
- Drops unused statistics

The update_statistics option controls whether the specified connection can send query feedback to the statistics governor. If this option is set to Off, the statistics governor does not receive query feedback from the specified connection. However, the statistics governor can still receive query feedback from other connections and perform maintenance operations on statistics. See "update_statistics option" [*SQL Anywhere Server - Database Administration*].

The statistics governor decides when to fix or create a statistics based on its health and usage. A statistic can be fixed or created either by gathering statistics during query execution, or by a separate process called the statistics cleaner. You can disable the statistics cleaner by using the StatisticsCleaner option for the sa_server_option system procedure without disabling the statistics governor, but when the statistics cleaner is turned off, statistics are only created or fixed when a query is run.

To reduce server workload, the statistics governor stops maintenance on statistics that are hard to fix or never used. Statistics that have been fixed numerous times within a short period of time and still return poor estimates are dropped and are not maintained for 30 days. Dropped statistics are recreated after 30 days, and regular maintenance is resumed. You can disable this feature using the DropBadStatistics option for the sa_server_option system procedure. Statistics that have not been used in the last 90 days are also dropped. To disable this feature, use the DropUnusedStatistics option for the sa_server_option system procedure. You can resume maintenance on a statistic at any time by using the CREATE STATISTICS, DROP STATISTICS, or ALTER STATISTICS statements.

Statistics are only monitored for tables that are loaded into memory, and these statistics are flushed every 30 minutes. During flushing, the health and usage of the statistics are checked, and the statistics governor performs maintenance on the statistics. The state information about a statistic (such as health, usage, and information about when to update or drop a statistic) does not persist between sessions. The state information is reset when the database server shuts down.

See also

- "CREATE STATISTICS statement" [*SQL Anywhere Server - SQL Reference*]
- "ALTER STATISTICS statement" [*SQL Anywhere Server - SQL Reference*]
- "DROP STATISTICS statement" [*SQL Anywhere Server - SQL Reference*]
- "update_statistics option" [*SQL Anywhere Server - Database Administration*]
- "sa_server_option system procedure" [*SQL Anywhere Server - SQL Reference*]

Automatic performance tuning

One of the most common constraints in a query is equality with a column value. The following example tests for equality of the Sex column.

```
SELECT *
FROM Employees
WHERE Sex = 'f';
```

Queries often optimize differently at the second execution. For the above type of constraint, SQL Anywhere learns from experience, automatically allowing for columns that have an unusual distribution of values. The database stores this information permanently unless you explicitly delete it using the DROP STATISTICS command. Note that subsequent queries with predicates over that column may cause the database server to recreate a histogram on the column. See "Updating column statistics to improve optimizer performance" on page 545.

Underlying assumptions of the optimizer

Several assumptions underlie the design direction and philosophy of the SQL Anywhere query optimizer. You can improve the quality or performance of your own applications through an understanding of the optimizer's decisions. These assumptions provide a context in which you may understand the information contained in the remaining sections.

Minimal administration work

Traditionally, high performance database servers have relied heavily on the presence of a knowledgeable, dedicated, database administrator. This person spent a great deal of time adjusting data storage and performance controls of all kinds to achieve good database performance. These controls often required continuing adjustment as the data in the database changed.

SQL Anywhere learns and adjusts as the database grows and changes. Each query betters its knowledge of the data distribution in the database. SQL Anywhere automatically stores and uses this information to optimize future queries.

Every query both contributes to this internal knowledge and benefits from it. Every user can benefit from knowledge that SQL Anywhere has gained through executing another user's query.

Statistics-gathering mechanisms are an integral part of the database server, and require no external mechanism. Should you find an occasion where it would help, you can provide the database server with index hints. These hints ensure that certain indexes are used during optimization, thereby overriding the decisions made by the optimizer based on selectivity estimations. If you encode these into a trigger or procedure, you then assume responsibility for updating the hints whenever appropriate. See "Updating column statistics to improve optimizer performance" on page 545, and "Working with indexes" on page 56.

Optimize for first row or for entire result set

The optimization_goal option allows you to specify whether query processing should be optimized towards returning the first row quickly, or towards minimizing the cost of returning the complete result set (the default behavior). See "optimization_goal option" [*SQL Anywhere Server - Database Administration*].

Optimize for mixed or OLAP workload

The optimization_workload option allows you to specify whether query processing should be optimized towards databases where updates, deletes, or inserts are commonly executed concurrently with queries (mixed workload) or whether the main form of update activity in the database is batch-style updates that are rarely executed concurrently with query execution.

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

Statistics are present and correct

The optimizer is self-tuning, storing all the needed information internally. The ISYSCOLSTAT system table is a persistent repository of data distributions and predicate selectivity estimates. At the completion of each query, SQL Anywhere uses statistics gathered during query execution to update ISYSCOLSTAT. As a result, all subsequent queries gain access to more accurate estimates.

The optimizer relies heavily on these statistics and, therefore, the quality of the access plans it generates depends heavily on them. If you recently inserted a lot of new rows, these statistics may no longer accurately describe the data. You may find that your subsequent queries execute unusually slowly.

If you have significantly altered your data, and you find that query execution is slow, you may want to execute DROP STATISTICS and/or CREATE STATISTICS. See "Updating column statistics to improve optimizer performance" on page 545.

Indexes can be used to satisfy a predicate

Often, SQL Anywhere can evaluate search conditions with the aid of indexes. Using indexes speeds optimizer access to data and reduces the amount of information read and processed from base tables. For

example, if a query contains a search condition `WHERE column-name=value`, and an index exists on the column, an index scan can be used to read only those rows of the table that satisfy the search condition.

Indexes also improve performance dramatically when joining tables.

Whenever possible, the optimizer attempts index-only retrieval to satisfy a query. With index-only retrieval, the database server uses only the data in the indexes to satisfy the query, and does not need to access rows in the table.

In the case where there are no indexes for the optimizer to use, a sequential table scan is performed instead, which can be expensive.

The optimizer automatically chooses to use the indexes it determines will lead to the best performance. However, you can also use index hints in your query to specify the indexes you want the optimizer to use. If any of the specified indexes cannot be used, an error is returned. Note that index hinting can result in poor performance and should only be attempted by experienced users. See "FROM clause" [*SQL Anywhere Server - SQL Reference*].

Use the Index Consultant to determine whether additional indexes are recommended for your database. See "Index Consultant" on page 164.

See also

- "Using predicates in queries" on page 551

Virtual memory is a scarce resource

The operating system and several applications frequently share the memory of a computer. SQL Anywhere treats memory as a scarce resource. Because it uses memory economically, SQL Anywhere can run on relatively small computers. This economy is important if you want your database to operate on portable computers or on older computers.

Reserving extra memory, for example to hold the contents of a cursor, may be expensive. If the buffer cache is full, one or more pages may have to be written to disk to make room for new pages. Some pages may need to be re-read to complete a subsequent operation.

In recognition of this situation, SQL Anywhere associates a higher cost with execution plans that require additional buffer cache overhead. This cost discourages the optimizer from choosing plans that use work tables.

On the other hand, the optimizer is careful to use memory where it improves performance. For example, it caches the results of subqueries when they will be needed repeatedly during the processing of the query.

The memory governor

The SQL Anywhere database server utilizes the cache, also called the **buffer pool**, to temporarily store (buffer) images of database pages in memory. These pages are typically table pages and index pages,

although there are several other types of physical pages stored in a SQL Anywhere database. In addition to these pages, the database server utilizes the cache for two other pools of memory. One of these pools is the virtual memory used for database server data structures, such as those that represent connections, statements, and cursors. The second pool consists of cache pages that are used as virtual storage for **query memory**.

Query execution algorithms, such as hash join and sorting, require memory to operate efficiently. SQL Anywhere uses a **memory governor** to decide how much query memory each statement can use for query execution. The memory governor is responsible for allocating a pool of query memory to statements to give efficient execution of the workload. The QueryMemPages database server property shows the number of pages in the query memory pool that are available for distribution. The pool size is set to be a proportion of the *maximum* cache size for the server; that is, the cache size's upper bound, which can be controlled by the -ch server option. The QueryMemPercentOfCache database server property gives the proportion of maximum cache size that can be query memory, which is 50%.

The memory governor grants individual statements a selected number of pages that the statement can then use for memory-intensive query processing algorithms. Memory in the query memory pool is still available for other purposes (such as buffering table or index pages) until the query processing algorithm uses the pages. Memory-intensive query processing algorithms that use query memory include all hash-based operators, such as hash distinct, hash group by, and hash join, and sorting and window operators.

When a statement begins executing, the memory governor uses the optimizer's estimates to determine how much memory would be useful to the statement. This estimate appears in the graphical plan as QueryMemMaxUseful. Query memory for the statement is allocated across the particular memory-intensive operators used in the access plan for that request. Parallel memory-intensive operators beneath an Exchange operator each receive their own allocation of query memory. Simple requests do not benefit from large amounts of memory, but requests that use hash-based operators or sorting can operate more efficiently if there is enough memory to hold all the needed rows in memory.

Increasing the database server multiprogramming level requires the database server to reserve some amount of query memory for each additional concurrent task, or request, reducing the amount available to any particular request. Also, the memory governor limits the number of memory-intensive requests that can execute concurrently. This maximum value is selected based on the performance characteristics of the computer running the database server, and the limit is shown with the server property QueryMemActiveMax. The memory governor also maintains a running estimate of the number of concurrent memory intensive requests, and this estimate is available as the database server property and Performance Monitor statistic QueryMemActiveEst. The memory governor uses this running average to decide how to assign memory from the query memory pool. If few memory-intensive requests have been executing, then more memory is assigned to each one. If many have been executing, each one is assigned less to share the query memory more evenly, taking into account the estimated number of query memory pages useful to each request.

If a memory-intensive statement begins executing and there are already the maximum number of concurrent memory-intensive requests executing, then incoming statements wait for one of the existing requests to release its allocated memory. The query_mem_timeout database option controls how long the incoming request waits for a memory grant. With the default setting of -1, the request waits for a database server-defined period of time. If no memory grant is available after waiting, then the statement's access plan is executed with a small amount of memory, which could lead it to perform slowly, possibly with a low-memory execution strategy if one exists for memory-intensive physical operators in that plan. The

database server property and Performance Monitor statistic QueryMemGrantWaiting shows the current number of requests that are waiting for a memory request to be granted, and QueryMemGrantWaited shows the total number of times that a request had to wait before a memory request was granted.

In the graphical plan, the value QueryMemNeedsGrant shows whether the memory governor considers this to be a simple request (no memory grant needed) or memory intensive (a memory grant is needed). If the memory governor classifies a request as not needing a memory grant, then the request begins executing immediately. Otherwise, the request asks to use a proportion of the query memory pool. The graphical plan value QueryMemLikelyGrant shows an estimate of how many pages are likely to be granted to the request for execution.

See also

- "QueryMemActiveMax server property" [*SQL Anywhere Server - Database Administration*]
- "QueryMemPages server property" [*SQL Anywhere Server - Database Administration*]
- "QueryMemPercentOfCache server property" [*SQL Anywhere Server - Database Administration*]
- "query_mem_timeout option" [*SQL Anywhere Server - Database Administration*]
- "Configuring the database server's multiprogramming level" [*SQL Anywhere Server - Database Administration*]
- "Graphical plan with statistics" on page 596
- "-ch dbeng12/dbsrv12 server option" [*SQL Anywhere Server - Database Administration*]

Using predicates in queries

A **predicate** is a conditional expression that, combined with the logical operators AND and OR, makes up the set of conditions in a WHERE, HAVING, or ON clause. In SQL, a predicate that evaluates to UNKNOWN is interpreted as FALSE.

A predicate that can exploit an index to retrieve rows from a table is called **sargable**. This name comes from the phrase *search argument-able*. Predicates that involve comparisons of a column with constants, other columns, or expressions may be sargable.

The predicate in the following statement is sargable. SQL Anywhere can evaluate it efficiently using the primary index of the Employees table.

```
SELECT *
FROM Employees
WHERE Employees.EmployeeID = 102;
```

In the plan, this appears as: `Employees<Employees>`.

In contrast, the following predicate is not sargable. Although the EmployeeID column is indexed in the primary index, using this index does not expedite the computation because the result contains all, or all except one, row.

```
SELECT *
FROM Employees
where Employees.EmployeeID <> 102;
```

In the plan, this appears as: `Employees<seq>`.

Similarly, no index can assist in a search for all employees whose given name ends in the letter k. Again, the only means of computing this result is to examine each of the rows individually.

Functions

In general, a predicate that has a function on the column name is not sargable. For example, an index would not be used on the following query:

```
SELECT *
FROM SalesOrders
WHERE YEAR ( OrderDate ) ='2000';
```

To avoid using a function, you can rewrite a query to make it sargable. For example, you can rephrase the above query:

```
SELECT *
FROM SalesOrders
WHERE OrderDate > '1999-12-31'
AND OrderDate < '2001-01-01';
```

A query that uses a function becomes sargable if you store the function values in a computed column and build an index on this column. A **computed column** is a column whose values are obtained from other columns in the table. For example, if you have a column called OrderDate that holds the date of an order, you can create a computed column called OrderYear that holds the values for the year extracted from the OrderDate column.

```
ALTER TABLE SalesOrders
ADD OrderYear INTEGER
COMPUTE ( YEAR( OrderDate ) );
```

You can then add an index on the column OrderYear in the ordinary way:

```
CREATE INDEX IDX_year
ON SalesOrders ( OrderYear );
```

If you then execute the following statement, the database server recognizes that there is an indexed column that holds that information and uses that index to answer the query.

```
SELECT * FROM SalesOrders
WHERE YEAR( OrderDate ) = '2000';
```

The domain of the computed column must be equivalent to the domain of the COMPUTE expression in order for the column substitution to be made. In the above example, if `YEAR( OrderDate )` had returned a string instead of an integer, the optimizer would not have substituted the computed column for the expression, and the index IDX_year could not have been used to retrieve the required rows.

For more information about computed columns, see "Working with computed columns" on page 15.

Examples

In each of these examples, attributes *x* and *y* are each columns of a single table. Attribute *z* is contained in a separate table. Assume that an index exists for each of these attributes.

Sargable	Non-sargable
x = 10	*x* < > 10
x IS NULL	*x* IS NOT NULL
x > 25	*x* = 4 OR *y* = 5
x = *z*	*x* = *y*
x IN (4, 5, 6)	*x* NOT IN (4, 5, 6)
x LIKE 'pat%'	*x* LIKE '%tern'
x = 20 - 2	*x* + 2 = 20
X IS NOT DISTINCT FROM *Y*+1	
X IS DISTINCT FROM *Y*+1	

Sometimes it may not be obvious whether a predicate is sargable. In these cases, you may be able to rewrite the predicate so it is sargable. For each example, you could rewrite the predicate *x* LIKE 'pat%' using the fact that u is the next letter in the alphabet after t: *x* >= 'pat' and *x* < 'pau'. In this form, an index on attribute *x* is helpful in locating values in the restricted range. Fortunately, SQL Anywhere makes this particular transformation for you automatically.

A sargable predicate used for indexed retrieval on a table is a **matching** predicate. A WHERE clause can have many matching predicates. The most suitable predicate can depend on the join strategy. The optimizer re-evaluates its choice of matching predicates when considering alternate join strategies. See "Discovery of exploitable conditions through predicate inference" on page 536.

Cost-based optimization with MIN and MAX functions

The min/max cost-based optimization is designed to exploit an existing index to compute efficiently the result of a simple aggregation query involving the MAX or MIN aggregate functions. The goal of this optimization is to be able to compute the result by retrieving only a few rows from the index. To be a candidate for this optimization, the query:

- must not contain a GROUP BY clause
- must be over a single table
- must contain only a single aggregate function (MAX or MIN) in the query's SELECT-list

Example

To illustrate this optimization, assume that an index called product_quantity (ShipDate ASC, Quantity ASC) exists on the SalesOrderItems table. Then the query

```
SELECT MIN( Quantity )
 FROM SalesOrderItems
 WHERE ShipDate = '2000-03-25';
```

is rewritten internally as

```
SELECT MAX( Quantity )
 FROM ( SELECT FIRST Quantity
        FROM SalesOrderItems
        WHERE ShipDate = '2000-03-25'
            AND Quantity IS NOT NULL
 ORDER BY ShipDate ASC, Quantity ASC ) AS s(Quantity);
```

The NULL_VALUE_ELIMINATED warning may not be generated for aggregate queries when this optimization is applied.

The execution plan (short form) for the rewritten query is:

```
GrByS[ RL[ SalesOrderItems<product_quantity> ] ]
```

Plan caching

Normally, the optimizer selects an execution plan for a query every time the query is executed. Optimizing at execution time allows the optimizer to choose a plan based on current system state, and the values of current selectivity estimates and estimates based on the values of host variables. For queries that are executed frequently, the cost of query optimization can outweigh the benefits of optimizing at execution time. To reduce the cost of optimizing these statements repeatedly, the SQL Anywhere server considers caching plans for:

- All statements performed inside stored procedures, user-defined functions, and triggers.
- SELECT, INSERT, UPDATE, or DELETE statements that qualify for bypass optimization. See "Query processing phases" on page 525.

  For INSERT statements, only INSERT...VALUES statements qualify for caching; INSERT...ON EXISTING statements do not qualify for caching.

  For UPDATE and DELETE statements, the WHERE clause must be present and contain search conditions that use the primary key to identify a row. No extra search conditions are allowed if plan caching is desired. Also, for UPDATE statements, a SET clause that contains a variable assignment disqualifies the statement from caching.

After one of these statements has been executed several times by a connection, the optimizer builds a reusable plan for the statement without knowing the host variable values. The reusable plan may have a higher cost because host variable values cannot be used for selectivity estimation or semantic query transformations. If the reusable plan has the same structure as the plans built in previous executions of the statement, the database server adds the reusable plan to the plan cache. The execution plan is not cached when the benefit of optimizing on each execution outweighs the savings from avoiding optimization.

If an execution plan uses a materialized view that was not referenced by the statement, and the materialized_view_optimization option is set to something other than Stale, then the execution plan is not

cached and the statement is optimized again the next time the stored procedure, user-defined function, or trigger is called.

The plan cache is a per-connection cache of the data structures used to execute an access plan. Reusing the cached plan involves looking up the plan in the cache and resetting it to an initial state. Typically, this is substantially faster than processing the statement through all of the query processing phases. Cached plans may be stored to disk if they are used infrequently, and they do not increase the cache usage. The optimizer periodically re-optimizes queries to verify that the cached plan is still efficient.

The maximum number of plans to cache is specified with the max_plans_cached option. The default is 20. To disable plan caching, set this option to 0. See "max_plans_cached option" [*SQL Anywhere Server - Database Administration*].

You can use the QueryCachedPlans statistic to show how many query execution plans are currently cached. This property can be retrieved using the CONNECTION_PROPERTY function to show how many query execution plans are cached for a given connection, or the DB_PROPERTY function can be used to count the number of cached execution plans across all connections. This property can be used in combination with QueryCachePages, QueryOptimized, QueryBypassed, and QueryReused to help determine the best setting for the max_plans_cached option. See "Connection properties" [*SQL Anywhere Server - Database Administration*].

You can use the database or QueryCachePages connection property to determine the number of pages used to cache execution plans. These pages occupy space in the temporary file, but are not necessarily resident in memory.

See also

- "Eligibility to skip query processing phases" on page 526
- "Improving performance with materialized views" on page 555
- "materialized_view_optimization option" [*SQL Anywhere Server - Database Administration*]
- "DB_PROPERTY function [System]" [*SQL Anywhere Server - SQL Reference*]
- "CONNECTION_PROPERTY function [System]" [*SQL Anywhere Server - SQL Reference*]

Improving performance with materialized views

A materialized view is a view whose result set is stored on disk, much like a base table, but that is computed, much like a view. Conceptually, a materialized view is both a view (it has a query specification) and a table (it has persistent materialized rows). So, many operations that you perform on tables can be performed on materialized views as well. For example, you can build indexes on, and unload from, materialized views.

Defining materialized views

In designing your application, consider defining materialized views for frequently-executed expensive queries or expensive parts of your queries, such as those involving intensive aggregation and join operations. Materialized views are designed to improve performance in environments where:

- the database is large
- frequent queries result in repetitive aggregation and join operations on large amounts of data
- changes to underlying data are relatively infrequent
- access to up-to-the-moment data is not a critical requirement

You do not have to change your queries to benefit from materialized views. For example, materialized views are ideal for use with data warehousing applications where the underlying data doesn't change very often.

The optimizer maintains a list of materialized views to consider as candidates for partially or fully satisfying a submitted query when optimizing. If the optimizer finds a candidate materialized view that can satisfy all or part of the query, it includes the view in the recommendations it makes for the enumeration phase of optimization, where the best plan is determined based on cost. The process used by the optimizer to match materialized views to queries is called **view matching**. Before a materialized view can be considered by the optimizer, the view must satisfy certain conditions. This means that unless a materialized view is explicitly referenced by the query, there is no guarantee that it will be used by the optimizer. You can, however, make sure that the conditions are met for the view to be considered.

If the optimizer determines that materialized view usage is allowed, then each candidate materialized view is examined. A materialized view is considered for use by the View Matching algorithm if:

- the materialized view is enabled for use by the database server. See "Enable and disable materialized views" on page 51.
- the materialized view is enabled for use in optimization. See "Enable and disable optimizer use of a materialized view" on page 52.
- the materialized view has been initialized. See "Initialize materialized views" on page 44.
- the materialized view meets all the optimizer requirements for consideration. See "Materialized views and the View Matching algorithm" on page 557.
- the values of some critical options used to create the materialized views match the options for the connection executing the query. See "Restrictions on materialized views" on page 39.
- the last refresh of the materialized view does not exceed the staleness threshold set for the materialized_view_optimization database option. See "Setting the optimizer staleness threshold for materialized views" on page 54.

If the materialized view meets the above criteria, and it is found to satisfy all or part of the query, the View Matching algorithm includes the materialized view in its recommendations for the enumeration phase of optimization, when the best plan is found based on cost. However, this does not mean that the materialized view will ultimately be used in the final execution plan. For example, materialized views that appear suitable for computing the result of a query may still not be used if another access plan, which doesn't use the materialized view, is estimated to be cheaper.

Determining the list of materialized view candidates

At any given time, you can obtain a list of all materialized views that are candidates to be considered by the optimizer, by executing the following command:

```
SELECT * FROM sa_materialized_view_info( ) WHERE AvailForOptimization='Y';
```

The list returned is specific to the requesting connection, since the optimizer takes into account option settings when generating the list. A materialized view is not considered a candidate if there is a mismatch between the options specified for the connection and the options that were in place when the materialized view was created. For a list of the options that must match, see "Restrictions on materialized views" on page 39.

To obtain a list of all materialized views that are not considered candidates for the connection because of a mismatch in option settings, execute the following from the connection that will execute the query:

```
SELECT * FROM sa_materialized_view_info( ) WHERE AvailForOptimization='O';
```

Determining if a materialized view was considered

You can see the list of materialized views used for a particular query by looking at the **Advanced Details** window of the query's graphical plan in Interactive SQL. See "Reading execution plans" on page 592.

You can also use Application Profiling mode in Sybase Central to determine whether a materialized view was considered during the enumeration phase of a query, by looking at the access plans enumerated by the optimizer. To see the access plans enumerated by the optimizer, tracing must be turned on, and must be configured to include the OPTIMIZATION_LOGGING tracing type. For more information about this tracing type, see "Application profiling" on page 158, and "Choosing a diagnostic tracing level" on page 171.

For more information about the enumeration phase of optimization, see "Query processing phases" on page 525.

Note
When snapshot isolation is in use, the optimizer does not consider materialized views that were refreshed after the start of the snapshot for the current transaction.

Materialized views and the View Matching algorithm

The View Matching algorithm determines whether materialized views can be used to satisfy a query. This determination takes place in two steps: a query evaluation step, and a materialized view evaluation step.

The optimizer includes a materialized view in the set of materialized views to be examined by the View Matching algorithm if the view definition:

- contains only one query block
- contains only one FROM clause
- does not contain any of the following constructs or specifications:
  - GROUPING SETS
  - CUBE
  - ROLLUP
  - subquery
  - derived table
  - UNION
  - EXCEPT
  - INTERSECT
  - materialized views
  - DISTINCT
  - TOP
  - FIRST
  - self-join
  - recursive join
  - FULL OUTER JOIN

The materialized view definition may contain a GROUP BY clause, and a HAVING clause, provided the HAVING clause does not contain subselects or subqueries.

Note
These restrictions only apply to the materialized views that are considered by the View Matching algorithm. If a materialized view is explicitly referenced in a query, the view is used by the optimizer as if it was a base table.

See also

- "Reading execution plans" on page 592
- "Query processing phases" on page 525
- "Application profiling" on page 158

Query evaluation

During query evaluation, the View Matching algorithm examines the query. If any of the following conditions are true, materialized views are not used to process the query.

- All the tables referenced by the query are updatable.

  The optimizer does not consider materialized views for a SELECT statement that is inherently updatable, or is explicitly declared in an updatable cursor. This situation can occur when using Interactive SQL, which utilizes updatable cursors by default for SELECT statements.

- The statement is a simple DML statement that uses optimizer bypass and is optimized heuristically. However, you can force cost-based optimization of any SELECT statement using the FORCE

OPTIMIZATION option of the OPTION clause. See "SELECT statement" [*SQL Anywhere Server - SQL Reference*].

- For queries contained inside stored procedures and user-defined functions, the query's execution plan has been cached. The database server may cache the execution plans for these queries so that they can be reused. For this class of queries, the query execution plan is cached after execution. The next time the query is executed, the plan is retrieved and all the phases up to the execution phase are skipped. See "Plan caching" on page 554.

Materialized view evaluation

Materialized view evaluation involves determining which of the existing materialized views can be used to compute all or parts of the query.

Once a materialized view has been matched with parts of a query, a decision is made whether to use the view in the final query execution plan; this decision is cost-based. The role of the enumeration phase is to generate plans containing views recommended by the View Matching algorithm and choose, based on the estimated cost of the plans, the best access plan which may or may not contain some of the materialized views.

If the materialized view is defined as a grouped-select-project-join query (also known as a **grouped query**, or a query containing a GROUP BY clause), then the View Matching algorithm can match it with grouped query blocks. If a materialized view is defined as a select-project-join query (that is, it is not a grouped query), then the View Matching algorithm can match it to any type of query block.

Listed below are the conditions necessary for the View Matching algorithm to decide if a view, V, matches part of a query block, QB, belonging to a query, Q. In general, V must contain a subset of the query QB's tables. The only exception is the extension tables of V. An extension table of V is a table that joins with exactly one row with the rest of the tables of V. For example, a primary key table is an extension table if its only predicate is an equijoin between a not-null foreign key column and its primary key column. For an example of a materialized view that contains an extension table, see "Example 2: Matching grouped-select-project-join views" on page 563.

- The option values used to create the materialized view V match the option values for the connection executing the query. For a list of the options that must match, see "Restrictions on materialized views" on page 39.

- The last refresh of the V materialized view, does not exceed the staleness threshold specified by the materialized_view_optimization database option, or by the MATERIALIZED VIEW OPTIMIZATION clause, if specified, in the SELECT statement. See "Setting the optimizer staleness threshold for materialized views" on page 54.

- All the tables used in V, with possible exceptions of some extension tables of V, are present in the QB. This set of common tables in the QB is hereinafter referred to as CT.

- No table in CT is updatable in the query Q.

- All tables in CT belong to the same side of an outer join in QB (that is, they are all in the preserved side of the outer join or all in the null-supplying side of an outer join of QB).
- It can be decided that the predicates in V subsume the subset of the predicates in QB that reference CT only. In other words, the predicates in V are less restrictive than those in QB. A predicate in QB that exactly matches one in V is called a **matched predicate**.
- Any expression of QB referencing tables in CT that is not used in a matched predicate must appear in the select list of V.
- If both V and QB are grouped, then QB doesn't contain extra tables besides the ones in CT. Additionally, the set of expressions in the GROUP BY clause of V must be equal to or a superset of the set of expressions in the GROUP BY clause of QB.
- If both V and QB are grouped on an identical set of expressions, all aggregate functions in QB must be also computed in V, or it is possible to compute them from V's aggregate functions. For example, if QB contains AVG(x) then V must contain AVG(x), or it must contain both SUM(x) and COUNT(x).
- If QB's GROUP BY clause is a subset of V's GROUP BY clause, then the simple aggregate functions of QB must be found among V's aggregate functions, while its composite aggregate functions have to be computed from simple aggregate functions of V. The simple aggregate functions are:

  - BIT_AND
  - BIT_OR
  - BIT_XOR
  - COUNT
  - LIST
  - MAX
  - MIN
  - SET_BITS
  - SUM
  - XMLAGG

  The composite aggregate functions that can be computed from the simple aggregate functions are:

  - SUM(x)
  - COUNT(x)
  - SUM(CAST(x AS DOUBLE))
  - SUM(CAST(x AS DOUBLE) * CAST(x AS DOUBLE))
  - VAR_SAMP(x)
  - VAR_POP(x)
  - VARIANCE(x)
  - STDDEV_SAMP(x)
  - STDDEV_POP(x)
  - STDDEV(x)

The following statistical aggregate functions:

- COVAR_SAMP(y,x)
- COVAR_POP(y,x)
- CORR(y,x)
- REGR_AVGX(y,x)
- REGR_AVGY(y,x)
- REGR_SLOPE(y,x)
- REGR_INTERCEPT(y,x)
- REGR_R2(y,x)
- REGR_COUNT(y,x)
- REGR_SXX(y,x)
- REGR_SYY(y,x)
- REGR_SXY(y,x)

can be computed from the following simple aggregate functions:

- SUM(y1)
- SUM(x1)
- COUNT(x1)
- COUNT(y1)
- SUM(x1*y1)
- SUM(y1*x1)
- SUM(x1*x1)
- SUM(y1*y1)

where x1 = CAST(IFNULL(x, x,y) AS DOUBLE)) and y1 = CAST(IFNULL(y,y,x) AS DOUBLE).

View Matching algorithm examples

Example 1: Matching select-project-join views

If a certain partition of a base table is frequently accessed by queries, then it may be beneficial to define a materialized view to store that partition. For example, the materialized view V_Canada defined below stores all the customers from the Customer table who live in Canada. As this materialized view is used when the State column is restricted to certain values, it is advisable to create the index V_Canada_State on the State column of the V_Canada materialized view.

```
CREATE MATERIALIZED VIEW V_Canada AS
 SELECT c.ID, c.City, c.State, c.CompanyName
  FROM Customers c
  WHERE c.State IN ( 'AB', 'BC', 'MB', 'NB', 'NL',
    'NT', 'NS', 'NU', 'ON', 'PE', 'QC', 'SK', 'YT' );
REFRESH MATERIALIZED VIEW V_Canada;
CREATE INDEX V_Canada_State on V_Canada( State );
```

Any query block that requires just a subset of customers living in Canada may benefit from this materialized view. For example, Query 1 below, which computes the total price of the products for all customers in Ontario for each company, may use the V_Canada materialized view in its access plans. An access plan for Query 1 using the V_Canada materialized view represents a valid plan as if Query 1 was

rewritten as Query 1_v, which is semantically equivalent to it. Note that the optimizer doesn't rewrite the query using the materialized views, instead the generated access plans using materialized views can theoretically be seen corresponding to the rewritten query.

The execution plan of the Query 1 uses the V_Canada materialized view, as shown here:
```
Work[ GrByH[ V_Canada<V_Canada_State> JNLO
SalesOrders<FK_CustomerID_ID> JNLO SalesOrderItems<FK_ID_ID> JNLO
Products<ProductsKey> ] ]
```

Query 1:

```
SELECT SUM( SalesOrderItems.Quantity
   * Products.UnitPrice ) AS Value
 FROM Customers c
  LEFT OUTER JOIN SalesOrders
   ON( SalesOrders.CustomerID = c.ID )
  LEFT OUTER JOIN SalesOrderItems
   ON( SalesOrderItems.ID = SalesOrders.ID )
  LEFT OUTER JOIN Products
   ON( Products.ID = SalesOrderItems.ProductID )
 WHERE c.State = 'ON'
 GROUP BY c.CompanyName;
```

Query 1_v:

```
SELECT SUM( SalesOrderItems.Quantity
   * Products.UnitPrice ) AS Value
  FROM V_Canada
   LEFT OUTER JOIN SalesOrders ON( SalesOrders.CustomerID = V_Canada.ID )
   LEFT OUTER JOIN SalesOrderItems ON( SalesOrderItems.ID = SalesOrders.ID )
   LEFT OUTER JOIN Products ON( Products.ID = SalesOrderItems.ProductID )
  WHERE V_Canada.State = 'ON'
  GROUP BY V_Canada.CompanyName;
```

Query 2 may use this view in both the main query block and the HAVING subquery. Some of the access plans enumerated by the optimizer using the V_Canada materialized view represent Query 2_v, which is semantically equivalent to Query 2 where the Customer table was replaced by the V_Canada view.

The execution plan is:
```
Work[ GrByH[ V_Canada<V_Canada_State> JNLO
SalesOrders<FK_CustomerID_ID> JNLO SalesOrderItems<FK_ID_ID> JNLO
Products<ProductsKey> ] ] : GrByS[ V_Canada<seq> JNLO
SalesOrders<FK_CustomerID_ID> JNLO SalesOrderItems<FK_ID_ID> JNLO
Products<ProductsKey>
```

Query 2:

```
SELECT  SUM( SalesOrderItems.Quantity
  * Products.UnitPrice ) AS Value
 FROM Customers c
  LEFT OUTER JOIN SalesOrders
   ON( SalesOrders.CustomerID = c.ID )
  LEFT OUTER JOIN SalesOrderItems
   ON( SalesOrderItems.ID = SalesOrders.ID )
  LEFT OUTER JOIN Products
   ON( Products.ID = SalesOrderItems.ProductID )
 WHERE c.State = 'ON'
 GROUP BY CompanyName
```



```
 HAVING Value >
  ( SELECT AVG( SalesOrderItems.Quantity
        * Products.UnitPrice ) AS Value
      FROM Customers c1
       LEFT OUTER JOIN SalesOrders
       ON( SalesOrders.CustomerID = c1.ID )
       LEFT OUTER JOIN SalesOrderItems
       ON( SalesOrderItems.ID = SalesOrders.ID )
        LEFT OUTER JOIN Products
       ON( Products.ID = SalesOrderItems.ProductID )
       WHERE c1.State IN ('AB', 'BC', 'MB', 'NB', 'NL', 'NT', 'NS',
      'NU', 'ON', 'PE', 'QC', 'SK', 'YT' ) );
```

Query 2_v:

```
SELECT  SUM( SalesOrderItems.Quantity
         * Products.UnitPrice ) AS Value
 FROM V_Canada
 LEFT OUTER JOIN SalesOrders
  ON( SalesOrders.CustomerID=V_Canada.ID )
 LEFT OUTER JOIN SalesOrderItems
  ON( SalesOrderItems.ID=SalesOrders.ID )
 LEFT OUTER JOIN Products
  ON( Products.ID=SalesOrderItems.ProductID )
WHERE V_Canada.State = 'ON'
GROUP BY V_Canada.CompanyName
HAVING Value >
   ( SELECT AVG( SalesOrderItems.Quantity
         * Products.UnitPrice ) AS Value
       FROM V_Canada
        LEFT OUTER JOIN SalesOrders
        ON( SalesOrders.CustomerID = V_Canada.ID )
        LEFT OUTER JOIN SalesOrderItems
        ON( SalesOrderItems.ID = SalesOrders.ID )
         LEFT OUTER JOIN Products
        ON( Products.ID = SalesOrderItems.ProductID )
        WHERE V_Canada.State IN ('AB', 'BC', 'MB',
       'NB', 'NL', 'NT', 'NS', 'NU', 'ON', 'PE', 'QC',
       'SK', 'YT' ) );
```

Example 2: Matching grouped-select-project-join views

The grouped materialized views have the potential for the highest performance impact on the grouped queries. If similar aggregations are used in frequently-executed queries, a materialized view should be defined to pre-aggregate data on a superset of the group by clauses used in those queries. Composite aggregate functions of the queries can be computed from the simple aggregates used in the views. So, it is recommended that only simple aggregate functions are stored in the materialized views.

The materialized view V_Quantity, below, precomputes the sum and count of quantities per product for each month and year. Query 3, below, can use this view to select only the months of the year 2000 (the short plan is `Work[ GrByH[ V_Quantity<seq> ] ]`, corresponding to Query 3_v).

Query 4, which doesn't reference the extension table SalesOrders, can still use V_Quantity as the view contains all the data necessary to compute Query 4 (the short plan is `Work[ GrByH[ V_Quantity<seq> ] ]`, corresponding to Query 4_v).

```
CREATE MATERIALIZED VIEW V_Quantity AS
 SELECT s.ProductID,
  Month( o.OrderDate ) AS month,
  Year( o.OrderDate ) AS year,
```

```
    SUM( s.Quantity ) AS q_sum,
    COUNT( s.Quantity  ) AS q_count
  FROM SalesOrderItems s KEY JOIN SalesOrders o
  GROUP BY s.ProductID, Month( o.OrderDate ),
    Year( o.OrderDate );
REFRESH MATERIALIZED VIEW V_Quantity;
```

Query 3:

```
SELECT s.ProductID,
  Month( o.OrderDate ) AS month,
  AVG( s.Quantity) AS avg,
  SUM( s.Quantity ) AS q_sum,
  COUNT( s.Quantity ) AS q_count
 FROM SalesOrderItems s KEY JOIN SalesOrders o
 WHERE year( o.OrderDate ) = 2000
 GROUP BY s.ProductID, Month( o.OrderDate );
```

Query 3_v:

```
SELECT V_Quantity.ProductID,
  V_Quantity.month AS month,
  SUM( V_Quantity.q_sum ) / SUM( V_Quantity.q_count )
   AS avg,
  SUM( V_Quantity.q_sum ) AS q_sum,
  SUM( V_Quantity.q_count ) AS q_count
 FROM V_Quantity
 WHERE V_Quantity.year  = 2000
 GROUP BY V_Quantity.ProductID, V_Quantity.month;
```

Query 4:

```
SELECT s.ProductID,
  AVG( s.Quantity ) AS avg,
  SUM( s.Quantity ) AS sum
 FROM SalesOrderItems s
 WHERE s.ProductID IS NOT NULL
 GROUP BY s.ProductID;
```

Query 4_v

```
SELECT V_Quantity.ProductID,
  SUM( V_Quantity.q_sum ) / SUM( V_Quantity.q_count )
   AS avg,
  SUM( V_Quantity.q_sum ) AS sum
 FROM V_Quantity
 WHERE V_Quantity.ProductID IS NOT NULL
 GROUP BY V_Quantity.ProductID;
```

Example 3: Matching complex queries

The View Matching algorithm is applied per query block, so it is possible to use more than one materialized view per query block and also more than one materialized view for the whole query. Query 5 below may use the three materialized views: V_Canada for one of the null-supplying sides of the LEFT OUTER JOIN; V_ShipDate, defined below, for the preserved side of the main query block; and V_Quantity for the subquery block. The execution plan for Query 5_v is:

```
Work[ Window[ Sort[ V_ShipDate<V_ShipDate_date> JNLO
( so<SalesOrdersKey> JH V_Canada<V_Canada_State> ) ] ] ] :
GrByS[V_Quantity<seq> ].
```



```
CREATE MATERIALIZED VIEW V_ShipDate AS
 SELECT s.ProductID, p.Description,
  s.Quantity, s.ShipDate, s.ID
 FROM SalesOrderItems s KEY JOIN Products p ON ( s.ProductId = p.ID )
 WHERE s.ShipDate >= '2000-01-01'
  AND s.ShipDate <= '2001-01-01';
REFRESH MATERIALIZED VIEW V_ShipDate;
CREATE INDEX V_ShipDate_date ON V_ShipDate( ShipDate );
```

Query 5:

```
SELECT p.ID, p.Description, s.Quantity,
  s.ShipDate, so.CustomerID, c.CompanyName,
  SUM( s.Quantity ) OVER ( PARTITION BY s.ProductID
                       ORDER BY s.ShipDate
                       ROWS BETWEEN UNBOUNDED PRECEDING
                       AND CURRENT ROW ) AS cumulative_qty
 FROM SalesOrderItems s JOIN Products p
  ON (s.ProductID = p.ID) LEFT OUTER JOIN (
   SalesOrders so JOIN Customers c
    ON ( c.ID = so.CustomerID AND c.State = 'ON' ) )
  ON (s.ID = so.ID )
 WHERE s.ShipDate >= '2000-01-01'
   AND s.ShipDate <= '2000-12-31'
   AND s.Quantity  > ( SELECT  AVG( s.Quantity ) AS avg
                       FROM SalesOrderItems s KEY JOIN SalesOrders o
                       WHERE year( o.OrderDate) = 2000 )
 FOR READ ONLY;
```

Query 5_v:

```
SELECT  V_ShipDate.ID, V_ShipDate.Description,
  V_ShipDate.Quantity, V_ShipDate.ShipDate,
  so.CustomerID, V_Canada.CompanyName,
  SUM( V_ShipDate.Quantity ) OVER ( PARTITION BY V_ShipDate.ProductID
                                    ORDER BY V_ShipDate.ShipDate
                                    ROWS BETWEEN UNBOUNDED PRECEDING
                                    AND CURRENT ROW ) AS cumulative_qty
 FROM V_ShipDate
  LEFT OUTER JOIN ( SalesOrders so JOIN V_Canada
   ON ( V_Canada.ID = so.CustomerID AND V_Canada.State = 'ON' ) )
  ON ( V_ShipDate.ID = so.ID )
 WHERE V_ShipDate.ShipDate >= '2000-01-01'
  AND V_ShipDate.ShipDate <= '2000-12-31'
  AND V_ShipDate.Quantity >
    ( SELECT SUM( V_Quantity.q_sum ) / SUM( V_Quantity.q_count )
      FROM V_Quantity
      WHERE V_Quantity.year = 2000 )
 FOR READ ONLY;
```

Example 4: Matching materialized views with OUTER JOINs

The view matching algorithm can match views and queries with OUTER JOINs using similar rules as for the views with only inner joins. Null-supplying sides of the OUTER JOINs of a materialized view may not appear in the query as long as all the tables in the null-supplying side are extension tables. The query is allowed to contain inner joins to match the view's outer joins. Queries 6_v, 7_v, 8_v, and 9_v below illustrate how a materialized view containing an OUTER JOIN in its definition can be used to answer queries.

Query 6 below matches exactly the materialized view V_SalesOrderItems_2000 and can be evaluated as if it is Query 6_v.

Query 7 contains some extra predicates on the preserved side of the outer join and it can still be computed using V_SalesOrderItems_2000. Note that the null-supplying table, Products, is an extension table in the view V_SalesOrderItems_2000. This means the view can also be matched with Query 8, which does not contain the Products table.

Query 9 contains only the inner join of the tables SalesOrderItems and Products, and it is matched with the V_SalesOrderItems_2000 view by selecting only those rows of the view which are not null-supplying rows from the table Products. The extra predicate, V.Description IS NOT NULL, in Query 9_v is used to select exactly those rows which are not null-supplied.

```
CREATE MATERIALIZED VIEW V_SalesOrderItems_2000 AS
 SELECT s.ProductID, p.Description,
  s.Quantity, s.ShipDate, s.ID
 FROM SalesOrderItems s LEFT OUTER JOIN Products p
    ON ( s.ProductId = p.ID )
 WHERE s.ShipDate >= '2000-01-01'
  AND s.ShipDate <= '2001-01-01';
REFRESH MATERIALIZED VIEW V_SalesOrderItems_2000;
CREATE INDEX V_SalesOrderItems_shipdate ON V_SalesOrderItems_2000( ShipDate );
```

Query 6:

```
SELECT s.ProductID, p.Description,
  s.Quantity, s.ShipDate, s.ID
 FROM SalesOrderItems s LEFT OUTER JOIN Products p
    ON ( s.ProductId = p.ID )
 WHERE s.ShipDate >= '2000-01-01'
  AND s.ShipDate <= '2001-01-01'
 FOR READ ONLY;
```

Query 6_v:

```
SELECT V.ProductID,  V.Description, V.Quantity, V.ShipDate,
  SUM( V.Quantity ) OVER ( PARTITION BY V.ProductID
                      ORDER BY V.ShipDate
                      ROWS BETWEEN UNBOUNDED PRECEDING
                      AND CURRENT ROW ) AS cumulative_qty
 FROM V_SalesOrderItems_2000 as V
 FOR READ ONLY;
```

Query 7:

```
SELECT s.ProductID,  p.Description, s.Quantity, s.ShipDate,
  SUM( s.Quantity ) OVER ( PARTITION BY s.ProductID
                      ORDER BY s.ShipDate
                      ROWS BETWEEN UNBOUNDED PRECEDING
                      AND CURRENT ROW ) AS cumulative_qty
 FROM SalesOrderItems s LEFT OUTER JOIN Products p
    ON (s.ProductID = p.ID )

 WHERE s.ShipDate >= '2000-01-01'
   AND s.ShipDate <= '2001-01-01'
   AND s.Quantity >= 50
 FOR READ ONLY;
```

Query 7_v:

```
SELECT V.ProductID,  V.Description, V.Quantity, V.ShipDate,
  SUM( V.Quantity ) OVER ( PARTITION BY V.ProductID
```



```
                        ORDER BY V.ShipDate
                        ROWS BETWEEN UNBOUNDED PRECEDING
                        AND CURRENT ROW ) AS cumulative_qty
  FROM V_SalesOrderItems_2000 as V
  WHERE V.Quantity >= 50
  FOR READ ONLY;
```

Query 8:

```
  SELECT s.ProductID, s.Quantity, s.ShipDate,
    SUM( s.Quantity ) OVER ( PARTITION BY s.ProductID
                        ORDER BY s.ShipDate
                        ROWS BETWEEN UNBOUNDED PRECEDING
                        AND CURRENT ROW ) AS cumulative_qty
   FROM SalesOrderItems s
   WHERE s.ShipDate >= '2000-01-01'
     AND s.ShipDate <= '2001-01-01'
     AND s.Quantity >= 50
   FOR READ ONLY;
```

Query 8_v:

```
  SELECT V.ProductID, V.Quantity, V.ShipDate,
    SUM( V.Quantity ) OVER ( PARTITION BY V.ProductID
                        ORDER BY V.ShipDate
                        ROWS BETWEEN UNBOUNDED PRECEDING
                        AND CURRENT ROW ) AS cumulative_qty
   FROM V_SalesOrderItems_2000 as V
   WHERE V.Quantity >= 50
   FOR READ ONLY;
```

Query 9:

```
  SELECT s.ProductID,  p.Description, s.Quantity, s.ShipDate,
    SUM( s.Quantity ) OVER ( PARTITION BY s.ProductID
                        ORDER BY s.ShipDate
                        ROWS BETWEEN UNBOUNDED PRECEDING
                        AND CURRENT ROW ) AS cumulative_qty
   FROM SalesOrderItems s JOIN Products p
        ON (s.ProductID = p.ID )

   WHERE s.ShipDate >= '2000-01-01'
     AND s.ShipDate <= '2001-01-01'
   FOR READ ONLY;
```

Query 9_v:

```
  SELECT V.ProductID,  V.Description, V.Quantity, V.ShipDate,
    SUM( V.Quantity ) OVER ( PARTITION BY V.ProductID
                        ORDER BY V.ShipDate
                        ROWS BETWEEN UNBOUNDED PRECEDING
                        AND CURRENT ROW ) AS cumulative_qty
   FROM V_SalesOrderItems_2000 as V
   WHERE V.Description IS NOT NULL
   FOR READ ONLY;
```

Query execution algorithms

The function of the optimizer is to translate certain SQL statements (SELECT, INSERT, UPDATE, or DELETE) into an efficient access plan made up of various relational algebra operators (join, duplicate elimination, union, and so on). The operators within the access plan may not be structurally equivalent to the original SQL statement, but the access plan's various operators will compute a result that is semantically equivalent to that SQL request.

Relational algebra operators in access plans

An access plan consists of a tree of relational algebra operators which, starting at the leaves of the tree, consume the base inputs to the query (usually rows from a table) and process the rows from bottom to top, so that the root of the tree yields the final result. Access plans can be viewed graphically for ease of comprehension. See "Reading execution plans" on page 592, and "Reading graphical plans" on page 595.

SQL Anywhere supports multiple implementations of these various relational algebra operations. For example, SQL Anywhere supports three different implementations of inner join: nested loops join, merge join, and hash join. Each of these operators can be advantageous to use in specific circumstances: some of the parameters that the query optimizer analyzes to make its choice include the amount of table data in cache, the characteristics and selectivity of the join predicate, the sortedness of the inputs to the join and the output from it, the amount of memory available to perform the join, and a variety of other factors.

SQL Anywhere may dynamically, at execution time, switch from the physical algebraic operator chosen by the optimizer to a different physical algorithm that is logically equivalent to the original. Typically, this alternative access plan is used in one of two circumstances:

- When the total amount memory used to execute the statement is close to a memory governor threshold, then a switch is made to a strategy that may execute more slowly, but that frees a substantial amount of memory for use by other operators (or other requests). When this occurs, the QueryLowMemoryStrategy property is incremented. This information also appears in the graphical plan for the statement. For information about the QueryLowMemoryStrategy property, see "Connection properties" [*SQL Anywhere Server - Database Administration*].

  The amount of memory that can be used by an operator dependent upon the multiprogramming level of the server, and the number of active connections.

  For more information about how the memory governor and the multiprogramming level, see:

  - "Threading in SQL Anywhere" [*SQL Anywhere Server - Database Administration*]
  - "Configuring the database server's multiprogramming level" [*SQL Anywhere Server - Database Administration*]
  - "The memory governor" on page 549

- If, at the beginning of its execution, the specific operator (a hash inner join, for example) determines that its inputs are not of the expected cardinality as that computed by the optimizer at optimization time. In this case, the operator may switch to a different strategy that will be less expensive to execute. Typically, this alternative strategy utilizes index nested loops processing. For the case of hash join, the QueryJHToJNLOptUsed property is incremented when this switch occurs. The occurrence of the join method switch is also included in the statement's graphical plan. For information about the

QueryJHToJNLOptUsed property, see "Connection properties" [*SQL Anywhere Server - Database Administration*].

Parallelism during query execution

SQL Anywhere supports two different kinds of parallelism for query execution: inter-query, and intra-query. Inter-query parallelism involves executing different requests simultaneously on separate CPUs. Each request (task) runs on a single thread and executes on a single processor.

Intra-query parallelism involves having more than one CPU handle a single request simultaneously, so that portions of the query are computed in parallel on multi-processor hardware. Processing of these portions is handled by the Exchange algorithm. See "Exchange algorithm (Exchange)" on page 588.

Intra-query parallelism can benefit a workload where the number of simultaneously-executing queries is usually less than the number of available processors. The maximum degree of parallelism is controlled by the setting of the max_query_tasks option. See "max_query_tasks option" [*SQL Anywhere Server - Database Administration*].

The optimizer estimates the extra cost of parallelism (extra copying of rows, extra costs for co-ordination of effort) and chooses parallel plans only if they are expected to improve performance.

Intra-query parallelism is not used for connections with the priority option set to background. See "priority option" [*SQL Anywhere Server - Database Administration*].

Intra-query parallelism is not used if the number of server threads that are currently handling a request (ActiveReq server property) recently exceeded the number of CPU cores on the computer that the database server is licensed to use. The exact period of time is decided by the server and is normally a few seconds. See "Database server properties" [*SQL Anywhere Server - Database Administration*].

Parallel execution

Whether a query can take advantage of parallel execution depends on a variety of factors:

- the available resources in the system at the time of optimization (such as memory, amount of data in cache, and so on)
- the number of logical processors on the computer
- the number of disk devices used for the storage of the database, and their speed relative to that of the processor and the computer's I/O architecture.

- the specific algebraic operators required by the request. SQL Anywhere supports five algebraic operators that can execute in parallel:
  - parallel sequential scan (table scan)
  - parallel index scan
  - parallel hash join, and parallel versions of hash semijoin and anti-semijoin
  - parallel nested loop joins, and parallel versions of nested loop semijoin and anti-semijoin
  - parallel hash filter
  - parallel hash group by

A query that uses unsupported operators can still execute in parallel, but the supported operators must appear below the unsupported ones in the plan (as viewed in dbisql). A query where most of the unsupported operators can appear near the top is more likely to use parallelism. For example, a sort operator cannot be parallelized but a query that uses an ORDER BY on the outermost block may be parallelized by positioning the sort at the top of the plan and all the parallel operators below it. In contrast, a query that uses a TOP *n* and ORDER BY in a derived table is less likely to use parallelism since the sort must appear somewhere other than the top of the plan.

By default, SQL Anywhere assumes that any dbspace resides on a disk subsystem with a single platter. While there can be advantages to parallel query execution in such an environment, the optimizer's I/O cost model for a single device makes it difficult for the optimizer to choose a parallel table or index scan unless the table data is fully resident in the cache. However, by calibrating the I/O subsystem using the ALTER DATABASE CALIBRATE PARALLEL READ statement, the optimizer can then cost with greater accuracy the benefits of parallel execution. The optimizer is much more likely to choose execution plans with some degree of parallelism for multiple spindles.

When intra-query parallelism is used for an access plan, the plan contains an Exchange operator whose effect is to merge (union) the results of the parallel computation of each subtree. The number of subtrees underneath the Exchange operator is the degree of parallelism. Each subtree, or access plan component, is a database server task. The database server kernel schedules these tasks for execution in the same manner as if they were individual SQL requests, based on the availability of execution threads (or fibers). This architecture means that parallel computation of any access plan is largely self-tuning, in that work for a parallel execution task is scheduled on a thread (fiber) as the server kernel allows, and execution of the plan components is performed evenly.

See also

- "-gn dbsrv12 server option" [*SQL Anywhere Server - Database Administration*]
- "max_query_tasks option" [*SQL Anywhere Server - Database Administration*]
- "Threading in SQL Anywhere" [*SQL Anywhere Server - Database Administration*]
- "-gtc dbeng12/dbsrv12 server option" [*SQL Anywhere Server - Database Administration*]
- "Configuring the database server's multiprogramming level" [*SQL Anywhere Server - Database Administration*]
- "Exchange algorithm (Exchange)" on page 588
- "Reading execution plans" on page 592
- "ALTER DATABASE statement" [*SQL Anywhere Server - SQL Reference*]

Parallelism in queries

A query is more likely to use parallelism if the query processes a lot more rows than are returned. In this case, the number of rows processed includes the size of all rows scanned plus the size of all intermediate results. It does not include rows that are never scanned because an index is used to skip most of the table. An ideal case is a single-row GROUP BY over a large table, which scans many rows and returns only one. Multi-group queries are also candidates if the size of the groups is large. Any predicate or join condition that drops a lot of rows is also a good candidate for parallel processing.

Following is a list of circumstances in which a query can not take advantage of parallelism, either at optimization or execution time:

- the server computer does not have multiple processors
- the server computer is not licensed to use multiple processors. You can check this by looking at the NumLogicalProcessorsUsed server property. However, note that hyperthreaded processors are not counted for intra-query parallelism so you must divide the value of NumLogicalProcessorsUsed by two if the computer is hyperthreaded.
- the max_query_tasks option is set to 1
- the priority option is set to background
- the statement containing the query is not a SELECT statement
- the value of ActiveReq has been greater than, or equal to, the value of NumLogicalProcessorsUsed at any time in the recent past (divide the number of processors by two if the computer is hyperthreaded)
- there are not enough available tasks.

See also

- "Parallelism during query execution" on page 569
- "Threading in SQL Anywhere" [*SQL Anywhere Server - Database Administration*]
- "max_query_tasks option" [*SQL Anywhere Server - Database Administration*]
- "priority option" [*SQL Anywhere Server - Database Administration*]
- max_query_tasks, priority, NumLogicalProcessorsUsed, and ActiveReq properties: "Database server properties" [*SQL Anywhere Server - Database Administration*]
- "CREATE DATABASE statement" [*SQL Anywhere Server - SQL Reference*]
- "ALTER TABLE statement" [*SQL Anywhere Server - SQL Reference*]

Table access methods

This section explains the various methods used to access tables, with table and index scans being the most common methods.

IndexScan method

IndexScan uses an index to determine which rows satisfy a search condition. Index scans help reduce the set of qualifying rows before accessing the table. Index scans also return rows in sorted order.

IndexScan appears in the short plan as *correlation-name*`<`*index-name*`>`, where *correlation-name* is the correlation name specified in the FROM clause, or the table name if none was specified, and *index-name* is the name of the index.

Indexes provide an efficient mechanism for reading a few rows from a large table. However, an index scan can be more expensive than a sequential scan when reading many rows from a table. Index scans cause pages to be read from the database in random order, which is more expensive than sequential reads. Index scans may also reference the same table page multiple times if there are several rows on the page that satisfy the search condition. If only a few pages are matched by the index scan, it is likely that the pages will remain in cache, and multiple access does not lead to extra I/O. However, if many pages are matched by the search condition, they may not all fit in cache. This can lead to the index scan reading the same page from disk multiple times.

The optimizer uses an index scan to satisfy a search condition if the search condition is sargable, and if the optimizer's estimate of the selectivity of the search condition is sufficiently low for the index scan to be cheaper than a sequential table scan.

An index scan can also evaluate non-sargable search conditions after rows are fetched from the index. Evaluating conditions in the index scan is slightly more efficient than evaluating them in a filter after the index scan.

Even if there are no search conditions to satisfy, indexes can also be used to satisfy an ordering requirement, either explicitly defined in an ORDER BY clause, or implicitly needed for a GROUP BY or DISTINCT clause. Ordered group-by and ordered distinct methods can return initial rows faster than hash-based grouping and distinct, but they may be slower at returning the entire result set.

The optimizer tends to prefer index scans over sequential table scans if the optimization_goal setting is first-row. This is because indexes tend to return the first few rows of a query faster than table scans.

When writing a query, you can specify index hints to tell the optimizer which indexes to use and how to use them. However, 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. See “FROM clause” [*SQL Anywhere Server - SQL Reference*].

IndexOnlyScan method (IO)

When an index used by the optimizer contains all the data from the underlying table that is required to satisfy the query, it may be possible to completely avoid reading values from the underlying table and retrieve the data directly from the indexes. This is referred to as an **index-only retrieval**. Index-only retrievals reduce the amount of I/O and cache required to satisfy a query, and improve performance. The optimizer performs an index only retrieval whenever possible.

See also

- "FROM clause" [*SQL Anywhere Server - SQL Reference*]
- "Using predicates in queries" on page 551
- "optimization_goal option" [*SQL Anywhere Server - Database Administration*]
- "Reading execution plans" on page 592

MultipleIndexScan method (MultIdx)

MultipleIndexScan is used when more than one index can or must be used to satisfy a query that contains a set of search conditions that are combined with the logical operators AND or OR. MultipleIndexScan combines multiple IndexScan methods with other operators to satisfy the search conditions.

When multiple indexes are used to evaluate predicates combined using an AND operator, MultipleIndexScan performs an index intersection operation. When used to evaluate predicates combined using an OR operator, MultipleIndexScan performs an index union operation. However, note that MultipleIndexScan is not restricted to the union or intersection operations; for example, MultipleIndexScan may perform and index union by using outer joins.

You can determine whether a multiple index scan is used for a particular query by examining the execution plan. In a short plan, a multiple index scan method appears as *table-name*`<MultIdx...`, followed by a list of the indexes that were used.

In long and graphical plans, the use of a multiple index scan is indicated by a MultipleIndexScan node, where the entries under the node provide details about which indexes were used, and how their results were combined.

See also

- "FROM clause" [*SQL Anywhere Server - SQL Reference*]
- "Using predicates in queries" on page 551
- "Reading execution plans" on page 592

ParallelIndexScan method

When ParallelIndexScan is used, individual IndexScan operators work together under an Exchange operator to do an index scan in parallel. As each IndexScan operator requires rows, it takes the next unprocessed leaf page and returns rows from the page, one at a time. In this way, pages are divided between the IndexScan operators to achieve parallel processing. Regardless of how the pages are distributed among the IndexScan operators, all the rows are visited.

TableScan method (seq)

TableScan reads all the rows in all the pages of a table in the order in which they are stored in the database. This is known as a sequential table scan.

Sequential table scans appear in the short and long text plan as *correlation_name<seq>*, where *correlation_name* is the correlation name specified in the FROM clause, or the table name if none was specified.

Sequential table scans are used when it is likely that most of the table pages have a row that match the query's search condition or a suitable index is not defined.

Although sequential table scans may read more pages than index scans, the disk I/O can be substantially cheaper because the pages are read in contiguous blocks from the disk (this performance improvement is best if the database file is not fragmented on the disk). Sequential I/O reduces disk head movement and rotational latency. For large tables, sequential table scans also read groups of several pages at a time. This further reduces the cost of sequential table scans relative to index scans.

Although sequential table scans may take less time than index scans that match many rows, they also cannot exploit the cache as effectively as index scans if the scan is executed many times. Since index scans are likely to access fewer table pages, it is more likely that the pages will be available in the cache, resulting in faster access. Because of this, it is much better to have an index scan for table accesses that are repeated, such as the right-hand side of a nested loops join.

For transactions executing at isolation level 3, SQL Anywhere acquires a lock on each row that is accessed —even if it does not satisfy the search condition. At this isolation level, sequential table scans acquire locks on all the rows in the table, while index scans only acquire locks on the rows that match the search condition. This means that sequential table scans may substantially reduce the throughput in multi-user environments. For this reason, the optimizer strongly prefers indexed access over sequential access at isolation level 3. Sequential scans can efficiently evaluate simple comparison predicates between table columns and constants during the scan. Other search conditions that refer only to the table being scanned are evaluated after these simple comparisons, and this approach is slightly more efficient than evaluating the conditions in a filter after the sequential scan.

ParallelTableScan method

When ParallelTableScan is used, individual TableScan operators work together under an Exchange operator to do a sequential table scan in parallel. As each TableScan operator requires rows, it takes the next unprocessed table page and returns rows from the page, one at a time. In this way, pages are divided between the TableScan operators to achieve parallel processing. Regardless of how the pages are distributed among the parallel TableScan operators, all the rows in the table are visited.

HashTableScan method (HTS)

HashTableScan scans the build side of a hash join as if it were an in-memory table, thereby converting a plan with first structure below, to that of the second structure below, where `idx` is an index that can be used to probe the join key values stored in the hash table:

```
table1<seq>*JH ( <operator>... ( table2<seq> ) )
```

```
table1<seq>*JF ( <operator>... ( HTS JNB table2<idx> ) )
```


When there are intervening operators between the hash join and the scan, a hash table scan reduces the number of rows needed that must be processed by other operators. This strategy is most useful when the index probes are highly selective, for example, when the number of rows in the build side is small compared to the cardinality of the index.

Note
If the build side of the hash join is large, it is more effective to do a regular sequential scan.

The optimizer computes a threshold build size, similar to how it computes the threshold for the hash join alternate execution. If the number of rows in the build side exceeds this threshold, HashTableScan is abandoned and the `(HTS JNB table<idx>)` is treated as a sequential scan `(table<seq>)` during execution.

Note
The sequential strategy is used if the build side of the hash table has to spill to disk.

RowIdScan method (ROWID)

RowIdScan is used to locate a row in a base or temporary table based on an equality comparison predicate that uses the ROWID function. The comparison predicate may refer to a constant literal, but more commonly the ROWID function is used with a row identifier value returned by a system function or procedure call, such as sa_locks.

RowId scans appear in the short and long text plan as `correlation-name<ROWID>`, where *correlation-name* is the correlation name specified in the FROM clause, or the table name if no correlation name was specified.

It is impossible for RowIdScan to differentiate between an invalid row identifier for the given table referenced by the ROWID function, and a situation where the given row identifier no longer exists. So, RowIdScan returns the empty set if the row identifier specified in the comparison predicate cannot be found in the table.

See also

- "ROWID function [Miscellaneous]" [*SQL Anywhere Server - SQL Reference*]
- "sa_locks system procedure" [*SQL Anywhere Server - SQL Reference*]

Types of algorithms

Join algorithms

SQL Anywhere supports a variety of different join implementations that the query optimizer chooses from. Each of the join algorithms has specific characteristics that make it more or less suitable for a given query and a given execution environment.

The order of the joins in an access plan may or may not correspond to the ordering of the joins in the original SQL statement; the query optimizer is responsible for choosing the best join strategy for each query based on the lowest execution cost. In some situations, query rewrite optimizations may be utilized for complex statements that either increase, or decrease, the number of joins computed for any particular statement.

There are three classes of join algorithms supported by SQL Anywhere, though each of them has additional variants:

- **Nested Loops Join** The most straightforward algorithm is Nested Loops Join. For each row on the left-hand side, the right-hand side is scanned for a match based on the join condition. Ordinarily, rows on the right-hand side are accessed through an index to reduce the overall execution cost. This scenario is frequently referred to as an **Index Nested Loops Join**.

  Nested Loops Join has variants that support LEFT OUTER and FULL OUTER joins. A nested loops implementation can also be used for semijoins (most often used for processing EXISTS subqueries).

  A Nested Loops Join can be utilized no matter what the characteristics of the join condition, although a join over inequality conditions can be very inefficient to compute.

  A Nested Loops FULL OUTER join is very expensive to execute over inputs of any size, and is only chosen by the query optimizer as a last resort when no other join algorithm is possible.

- **Merge Join** A Merge Join relies on its two inputs being sorted on the join attributes. The join condition must contain at least one equality predicate in order for this method to be chosen by the query optimizer. The basic algorithm is a straightforward merge of the two inputs: when the values of the two join attributes differ, the algorithm scrolls to the next row of the left or right-hand side, depending on which side has the lower of the two values. Backtracking may be necessary when there is more than one match.

  There are Merge Join variants to support LEFT OUTER and FULL OUTER joins. Merge Join for FULL OUTER Joins is considerably more efficient than its nested loops counterpart.

  The basic Merge Join algorithm is also used to support the SQL set operators EXCEPT and INTERSECT, although these variants are explicitly named as EXCEPT or INTERSECT algorithms within an access plan.

- **Hash Join** A Hash Join is the most versatile join method supported by the SQL Anywhere database server. The Hash Join algorithm builds an in-memory hash table of the smaller of its two inputs, and then reads the larger input and probes the in-memory hash table to find matches.

  Hash Join variants exist to support LEFT OUTER join, FULL OUTER join, semijoin, and anti-semijoin. In addition, SQL Anywhere supports hash join variants for recursive INNER and LEFT OUTER joins when a recursive UNION query expression is being used.

  The Hash Inner Join, Left Outer Join, Semijoin, and Antisemijoin algorithms can be executed in parallel.

  If the in-memory hash table constructed by the algorithm does not fit into available memory, the Hash Join algorithm splits the input into partitions (possibly recursively for very large inputs) and performs the join on each partition independently. If there is not enough cache memory to hold all the rows that

have a particular value of the join attributes, then, if possible, each Hash Join dynamically switches to an index-based nested loops strategy after first discarding the interim results to avoid exhausting the statement's memory consumption quota.

Variants of Hash Join are also utilized to support the SQL query expressions EXCEPT and INTERSECT, although these variants are explicitly named as EXCEPT or INTERSECT algorithms within an access plan.

HashJoin algorithms (JH, JHSP, JHFO, JHAP, JHO, JHPO)

HashJoin builds an in-memory hash table of the smaller of its two inputs, and then reads the larger input and probes the in-memory hash table to find matches, which are written to a work table. If the smaller input does not fit into memory, HashJoin partitions both inputs into smaller work tables. These smaller work tables are processed recursively until the smaller input fits into memory.

HashJoin also:

- computes all the rows in its result before returning the first row
- uses a work table, which provides insensitive semantics unless a value-sensitive cursor has been requested
- can be executed in parallel
- locks rows in its inputs before they are copied to memory

HashJoin has the best performance if the smaller input fits into memory, regardless of the size of the larger input. In general, the optimizer chooses hash join if one of the inputs is expected to be substantially smaller than the other.

If HashJoin executes in an environment where there is not enough cache memory to hold all the rows that have a particular value of the join attributes, then it is not able to complete. In this case, HashJoin discards the interim results and an indexed-based NestedLoopsJoin is used instead. All the rows of the smaller table are read and used to probe the work table to find matches. This indexed-based strategy is significantly slower than other join methods, and the optimizer avoids generating access plans using a hash join if it detects that a low memory situation may occur during query execution.

The amount of memory that can be used by a HashJoin operator is dependent upon the multiprogramming level of the server, and the number of active connections. See “Threading in SQL Anywhere” [*SQL Anywhere Server - Database Administration*], and “Configuring the database server's multiprogramming level” [*SQL Anywhere Server - Database Administration*].

When the nested loops strategy is needed due to low memory, a performance counter is incremented. You can read this monitor with the QueryLowMemoryStrategy database or connection property, or in the Query: Low Memory Strategies counter in the Windows Performance Monitor.

In low memory conditions, HashJoin is disabled on Windows Mobile.

Note
The Windows Performance Monitor may not be available on Windows Mobile.

For more information, see QueryLowMemoryStrategy in "Connection properties" [*SQL Anywhere Server - Database Administration*], and "Configuring the database server's multiprogramming level" [*SQL Anywhere Server - Database Administration*].

RecursiveHashJoin algorithm (JHR)

RecursiveHashJoin is a variant of HashJoin, and is used for recursive union queries. For more information, see "HashJoin algorithms (JH, JHSP, JHFO, JHAP, JHO, JHPO)" on page 577, and "Recursive common table expressions" on page 435.

RecursiveLeftOuterHashJoin algorithm (JHRO)

RecursiveLeftOuterHashJoin is a variant of HashJoin, and is used for certain recursive union queries. For more information, see "HashJoin algorithms (JH, JHSP, JHFO, JHAP, JHO, JHPO)" on page 577, and "Recursive common table expressions" on page 435.

HashSemijoin algorithm (JHS)

HashSemijoin performs a semijoin between the left-hand side and the right-hand side. The right-hand side is only used to determine which rows from the left-hand side appear in the result. With HashSemijoin, the right-hand side is read to form an in-memory hash table which is subsequently probed by each row from the left-hand side. If a match is found, the left-hand side row is output to the result and the match process starts again for the next left-hand side row. At least one equality join condition must be present for HashSemijoin to be considered by the query optimizer. As with NestedLoopsSemijoin, HashSemijoin is used when the join's inputs include table expressions from an existentially-quantified (IN, SOME, ANY, EXISTS) nested query that has been rewritten as a join. HashSemijoin tends to outperform NestedLoopsSemijoin when the join condition includes inequalities, or if a suitable index does not exist to make indexed retrieval of the right-hand side sufficiently inexpensive.

As with HashJoin, HashSemijoin may revert to a nested loops semijoin strategy if there is insufficient cache memory to allow the operation to complete. Should this occur, a performance counter is incremented. You can read this monitor with the QueryLowMemoryStrategy database or connection property, or in the Query: Low Memory Strategies counter in the Windows Performance Monitor.

The amount of memory that can be used by a HashSemijoin operator is dependent upon the multiprogramming level of the server, and the number of active connections. See "Threading in SQL Anywhere" [*SQL Anywhere Server - Database Administration*], and "Configuring the database server's multiprogramming level" [*SQL Anywhere Server - Database Administration*].

Note
The Windows Performance Monitor may not be available on Windows Mobile.

For more information, see QueryLowMemoryStrategy in "Connection properties" [*SQL Anywhere Server - Database Administration*].

HashAntisemijoin algorithm (JHA)

HashAntisemijoin performs an anti-semijoin between the left-hand side and the right-hand side. The right-hand side is only used to determine which rows from the left-hand side appear in the result. With HashAntisemijoin, the right-hand side is read to form an in-memory hash table that is subsequently probed by each row from the left-hand side. Each left-hand side row is output only if it fails to match any row from the right-hand side. HashAntisemijoin is used when the join's inputs include table expressions from a quantified (NOT IN, ALL, NOT EXISTS) nested query that can be rewritten as an antijoin. HashAntisemijoin tends to outperform the evaluation of the search condition referencing the quantified query if a suitable index does not exist to make indexed retrieval of the right-hand side sufficiently inexpensive.

As with HashJoin, HashAntisemijoin may revert to a nested loops strategy if there is insufficient cache memory to allow the operation to complete. Should this occur, a performance counter is incremented. You can read this monitor with the QueryLowMemoryStrategy database or connection property, or in the Query: Low Memory Strategies counter in the Windows Performance Monitor.

The amount of memory that can be used by a HashAntisemijoin operator is dependent upon the multiprogramming level of the server, and the number of active connections. See "Threading in SQL Anywhere" [*SQL Anywhere Server - Database Administration*], and "Configuring the database server's multiprogramming level" [*SQL Anywhere Server - Database Administration*].

Note
The Windows Performance Monitor may not be available on Windows Mobile.

For more information, see QueryLowMemoryStrategy in "Connection properties" [*SQL Anywhere Server - Database Administration*].

MergeJoin algorithms (JM, JMFO, JMO)

MergeJoin reads two inputs that are both ordered by the join attributes. For each row of the left input, the algorithm reads all the matching rows of the right input by accessing the rows in sorted order.

If the inputs are not already ordered by the join attributes (perhaps because of an earlier merge join or because an index was used to satisfy a search condition), then the optimizer adds a sort to produce the correct row order. This sort adds cost to the merge join.

One advantage of MergeJoin, as compared to HashJoin, is that the cost of sorting can be amortized over several joins, if the merge joins are over the same attributes. The optimizer chooses MergeJoin over HashJoin if the sizes of the inputs are likely to be similar, or if it can amortize the cost of the sort over several operations.

NestedLoopsJoin algorithms (JNL, JNLFO, JNLO)

NestedLoopsJoin computes the join of its left and right-hand sides by completely reading the right-hand side for each row of the left-hand side. (The syntactic order of tables in the query does not matter because the optimizer chooses the appropriate join order for each block in the request.)

The optimizer may choose NestedLoopsJoin if the join condition does not contain an equality condition, or if the statement is being optimized with a first row optimization goal (that is, either the optimization_goal option is set to First-Row, or FASTFIRSTROW is specified as a table hint in the FROM clause.

Since NestedLoopsJoin reads the right-hand side many times, it is very sensitive to the cost of the right-hand side. If the right-hand side is an index scan or a small table, then the right-hand side can likely be computed using cached pages from previous iterations. However, if the right-hand side is a sequential table scan or an index scan that matches many rows, then the right-hand side needs to be read from disk many times. Typically, NestedLoopsJoin is less efficient than other join methods. However, NestedLoopsJoin can provide the first matching row quickly compared to join methods that must compute their entire result before returning.

NestedLoopsJoin is the only join algorithm that can provide sensitive semantics for queries containing joins. This means that sensitive cursors on joins can only be executed with NestedLoopsJoin.

See also

- "optimization_goal option" [*SQL Anywhere Server - Database Administration*]
- "FROM clause" [*SQL Anywhere Server - SQL Reference*]

NestedLoopsSemijoin algorithm (JNLS)

Similar to NestedLoopsJoin, NestedLoopsSemijoin joins its inputs by scanning the right-hand side for each row of the left-hand side. As with NestedLoopsJoin, the right-hand side may be read many times, so for larger inputs an index scan is preferable.

NestedLoopsSemijoin differs from NestedLoopsJoin in two respects. First, NestedLoopsSemijoin only outputs values from the left-hand side; the right-hand side is used only for restricting which rows of the left-hand side appear in the result. Second, NestedLoopsSemijoin stops each search of the right-hand side when the first match is encountered. NestedLoopsSemijoin can be used when inputs to the join include table expressions from an existentially-quantified (IN, SOME, ANY, EXISTS) nested query that has been rewritten as a join.

NestedLoopsAntisemijoin algorithm (JNLA)

Similar to the NestedLoopsJoin algorithm, NestedLoopsAntisemijoin joins its inputs by scanning the right-hand side for each row of the left-hand side. As with NestedLoopsJoin, the right-hand side may be read many times, so for larger inputs an index scan is preferable. NestedLoopsAntisemijoin differs from NestedLoopsJoin in that it only outputs values from the left-hand side; the right-hand side is used only for

restricting which rows of the left-hand side appear in the result. Specifically, values from the left-hand side are only included if they have no corresponding value on the right-hand side.

Duplicate elimination algorithms

A duplicate elimination operator produces an output that has no duplicate rows. Duplicate elimination nodes may be introduced by the optimizer, for example, when converting a nested query into a join.

For more information, see "HashDistinct algorithm (DistH)" on page 581, and "OrderedDistinct algorithm (DistO)" on page 581.

HashDistinct algorithm (DistH)

HashDistinct takes a single input and returns all distinct rows. HashDistinct does this by reading its input, and building an in-memory hash table. If an input row is found in the hash table, it is ignored; otherwise, it is written to a work table. If the input does not completely fit into the in-memory hash table, it is partitioned into smaller work tables, and processed recursively.

HashDistinct also:

- works very well if the distinct rows fit into an in-memory table, irrespective of the total number of rows in the input.
- uses a work table, and as such can provide insensitive or value sensitive semantics.
- returns a row when it finds one that has not previously been returned. However, the results of a hash distinct must be fully materialized before returning from the query. If necessary, the optimizer adds a work table to the execution plan to ensure this.
- locks the rows of its input.

The optimizer avoids generating access plans using the hash distinct algorithm if it detects that a low memory situation may occur during query execution. If HashDistinct executes in an environment where there is very little cache memory available, then it is not able to complete. In this case, HashDistinct discards its interim results, and an internal low memory approach is used instead.

The amount of memory that can be used by a HashDistinct operator is dependent upon the multiprogramming level of the server, and the number of active connections. See "Threading in SQL Anywhere" [*SQL Anywhere Server - Database Administration*], and "Configuring the database server's multiprogramming level" [*SQL Anywhere Server - Database Administration*].

OrderedDistinct algorithm (DistO)

If the input is ordered by all the columns, then OrderedDistinct can be used. This algorithm reads each row and compares it to the previous row. If it is the same, it is ignored; otherwise, it is output.

OrderedDistinct is effective if rows are already ordered (perhaps because of an index or a merge join); if the input is not ordered, the optimizer inserts a sort. No work table is used by the OrderedDistinct itself, but one is used by any inserted sort.

Grouping algorithms

Grouping algorithms compute a summary of their input. They are applicable only if the query contains a GROUP BY clause, or if the query contains aggregate functions (such as `SELECT COUNT(*) FROM T`).

For more information, see "HashGroupBy algorithm (GrByH)" on page 582, "OrderedGroupBy algorithm (GrByO)" on page 583, and "SingleRowGroupBy algorithm (GrByS)" on page 583.

HashGroupBy algorithm (GrByH)

HashGroupBy builds an in-memory hash table containing one row per group. As input rows are read, the associated group is looked up in the work table. The aggregate functions are updated, and the group row is rewritten to the work table. If no group record is found, a new group record is initialized and inserted into the work table.

HashGroupBy computes all the rows of its result before returning the first row, and can be used to satisfy a fully-sensitive or values-sensitive cursor. The results of the hash group by must be fully materialized before returning from the query. If necessary, the optimizer adds a work table to the execution plan to ensure this.

HashGroupBy can be executed in parallel.

HashGroupBy works very well if the groups fit into memory, regardless of the size of the input. If the hash table doesn't fit into memory, the input is partitioned into smaller work tables, which are recursively partitioned until they fit into memory. The optimizer avoids generating access plans using HashGroupBy if it detects that a low memory situation may occur during query execution. If there is not enough memory for the partitions, the optimizer discards the interim results from the HashGroupBy, and uses an internal low memory strategy instead.

The amount of memory that can be used by a HashGroupBy operator is dependent upon the multiprogramming level of the server, and the number of active connections. See "Threading in SQL Anywhere" [*SQL Anywhere Server - Database Administration*], and "Configuring the database server's multiprogramming level" [*SQL Anywhere Server - Database Administration*].

ClusteredHashGroupBy algorithm (GrByHClust)

Sometimes values in the grouping columns of the input table are clustered, so that similar values appear close together. For example, if a table contains a column that is always set to the current date, all rows with a single date tend to be relatively close within the table. ClusteredHashGroupBy exploits this clustering.

The optimizer may use ClusteredHashGroupBy when grouping tables that are significantly larger than the available memory. In particular, it is effective when the HAVING predicate returns only a small proportion of rows.

ClusteredHashGroupBy can lead to significant wasted work on the part of the optimizer if it is chosen in an environment where data is being updated at the same time that queries are being executed. ClusteredHashGroupBy is therefore most appropriate for OLAP workloads characterized by occasional batch-style updates and read-based queries. Set the optimization_workload option to OLAP to instruct the optimizer that it should include ClusteredHashGroupBy in the possibilities it investigates. See "optimization_workload option" [*SQL Anywhere Server - Database Administration*].

When creating an index or foreign key that can be used in an OLAP workload, specify the FOR OLAP WORKLOAD clause. This clause causes the database server to maintain a statistic used by ClusteredHashGroupBy regarding the maximum page distance between two rows within the same key. See "CREATE INDEX statement" [*SQL Anywhere Server - SQL Reference*], "CREATE TABLE statement" [*SQL Anywhere Server - SQL Reference*], and "ALTER TABLE statement" [*SQL Anywhere Server - SQL Reference*].

For more information about OLAP workloads, see "OLAP support" on page 445.

HashGroupBySets algorithm (GrByHSets)

A variant of HashGroupBy, HashGroupBySets is used when performing GROUPING SETS queries.

HashGroupBySets cannot be executed in parallel.

For more information, see "HashGroupBy algorithm (GrByH)" on page 582.

OrderedGroupBy algorithm (GrByO)

OrderedGroupBy reads an input that is ordered by the grouping columns. As each row is read, it is compared to the previous row. If the grouping columns match, then the current group is updated; otherwise, the current group is output and a new group is started.

OrderedGroupBySets algorithm (GrByOSets)

A variant of OrderedGroupBy, OrderedGroupBySets is used when performing GROUPING SETS queries. This algorithm requires that the input be sorted by grouping columns. See "OrderedGroupBy algorithm (GrByO)" on page 583.

SingleRowGroupBy algorithm (GrByS)

When no GROUP BY is specified, SingleRowGroupBy is used to produce a single row aggregate. A single group row is kept in memory and updated for each input row.

SortedGroupBySets algorithm (GrBySSets)

SortedGroupBySets is used when processing OLAP queries that contain GROUPING SETS.

Query expression algorithms

The query expression algorithms can be broken into the following categories:

- Except algorithms, which include MergeExcept and HashExcept
- Intersect algorithms, which include MergeIntersect and HashIntersect
- Union algorithms, which include Union, UnionAll, and RecursiveUnion

Except algorithms (EAH, EAM, EH, EM)

The SQL Anywhere query optimizer chooses between two physical implementations of the set difference SQL operator EXCEPT: a sort-based variant, MergeExcept (EM) and a hash-based variant, HashExcept (EH).

MergeExcept uses MergeJoin to compute the set difference between the two inputs through analyzing row matches in sorted order. Often, an explicit sort of the two inputs is required. Similarly, HashExcept uses HashAntisemijoin to compute the set difference between the two inputs, and a left outer hash join to compute the difference of the two inputs (EXCEPT ALL).

HashExcept may dynamically switch to a nested loops strategy if a memory shortage is detected. When this occurs, a performance counter is incremented. You can read this monitor with the QueryLowMemoryStrategy database or connection property, in the QueryLowMemoryStrategy statistic in the graphical plan (when run with statistics), or in the Query: Low Memory Strategies counter in the Windows Performance Monitor.

HashExcept is disabled on Windows Mobile in low memory situations.

EXCEPT, MergeExcept and HashExcept are coupled with one of the DISTINCT algorithms to ensure that the result does not contain duplicates. For EXCEPT ALL, HashExceptAll and MergeExceptAll are coupled with RowReplicate, which computes the correct number of duplicate rows in the result.

See also

- "EXCEPT statement" [*SQL Anywhere Server - SQL Reference*]
- "Performing set operations on query results with UNION, INTERSECT, and EXCEPT" on page 380
- "QueryLowMemoryStrategy connection property" [*SQL Anywhere Server - Database Administration*]
- "QueryLowMemoryStrategy database property" [*SQL Anywhere Server - Database Administration*]
- "Query: Low memory strategies statistic" on page 208

Intersect algorithms (IH, IM, IAH, IAM)

The SQL Anywhere query optimizer chooses between two physical implementations of the set intersection SQL operator INTERSECT: a sort-based variant, MergeIntersect (IM), and a hash-based variant, HashIntersect (IH).

MergeIntersect uses MergeJoin to compute the set intersection between the two inputs through analyzing row matches in sorted order. Often, an explicit sort of the two inputs is required. Similarly, HashIntersect uses HashJoin to compute the set and bag intersection between the two inputs (INTERSECT and INTERSECT ALL).

If necessary, HashIntersect may dynamically switch to a nested loops strategy if a memory shortage is detected. When this occurs, a performance counter is incremented. You can read this monitor with the QueryLowMemoryStrategy database or connection property, in the QueryLowMemoryStrategy statistic in the graphical plan (when run with statistics), or in the Query: Low Memory Strategies counter in the Windows Performance Monitor.

HashIntersect is disabled on Windows Mobile in low memory situations.

INTERSECT, MergeIntersect or HashIntersect are coupled with one of the DISTINCT algorithms to ensure that the result does not contain duplicates. For INTERSECT ALL operations, MergeIntersectAll and HashIntersectAll are coupled with RowReplicate, which computes the correct number of duplicate rows in the result.

See also

- "INTERSECT statement" [*SQL Anywhere Server - SQL Reference*]
- "Performing set operations on query results with UNION, INTERSECT, and EXCEPT" on page 380
- "QueryLowMemoryStrategy connection property" [*SQL Anywhere Server - Database Administration*]
- "QueryLowMemoryStrategy database property" [*SQL Anywhere Server - Database Administration*]
- "Query: Low memory strategies statistic" on page 208

RecursiveTable algorithm (RT)

A recursive table is a common table expression constructed as a result of a WITH clause in a query, where the WITH clause is used for recursive union queries. Common table expressions are temporary views that are known only within the scope of a single SELECT statement. See "Common table expressions" on page 429.

RecursiveUnion algorithm (RU)

RecursiveUnion is employed during the execution of recursive union queries. See "Recursive common table expressions" on page 435.

RowReplicate algorithm (RR)

RowReplicate is used during the execution of set operations such as EXCEPT ALL and INTERSECT ALL. It is a feature of such operations that the number of rows in the result set is explicitly related to the number of rows in the two sets being operated on. RowReplicate ensures that the number of rows in the result set is correct. See "Performing set operations on query results with UNION, INTERSECT, and EXCEPT" on page 380.

UnionAll algorithm (UA)

UnionAll reads rows from each of its inputs and outputs them, regardless of duplicates. This algorithm is used to implement UNION and UNION ALL statements. In the UNION case, a duplicate elimination algorithm such as HashDistinct or OrderedDistinct is needed to remove any duplicates generated by UnionAll.

See "HashDistinct algorithm (DistH)" on page 581, and "OrderedDistinct algorithm (DistO)" on page 581.

Sorting algorithms

Sorting algorithms are applicable when the query includes an ORDER BY clause, or when the query's execution strategy requires a total sort of its input.

For more information, see "Sort algorithm (Sort)" on page 586 and "UnionAll algorithm (UA)" on page 586.

Sort algorithm (Sort)

Sort reads its input into memory, sorts it in memory, and then outputs the results. If the input does not completely fit into memory, then several sorted runs are created and then merged together. Sort does not return any rows until it has read all the input rows. Sort locks its input rows.

If Sort executes in an environment where there is very little cache memory available, it may not be able to complete. In this case, Sort orders the remainder of the input using an indexed-based sort method. Input rows are read and inserted into a work table, and an index is built on the ordering columns of the work table. In this case, rows are read from the work table using a complex index scan. This indexed-based strategy is significantly slower. The optimizer avoids generating access plans using Sort if it detects that a low memory situation may occur during query execution. When the index-based strategy is needed due to low memory, a performance counter is incremented; you can read this monitor with the QueryLowMemoryStrategy property, or in the Query: Low Memory Strategies counter in the Windows Performance Monitor.

The amount of memory that can be used by a Sort operator is dependent upon the multiprogramming level of the server, and the number of active connections. See "Threading in SQL Anywhere" [*SQL Anywhere Server - Database Administration*], and "Configuring the database server's multiprogramming level" [*SQL Anywhere Server - Database Administration*].

Sort performance is affected by the size of the sort key, the row size, and the total size of the input. For large rows, it may be cheaper to use a VALUES SENSITIVE cursor. In that case, columns in the SELECT-

list are not copied into the work tables used by the sort. While Sort does not write output rows to a work table, the results of Sort must be materialized before rows are returned to the application. If necessary, the optimizer adds a work table to ensure this.

SortTopN algorithm (SrtN)

SortTopN is used for queries that contain a TOP N clause and an ORDER BY clause. It is an efficient algorithm for sorting only those rows required in the result set.

Subquery and function caching

When SQL Anywhere processes a subquery, it caches the result. This caching is done on a request-by-request basis; cached results are never shared by concurrent requests or connections. Should SQL Anywhere need to re-evaluate the subquery for the same set of correlation values, it can simply retrieve the result from the cache. In this way, SQL Anywhere avoids many repetitious and redundant computations. When the request is completed (the query's cursor is closed), SQL Anywhere releases the cached values.

As the processing of a query progresses, SQL Anywhere monitors the frequency with which cached subquery values are reused. If the values of the correlated variable rarely repeat, then SQL Anywhere needs to compute most values only once. In this situation, SQL Anywhere recognizes that it is more efficient to recompute occasional duplicate values, than to cache numerous entries that occur only once. So, the database server suspends the caching of this subquery for the remainder of the statement and proceeds to re-evaluate the subquery for each and every row in the outer query block.

SQL Anywhere also does not cache if the size of the dependent column is more than 255 bytes. In such cases, you may want to rewrite your query or add another column to your table to make such operations more efficient.

Function caching

Some built-in and user-defined functions are cached in the same way that subquery results are cached. This can result in a substantial improvement for expensive functions that are called during query processing with the same parameters. However, it may mean that a function is called fewer times than would otherwise be expected.

For a function to be cached, it must satisfy two conditions:

- It must always return the same result for a given set of parameters.
- It must have no side effects on the underlying data.

Functions that satisfy these conditions are called **deterministic** or **idempotent** functions. SQL Anywhere treats all user-defined functions as deterministic (unless they specifically declared NOT DETERMINISTIC at creation time). 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 semantics.

Built-in functions are treated as deterministic with a few exceptions. The RAND, NEWID, and GET_IDENTITY functions are treated as non-deterministic, and their results are not cached.

For more information about user-defined functions, see "CREATE FUNCTION statement" [*SQL Anywhere Server - SQL Reference*].

Miscellaneous algorithms

The following are additional algorithms that can be used in an access plan.

DecodePostings (DP)

A text index is stored in compressed chunks in a table. DecodePostings decodes positional information for the terms in the text index.

See also "Full text search" on page 288.

DerivedTable algorithm (DT)

A derived table is a SELECT statement included in the FROM clause of a query. The result set of the SELECT statement is logically treated as if it were a table. The query optimizer may also generate derived tables during query rewrites, for example in queries including the set operators UNION, INTERSECT, or EXCEPT. The graphical plan displays the name of the derived table and the list of columns that were computed.

A derived table embodies a portion of an access plan that cannot be merged, or flattened, into the other parts of the statement's access plan without changing the query's result. A derived table is used to enforce the semantics of derived tables specified in the original statement, and may appear in a plan due to query rewrite optimizations and a variety of other reasons, particularly when the query involves one or more outer joins.

For more information about derived tables, see "The FROM clause: Specifying tables" on page 265 and "FROM clause" [*SQL Anywhere Server - SQL Reference*].

Example

The following query has derived tables in its graphical plan:

```
SELECT EmployeeID FROM Employees
UNION ALL
SELECT DepartmentID FROM (
    SELECT TOP 5 DepartmentID
    FROM Departments
    ORDER BY DepartmentName DESC ) MyDerivedTable;
```

Exchange algorithm (Exchange)

Exchange is used to implement intra-query parallelism when processing a SELECT statement. An Exchange operator has two or more child subtrees, each of which executes in parallel. As each subtree executes, it fills up buffers of rows that are then consumed by the parent operator of the exchange. The result of an exchange is the union of the results from its children. Each child of an exchange uses one task, as does the parent. Therefore, a plan using a single exchange with two children requires three tasks to execute.

Exchange is only used when processing SELECT statements, and when intra-query parallelism is enabled.

For more information about parallelism, see "Threading in SQL Anywhere" [*SQL Anywhere Server - Database Administration*].

Filter algorithms (Filter, PreFilter)

Filters apply search conditions including any type of predicate, comparisons involving subselects, and EXISTS and NOT EXISTS subqueries (and other forms of quantified subqueries). The search conditions appear in the statement in the WHERE and HAVING clauses, and in the ON conditions of JOINS in the FROM clause.

The optimizer is free to simplify and alter the set of predicates in the search condition as it sees fit, and to construct an access plan that applies the conditions in an order different from the order specified in the original statement. Query rewrite optimizations may make substantial changes to the set of predicates evaluated in a plan.

In many situations, a predicate in a query may not result in the existence of Filter in the access plan. For example, various algorithms, such as IndexScan, have the ability to enforce the application of a predicate without the need for an explicit operator. For example, consider a BETWEEN predicate involving two literal constants, and the column referenced in the predicate is indexed. The BETWEEN predicate can be enforced by the lower and upper bounds of the index scan, and the plan for the query will not contain a Filter. Predicates that are join conditions also do not normally appear in an access plan as a filter.

PreFilter is the same as Filter, except that the expressions used in the predicates of a PreFilter do not depend on any table or view referenced in the query. As a simple example, the search condition in the clause `WHERE 1 = 2` can be evaluated as a prefilter.

See also

- "The WHERE clause: Specifying rows" on page 269
- "EXISTS search condition" [*SQL Anywhere Server - SQL Reference*]

Hash filter algorithms (HF, HFP)

A hash filter, sometimes referred to as a bloom filter, is a data structure that represents the distribution of values in a column or set of columns. The hash filter can be viewed as a (long) bit string where a 1 bit indicates the existence of a particular row, and a 0 bit indicates the lack of any row at that bit position. By hashing the values from a set of rows into bit positions in the filter, the database server can determine whether there is a matching row of that value (subject to the existence of hash collisions).

For example, consider the plan:

```
R<idx> *JH S<seq> JH* T<idx>
```

Here you are joining R to S and T. The database server reads all the rows of R before reading any row from T. If a hash filter is built using the rows of R returned by the index scan, then the database server can immediately reject rows of T that can not possibly join with R. This reduces the number of rows that must be stored in the second hash join.

A hash filter may be used in queries that satisfy both of the following conditions:

- An operation in the query reads its entire input before returning a row to later operations. For example, a hash join of two tables on a single column requires that all the relevant rows from one of the inputs be read to form the hash table for the join.
- A subsequent operation in the query's access plan refers to the rows in the result of that operation. For example, a second join on the same column as the first would use only those rows that satisfied the first join.

In this circumstance, the hash filter constructed as a result of the first join can substantially improve the performance of the second join. This is achieved by performing a lookaside operation into the hash filter's bit string to determine if any row has been previously processed successfully by the first join—if no such row exists, the hash table probe for the second join can be avoided entirely since the lack of a 1 bit in the hash filter signifies that the probe would fail to yield a match.

InList algorithm (IN)

InList is used when an IN-list predicate can be satisfied using an index. For example, in the following query, the optimizer recognizes that it can access the Employees table using its primary key index.

```
SELECT *
FROM Employees
WHERE EmployeeID IN ( 102, 105, 129 );
```

To do this, a join is built with a special in-list table on the left-hand side. Rows are fetched from the in-list table and used to probe the Employees table.

To use InList, each of the elements in the IN list predicate must be a constant, or a value that could be evaluated to a constant value at optimization time (such as CURRENT DATE, CURRENT TIMESTAMP, and non-deterministic system and user-defined functions), or a value that is constant within one execution of a query block (outer references). For example, the following query qualifies for InList.

```
SELECT *, (
   SELECT FIRST GivenName
    FROM Employees e
    WHERE e.DepartmentID IN ( 500, d.DepartmentID )
    ORDER BY e.DepartmentID )
FROM Departments d;
```

Multiple IN-list predicates can be satisfied using the same index.

OpenString algorithm (OpenString)

OpenString is used when the FROM clause of a SELECT statement contains an OPENSTRING clause. Rows are fetched from the BLOB or file specified in the OPENSTRING clause. See "FROM clause" [*SQL Anywhere Server - SQL Reference*].

An OpenString operator also appears in the plan for a LOAD TABLE statement.

ProcCall algorithm (PC)

ProcCall, used for procedures in a FROM clause, executes the procedure call and returns the rows in its result set. It is not able to fetch backwards and therefore appears below a work table if this is required by the cursor type.

Every time ProcCall is called, the database server notes the argument values, the number of rows returned, and the total time used to fetch all rows. This information is used by the optimizer to estimate the cost and cardinality of subsequent procedure calls. For each procedure, the database server maintains a moving average of the number of rows returned and a moving average of the total execution time. The database server also maintains a limited number of separate moving averages for specific argument values. This information is stored persistently in the stats column of the SYSPROCEDURE system table, in a binary format intended only for internal use.

For information about the restrictions on multiple result sets, and schema-matching requirements, see the procedure clause of the FROM clause, "FROM clause" [*SQL Anywhere Server - SQL Reference*].

See also

- "SYSPROCEDURE system view" [*SQL Anywhere Server - SQL Reference*]
- "How the optimizer uses procedure statistics" on page 544

RowConstructor algorithm (ROWS)

RowConstructor is a specialized operator that creates a virtual row for use as the input to other algorithms. RowConstructor is used in the following two ways:

- With an INSERT ... VALUES statement, the expressions referenced in the VALUES clause (typically literal constants and/or host variables) form a virtual row to be inserted. In this case, a row constructor appears in the graphical plan underneath an INSERT.
- Direct or indirect references to the system table SYS.DUMMY are transformed automatically to use RowConstructor, replacing the need for a table scan of SYS.DUMMY, and eliminating the need to latch the (single) page of the DUMMY table.

With short or long text plans, the plan string continues to contain a reference to the table SYS.DUMMY, even though RowConstructor was used instead of performing a table scan of SYS.DUMMY.

See also

- "DUMMY system table" [*SQL Anywhere Server - SQL Reference*]
- "INSERT statement" [*SQL Anywhere Server - SQL Reference*]
- "Reading execution plans" on page 592

RowLimit algorithm (RL)

RowLimit returns the first *n* rows of its input and ignores the remaining rows. Row limits are set by the TOP n or FIRST clause of the SELECT statement. See "SELECT statement" [*SQL Anywhere Server - SQL Reference*].

TermBreaker algorithm (TermBreak)

The term breaker algorithm used for full text searching. See the description for how the term breaker is used in "How to manage text configuration objects" on page 323.

Window algorithm (Window)

Window is used when evaluating OLAP queries that employ window functions. See "Window functions" on page 456.

Reading execution plans

An execution plan is the set of steps the database server uses to access information in the database related to a statement. The execution plan for a statement can be saved and reviewed, regardless of whether it was just optimized, whether it bypassed the optimizer, or whether its plan was cached from previous executions. A query execution plan may not correspond exactly to the syntax used in the original statement, and may use materialized views instead of the base tables explicitly specified in the query. However, the operations described in the execution plan are semantically equivalent to the original query.

You can view the execution plan in Interactive SQL or using SQL functions. You can choose to retrieve the execution plan in several different formats:

- Short text plan
- Long text plan
- Graphical plan
- Graphical plan with root statistics
- Graphical plan with full statistics
- UltraLite (short, long, or graphical)

You can also obtain plans for SQL queries with a particular cursor type by using the GRAPHICAL_PLAN and EXPLANATION functions. See "GRAPHICAL_PLAN function

[Miscellaneous]" [*SQL Anywhere Server - SQL Reference*], and "EXPLANATION function [Miscellaneous]" [*SQL Anywhere Server - SQL Reference*].

Additional reading

For more information about phases a statement goes through until it is executed, see "Query processing phases" on page 525.

For more information about the rules that the database server follows when rewriting your query, see:

- "Semantic query transformations" on page 528
- "Rewriting subqueries as EXISTS predicates" on page 538
- "Improving performance with materialized views" on page 555

For information about the algorithms and methods that the optimizer uses to implement your query, see "Query execution algorithms" on page 567.

For more information about how to access a graphical plan, see "Viewing graphical plans" on page 604.

For information about how to read execution plans, see "Reading text plans" on page 593, and "Reading graphical plans" on page 595.

Reading text plans

There are two types of text representation of a query execution plan: short and long. Use the SQL functions to access the text plan. See "Viewing short and long text plans" on page 595.

There is also a graphical version of the plan. See "Reading graphical plans" on page 595.

Short text plan

The short text plan is useful when you want to compare plans quickly. It provides the least amount of information of all the plan formats, but it provides it on a single line.

In the following example, the plan starts with `Work[Sort` because the ORDER BY clause causes the entire result set to be sorted. The Customers table is accessed by its primary key index, CustomersKey. An index scan is used to satisfy the search condition because the column Customers.ID is a primary key. The abbreviation JNL indicates that the optimizer chose a merge join to process the join between Customers and SalesOrders. Finally, the SalesOrders table is accessed using the foreign key index FK_CustomerID_ID to find rows where CustomerID is less than 100 in the Customers table.

```
SELECT EXPLANATION ('SELECT GivenName, Surname, OrderDate
FROM Customers JOIN SalesOrders
WHERE CustomerID < 100
ORDER BY OrderDate');
```

```
Work[ Sort[ Customers<CustomersKey> JNL
SalesOrders<FK_CustomerID_ID> ] ]
```

For more information about code words used in the plan, see "Execution plan abbreviations" on page 609.

Colons separate join strategies

The following command contains two **query blocks**: the outer select block referencing the SalesOrders and SalesOrderItems tables, and the subquery that selects from the Products table.

```
SELECT EXPLANATION ('SELECT *
FROM SalesOrders AS o
   KEY JOIN SalesOrderItems AS I
WHERE EXISTS
   (  SELECT *
      FROM Products p
      WHERE p.ID = 300 )');
```

```
o<seq> JNL i<FK_ID_ID> : p<ProductsKey>
```

Colons separate join strategies of the different query blocks. Short plans always list the join strategy for the main block first. Join strategies for other query blocks follow. The order of join strategies for these other query blocks may not correspond to the order of the query blocks in your statement, or to the order in which they execute.

For more information about the abbreviations used in a plan, see "Execution plan abbreviations" on page 609.

Long text plan

The long text plan provides a little more information than the short text plan, and provides information in a way that is easy to print and view without scrolling.

In the following example, the first line of the long text plan is `Plan[ Total Cost Estimate: 6.46e-005 ]` The word Plan indicates the start of a query block. The Total Cost Estimate is the optimizer estimated time, in milliseconds, for the execution of the plan. The Estimated Cache Pages is the estimated current cache size available for processing the statement.

The plan indicates that the results are sorted, and that a Nested Loops Join is used. On the same line as the join operator, there is either the word TRUE or the residual search condition and its selectivity estimate (which is evaluated for all the rows produced by the join operator). The IndexScan lines indicate that the Customers and SalesOrders tables are accessed via indexes CustomersKey and FK_CustomerID_ID respectively.

```
SELECT PLAN ('SELECT GivenName, Surname, OrderDate, Region, Country
FROM Customers JOIN SalesOrders ON ( SalesOrders.CustomerID = Customers.ID )
WHERE CustomerID < 100 AND ( Region LIKE ''Eastern''
      OR Country LIKE ''Canada'' )
ORDER BY OrderDate');

( Plan [ Total Cost Estimate: 6.46e-005, Costed Best Plans: 1, Costed Plans:
10, Optimization Time: 0.0011462,
Estimated Cache Pages: 348 ]
  ( WorkTable
    ( Sort
```



```
      ( NestedLoopsJoin
       ( IndexScan Customers CustomersKey[ Customers.ID < 100 : 0.0001% Index
| Bounded ] )
        ( IndexScan SalesOrders FK_CustomerID_ID[ Customers.ID =
SalesOrders.CustomerID : 0.79365% Statistics ]
          [  ( SalesOrders.CustomerID < 100 : 0.0001% Index | Bounded )
          AND  ( ( ((Customers.Country LIKE 'Canada' : 100% Computed)
          AND (Customers.Country = 'Canada' : 5% Guess))
          OR ((SalesOrders.Region LIKE 'Eastern' : 100% Computed)
          AND (SalesOrders.Region = 'Eastern' : 5% Guess)) ) : 100%
Guess )  ] )
       )
    )
  )
)
```

For more information about the abbreviations used in a plan, see "Execution plan abbreviations" on page 609.

Viewing short and long text plans

To view a short text plan (SQL)

1. Connect to a database as a user with DBA authority.
2. Execute the EXPLANATION function. See "EXPLANATION function [Miscellaneous]" [*SQL Anywhere Server - SQL Reference*].

To view a long text plan (SQL)

1. Connect to a database as a user with DBA authority.
2. Execute the PLAN function. See "PLAN function [Miscellaneous]" [*SQL Anywhere Server - SQL Reference*].

Reading graphical plans

The graphical plan feature in Interactive SQL displays the execution plan for a query in the **Plan Viewer** window. The execution plan consists of a tree of relational algebra operators which, starting at the leaves of the tree, consume the base inputs of the query (usually rows from a table) and process the rows from bottom to top, so that the root of the tree yields the final result. Nodes in this tree correspond to specific algebraic operators, though not all query evaluation performed by the server is represented by nodes. For example, the effects of subquery and function caching are not directly displayed in a graphical plan.

Nodes displayed in the graphical plan are different shapes that indicate the type of operation performed:

- Hexagons represent operations that materialize data.
- Trapezoids represent index scans.

- Rectangles with square corners represent table scans.
- Rectangles with round corners represent operations not listed above.

You can use a graphical plan to diagnose performance issues with specific queries. For example, the information in the plan can help you decide if a table requires an index to improve the performance of this specific query. You can save the graphical plan for a query for future reference by pressing the **Save** button in the **Plan Viewer**. SQL Anywhere graphical plans are saved with the extension *.saplan*.

Possible performance issues are identified by thick lines and red borders in the graphical plan. For example:

- Thicker lines between nodes in a plan indicate a corresponding increase in the number of rows processed. The presence of a thick line over a table scan may indicate that the creation of an index might be required.
- Red borders around a node indicate that the operation was expensive in comparison with the other operations in the execution plan.

Node shapes and other graphical components of the plan can be customized within Interactive SQL. See "Customizing the appearance of graphical plans" on page 603.

You can view either a graphical plan, a graphical plan with a summary or a graphical plan with detailed statistics. All three plans allow you to view the parts of the plan that are estimated to be the most expensive. Generating a graphical plan with statistics is more expensive because it provides the actual query execution statistics as monitored by the database server when the query is executed. Graphical plans with statistics permits direct comparison between the estimates used by the query optimizer in constructing the access plan with the actual statistics monitored during execution. Note, however, that the optimizer is often unable to estimate precisely a query's cost, so expect differences between the estimated and actual values.

To view a graphical plan, see "Viewing graphical plans" on page 604. Graphical plans are also available using the Application Profiling mode in Sybase Central. For more information about the Application Profiling features of Sybase Central, see "Application profiling" on page 158.

For more information about text plans, see "Reading text plans" on page 593.

Graphical plan with statistics

The graphical plan provides more information than the short or long text plans. The graphical plan with statistics, though more expensive to generate, provides the actual query execution statistics as monitored by the database server when the query is executed, and permits direct comparison between the estimates used by the optimizer in constructing the access plan with the actual statistics monitored during execution. Significant differences between actual and estimated statistics might indicate that the optimizer does not have enough information to correctly estimate the query's cost, which may result an inefficient execution plan.

To generate a graphical plan with statistics, the database server must execute the statement. The generation of a graphical plan for long-running statements might take a significant amount of time. If the

statement is an UPDATE, INSERT, or DELETE, only the read-only portion of the statement is executed; table modifications are not performed. However, if a statement contains user-defined functions, they are executed as part of the query. If the user-defined functions have side effects (for example, modifying rows, creating tables, sending messages to the console, and so on), these changes are made when getting the graphical plan with statistics. Sometimes you can undo these side effects by issuing a ROLLBACK statement after getting the graphical plan with statistics. See "ROLLBACK statement" [*SQL Anywhere Server - SQL Reference*].

Analyzing performance using the graphical plan with statistics

You can use the graphical plan with statistics to identify database performance issues. For detailed field descriptions of the graphical plan with statistics, see "Node Statistics field descriptions" on page 605, and "Optimizer Statistics field descriptions" on page 606.

Identifying query execution issues

You can display database options and other global settings that affect query execution for the root operator node.

Reviewing selectivity performance

The selectivity of a predicate (conditional expression) is the percentage of rows that satisfy the condition. The estimated selectivity of predicates provides the information on which the optimizer bases its cost estimates. Accurate selectivity estimates are critical for the proper operation of the optimizer. For example, if the optimizer mistakenly estimates a predicate to be highly selective (for example, a selectivity of 5%), but in reality, the predicate is much less selective (for example, 50%), then performance might suffer. Although selectivity estimates might not be precise, a significantly large error might indicate a problem.

If you determine that the selectivity information for a key part of your query is inaccurate, you can use CREATE STATISTICS to generate a new set of statistics for the column(s). In rare cases, you may want to supply explicit selectivity estimates, although this approach can introduce problems when you later update the statistics.

Selectivity statistics are not displayed if the query is determined to be a bypass query. For more information about bypass queries, see "How the optimizer works" on page 540, and "Explicit selectivity estimates" [*SQL Anywhere Server - SQL Reference*].

Indicators of poor selectivity occur in the following places:

- **RowsReturned, actual and estimated** **RowsReturned** is the number of rows in the result set. The **RowsReturned** statistic appears in the table for the root node at the top of the tree. If the estimated row count is significantly different from the actual row count, the selectivity of predicates attached to this node or to the subtree may be incorrect.

- **Predicate selectivity, actual and estimated** Look for the **Predicate** subheading to see predicate selectivities. For information about reading the predicate information, see "Viewing selectivity in the graphical plan" on page 601.

If the predicate is over a base column for which there is no histogram, executing a CREATE STATISTICS statement to create a histogram may correct the problem. See "CREATE STATISTICS statement" [*SQL Anywhere Server - SQL Reference*].

If selectivity error remains a problem, you may want to consider specifying a user estimate of selectivity along with the predicate in the query text.

- **Estimate source** The source of selectivity estimates is also listed under the Predicate subheading in the **Statistics** pane.

  When the source of a predicate selectivity estimate is **Guess**, the optimizer has no information to use to determine the filtering characteristics of that predicate, which may indicate a problem (such as a missing histogram). If the estimate source is **Index** and the selectivity estimate is incorrect, your problem may be that the index is unbalanced; you may benefit from defragmenting the index with the REORGANIZE TABLE statement. See "REORGANIZE TABLE statement" [*SQL Anywhere Server - SQL Reference*].

Reviewing cache performance

If the number of cache reads (**CacheRead** field) and cache hits (**CacheHits** field) are the same, then all the objects processed for this SQL statement are resident in cache. When cache reads are greater than cache hits, it indicates that the database server is reading table or index pages from disk as they are not already resident in the server's cache. In some circumstances, such as hash joins, this is expected. In other circumstances, such as nested loops joins, a poor cache-hit ratio might indicate there is insufficient cache (buffer pool) to permit the query to execute efficiently. In this situation, you might benefit from increasing the server's cache size.

For more information about cache management, see "Use the cache to improve performance" on page 226.

Identifying ineffective indexes

It is often not obvious from query execution plans whether indexes help improve performance. Some of the scan-based algorithms used in SQL Anywhere provide excellent performance for many queries without using indexes.

For more information about indexes and performance, see "Use indexes effectively" on page 225 and "Index Consultant" on page 164.

Identifying data fragmentation problems

The **Runtime** and **FirstRowRunTime** actual and estimated values are provided in the root node statistics. Only **RunTime** appears in the **Subtree Statistics** section if it exists for that node.

The interpretation of **RunTime** depends on the statistics section in which it appears. In **Node Statistics**, **RunTime** is the cumulative time the corresponding operator spent during execution *for this node alone*. In **Subtree Statistics**, **RunTime** represents the total execution time spent for the entire operator subtree immediately beneath this node. So, for most operators **RunTime** and **FirstRowRunTime** are independent measures that should be separately analyzed.

FirstRowRunTime is the time required to produce the first row of the intermediate result of this node.

If a node's **RunTime** is greater than expected for a table scan or index scan, you may improve performance by executing the REORGANIZE TABLE statement. You can use the sa_table_fragmentation() and the sa_index_density() system procedures to determine whether the table or index are fragmented.

For more information, see "REORGANIZE TABLE statement" [*SQL Anywhere Server - SQL Reference*] and "Reduce table fragmentation" on page 215.

For more information about code words used in the plan, see "Execution plan abbreviations" on page 609.

Viewing detailed graphical plan node information

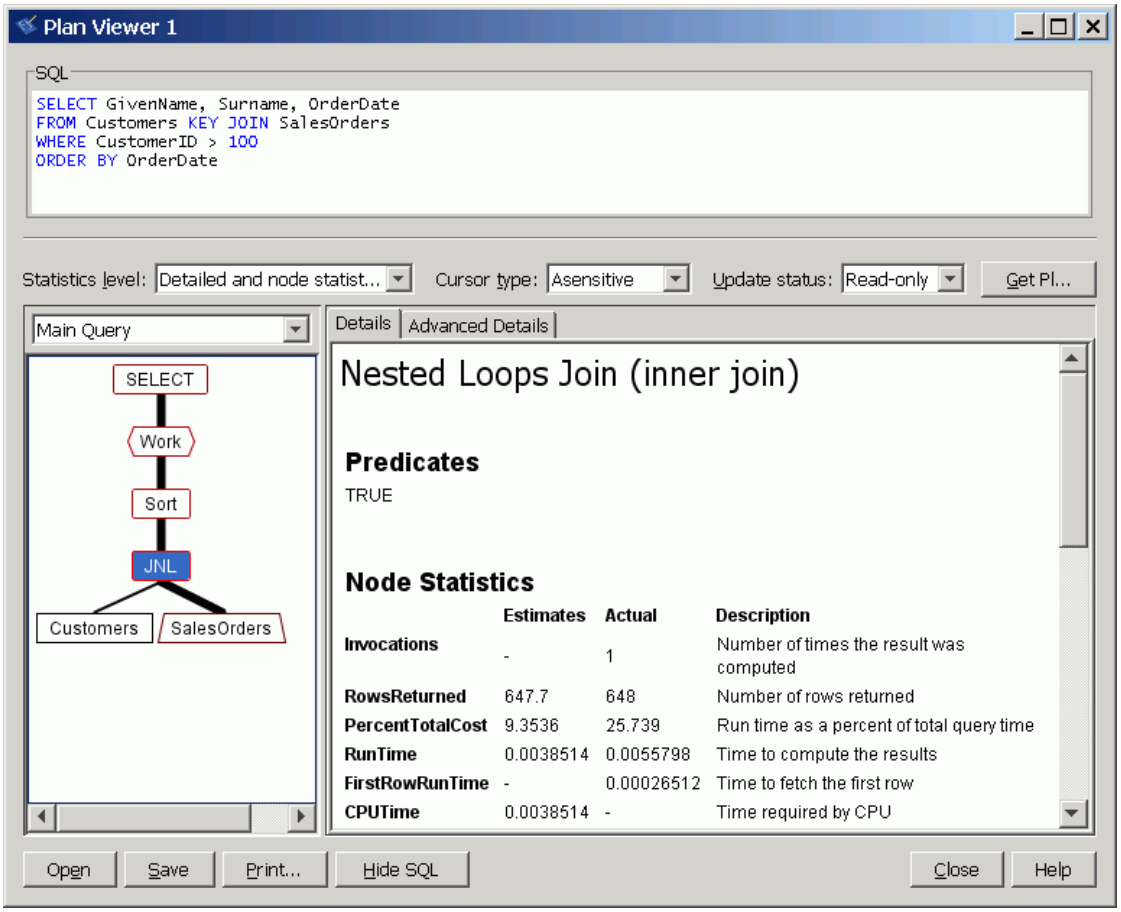
To view detailed node information in the graphical plan, in the left pane click the node in the graphical diagram. Details about the node appear on the right in the **Details** and **Advanced Details** panes. In the **Details** pane, statistics for the node appear in three sections:

- **Node Statistics**
- **Subtree Statistics**
- **Optimizer Statistics**

Node statistics are statistics related to the execution of the specific node. Leaf nodes have a **Details** pane that displays estimated and actual statistics for the operator. When a leaf node appears on the right side of a parent node, you can fetch rows from the parent operator multiple times. For example, with a nested loop join the leaf node (a sequential, index, or RowID scan node) contains both per-invocation (average) and cumulative actual run-time statistics.

When a node is not a leaf node it consumes intermediate result(s) from other nodes and the **Details** pane displays the estimated and actual cumulative statistics for this node's entire subtree in the **Subtree Statistics** section. Optimizer statistic information representing the entire SQL request is present only for root nodes. Optimizer statistics values are related specifically to the optimization of the statement, and include values such as the optimization goal setting, the optimization level setting, the number of plans considered, and so on.

In the example shown below, the **nested loops join (JNL)** node is selected and the information displayed in the right pane pertains only to that node. For example, the **Predicates** description is `TRUE`, indicating that a predicate is not applied. If you click the **Customers** node, the **Predicate** value changes to `Customers.ID > 100 : 100% Index; true 126/126 100%.`



The information displayed in the **Advanced Details** pane is dependent on the specific operator. For root nodes, the **Advanced Details** pane contains the setting of all connection options in effect when the query was optimized. With other node types, the **Advanced Details** pane might contain information about which indexes or materialized views were considered for the processing of the particular node.

To obtain context-sensitive help for each node in the graphical plan, right-click the node and choose **Help**.

For more information about the abbreviations used in the plan, see "Execution plan abbreviations" on page 609.

Note
If a query is recognized as a bypass query, some optimization steps are bypassed and neither the **Query Optimizer** section nor the **Predicate** section appear in the graphical plan. For more information about bypassed queries, see "How the optimizer works" on page 540.

See also

- "Reading graphical plans" on page 595
- "Viewing graphical plans" on page 604
- "Reading execution plans" on page 592
- "Node Statistics field descriptions" on page 605
- "Optimizer Statistics field descriptions" on page 606

Viewing selectivity in the graphical plan

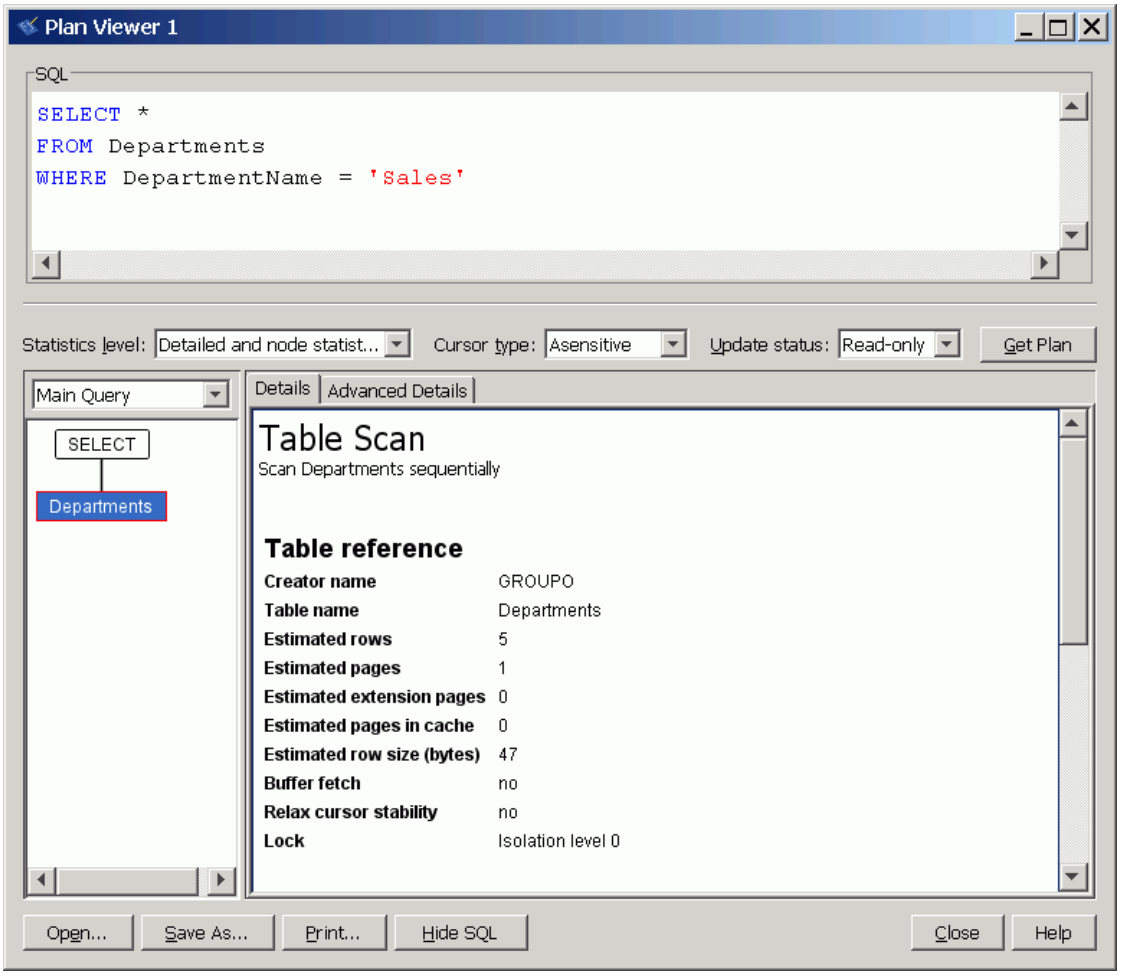
In the example shown below, the selected node represents a scan of the Departments table, and the statistics pane shows the **Predicate** as the search condition, its selectivity estimation, and its real selectivity.

In the **Details** pane, statistics about an individual node are divided into three sections: **Node Statistics**, **Subtree Statistics**, and **Optimizer Statistics**.

Node statistics pertain to the execution of this specific node. If the node is not a leaf node in the plan, and therefore consumes an intermediate result(s) from other nodes, the **Details** pane shows a **Subtree Statistics** section that contains estimated and actual cumulative statistics for this node's entire subtree. Optimizer statistics information is present only for root nodes, which represent the entire SQL request.

Selectivity information may not be displayed for bypass queries. For more information about bypass queries, see "How the optimizer works" on page 540.

The access plan depends on the statistics available in the database, which, in turn, depends on what queries have previously been executed. You may see different statistics and plans from those shown here.



This predicate description is

```
Departments.DepartmentName = 'Sales' : 20% Column; true 1/5 20%
```

This can be read as follows:

- `Departments.DepartmentName = 'Sales'` is the predicate.
- `20%` is the optimizer's estimate of the selectivity. That is, the optimizer is basing its query access selection on the estimate that 20% of the rows satisfy the predicate.

  This is the same output as is provided by the ESTIMATE function. For more information, see "ESTIMATE function [Miscellaneous]" [*SQL Anywhere Server - SQL Reference*].
- `Column` is the source of the estimate. This is the same output as is provided by the ESTIMATE_SOURCE function. For a complete list of the possible sources of selectivity estimates, see "ESTIMATE_SOURCE function [Miscellaneous]" [*SQL Anywhere Server - SQL Reference*].

- `true 1/5 20%` is the actual selectivity of the predicate during execution. The predicate was evaluated five times, and was true once, so its real selectivity is 20%.

  If the actual selectivity is very different from the estimate, and if the predicate was evaluated a large number of times, the incorrect estimates could cause a significant problem with query performance. Collecting statistics on the predicate may improve performance by giving the optimizer better information on which to base its choices.

- For more information on selectivity estimate sources used by the optimizer, see "Selectivity estimate sources" on page 542.

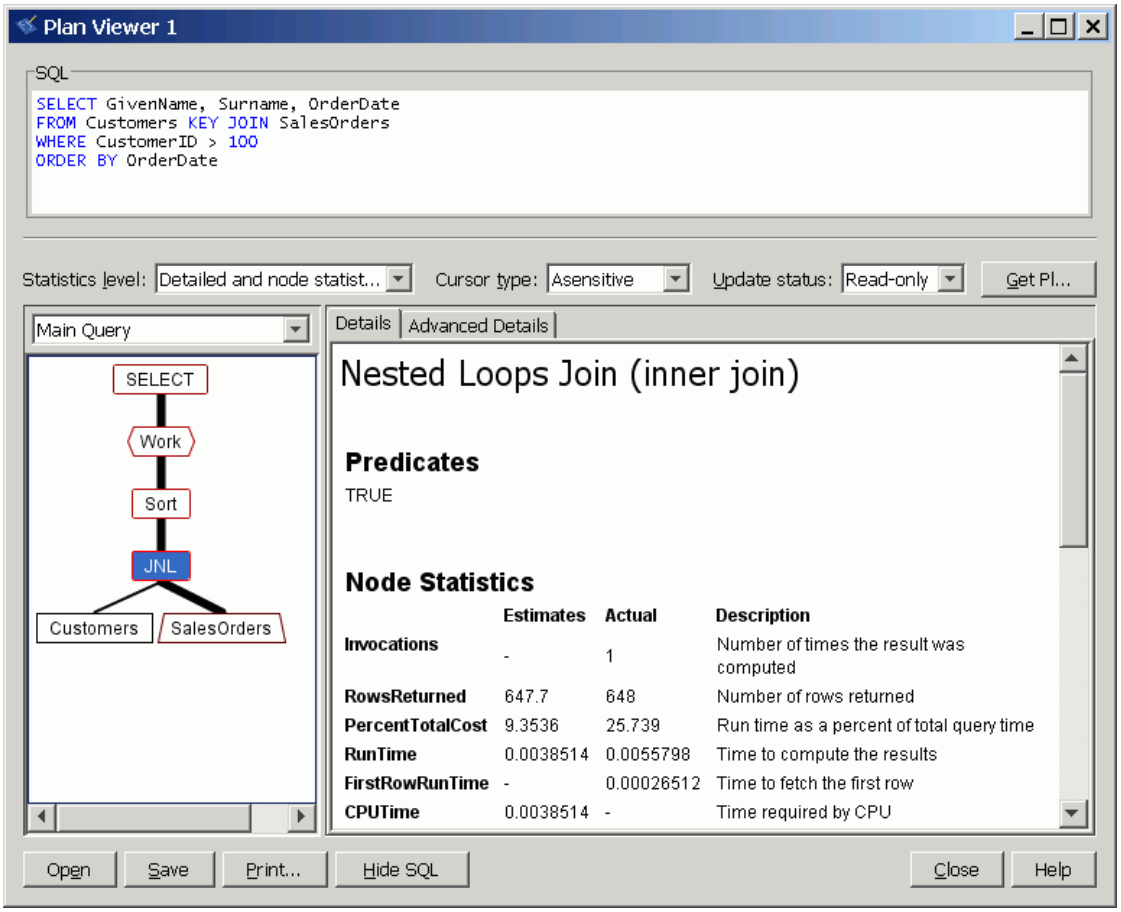
Note
If you select the graphical plan, but not the graphical plan with statistics, the final two statistics are not displayed.

Customizing the appearance of graphical plans

After executing the graphical plan you can customize the appearance of items in the plan. To change the appearance of the graphical plan, right-click the plan in the lower left pane of the Interactive SQL Plan Viewer, select **Customize**, and change the settings. Your changes are applied to subsequent graphical plans that are displayed.

To print a graphical plan, right-clicking the plan and choose **Print**.

Following is a query presented with its corresponding graphical plan. The diagram is in the form of a tree, indicating that each node requests rows from the nodes beneath it.



Viewing graphical plans

Use either Interactive SQL or the GRAPHICAL_PLAN function to view graphical plans. The GRAPHICAL_PLAN function displays a graphical plan in XML format, as a string. To access text plans, see “Reading text plans” on page 593.

To view a graphical plan (Interactive SQL)

1. Start Interactive SQL and connect to the SQL Anywhere database.
2. Choose **Tools** » **Plan Viewer** (or press Shift+F5).
3. Type a statement in the **SQL** pane.
4. Select a **Statistics level**, a **Cursor type** and an **Update status**.
5. Click **Get Plan**.

To view a graphical plan (SQL)

1. Connect to a database as a user with DBA authority.
2. Execute the GRAPHICAL_PLAN function. See "GRAPHICAL_PLAN function [Miscellaneous]" [*SQL Anywhere Server - SQL Reference*].

See also

- "Using the Plan Viewer to view graphical plans" [*SQL Anywhere Server - Database Administration*]
- "Execution plan abbreviations" on page 609

Node Statistics field descriptions

Below are descriptions of the fields displayed in the **Node Statistics** section of a graphical plan.

Field	Description
CacheHits	The total number of cache read requests by this operator which were satisfied by the buffer pool that did not require a disk read operation.
CacheRead	Total number of attempts made by this operator to read a page of the database file, typically for table and/or index pages.
CPUTime	The CPU time incurred by the processing algorithm represented by this node.
DiskRead	The cumulative number of pages that have been read from disk as a result of this node's processing.
DiskReadTime	The cumulative elapsed time required to perform disk reads for database pages required by this node for processing.
DiskWrite	The commutative number of pages that have been written to disk as a result of this node's processing.
DiskWriteTime	The cumulative elapsed time required to perform disk writes for database pages as required by this node's processing algorithm.
FirstRowRunTime	The **FirstRowRunTime** value is the actual elapsed time required to produce the first row of the intermediate result of this node.
Invocations	The number of times the node was called to compute a result, and return that result to the parent node. Most nodes are called only once. However, if the parent of a scan node is a nested loop join, then the node might be executed multiple times, and could possibly return a different set of rows after each invocation.
PercentTotalCost	The **RunTime** spent computing the result within this particular node, expressed as a percentage of the total RunTime for the statement.

Field	Description
QueryMemMaxUseful	The estimated amount of query memory that is expected to be used for this particular operator. If the actual amount of query memory used, which is reported as the **Actual** statistic, differs significantly then it may indicate a potential problem with result set size estimation by the query optimizer. A probable cause of this estimation error is inaccurate or missing predicate selectivity estimates.
RowsReturned	The number of rows returned to the parent node as a result of processing the request. **RowsReturned** is often, but not necessarily, identical to the number of rows in the (possibly derived) object represented by that node. Consider a leaf node that represents a base table scan. It is possible for the **RowsReturned** value to be smaller or larger than the number of rows in the table. **RowsReturned** are smaller if the parent node fails to request all the table's rows in computing the final result. **RowsReturned** may be greater in a case such as a GROUP BY GROUPING SETS query, where the parent Group By Hash Grouping Sets node requires multiple passes over the input to compute the different groups. A significant difference between the estimated rows returned and the actual number returned could indicate that the optimizer might be operating with poor selectivity information.
RunTime	This value is a measure of wall clock time, including waits for input/output, row locks, table locks, internal server concurrency control mechanisms, and actual runtime processing. The interpretation of **RunTime** depends on the statistics section in which it appears. In Node Statistics, **RunTime** is the cumulative time the node's corresponding operator spent during execution for this node alone. Both estimated and actual values for this statistic appear in the Node Statistics section. If a node's **RunTime** is greater than expected for a table scan or index scan, then further analysis may help pinpoint the problem. The query may be contending for shared resources and may block as a result; you can monitor blocked connections using the sa_locks() system procedure. As another example, the database page layout on the disk may be suboptimal, or a table may suffer from internal page fragmentation. You may improve performance by executing the REORGANIZE TABLE statement. You can use the sa_table_fragmentation() and the sa_index_density() system procedures to determine whether the table or index are fragmented.

Optimizer Statistics field descriptions

Below are descriptions of the fields displayed in the **Optimizer Statistics** section of a graphical plan. **Optimizer Statistics** provide information about the state of the database server and about the optimization of the selected statement.

Field	Description
Optimization Method	The algorithm used to choose an execution strategy. Values returned: • Bypass costed • Bypassed costed simple • Bypass heuristic • Bypassed then optimized • Optimized • Reused • Reused (simple)
Costed Best Plans	When the query optimizer enumerates different query execution strategies, it tracks the number of times it finds a strategy whose estimated cost is cheaper than the best strategy found before the current one. It is difficult to predict how often this will occur for any particular query, but a lower number indicates significant pruning of the search space by the optimizer's algorithms, and, typically, faster optimization times. Since the optimizer starts the enumeration process at least once for each query block in the given statement, **Costed Best Plans** represents the cumulative count. See "How the optimizer works" on page 540. If the values for **Costed Best Plans**, **Costed Plans**, and **Optimization** time are 0, then the statement was not optimized by the SQL Anywhere optimizer. Instead, the database server bypassed the statement and generated the execution plan without optimizing the statement, or the plan for the statement was cached. See "Query processing phases" on page 525.
Costed Plans	The number of different access plans considered by the optimizer for this request whose costs were partially or fully estimated. As with **Costed Best Plans**, smaller values normally indicate faster optimization times and larger values indicate more complex SQL queries. If the values for **Costed Best Plans**, **Costed Plans**, and **Optimization Time** are 0, then the statement was not optimized. Instead, the database server bypassed the statement and generated the execution plan without optimizing the statement. See "Query processing phases" on page 525.
Optimization Time	The elapsed time spent optimizing the statement. If the values for **Costed Best Plans**, **Costed Plans**, and **Optimization Time** are 0, then the statement was not optimized. Instead, the database server bypassed the statement and generated the execution plan without optimizing the statement. See "Query processing phases" on page 525.

Field	Description
Estimated Cache Pages	The estimated current cache size available for processing the statement. To reduce inefficient access plans, the optimizer assumes that one-half of the current cache size is available for processing the selected statement.
CurrentCacheSize	The database server's cache size in kilobytes at the time of optimization.
QueryMemMaxUseful	The number of pages of query memory that are useful for this request. If the number is zero, then the statement's execution plan contains no memory-intensive operators and is not subject to control by the server's memory governor. See "The memory governor" on page 549.
QueryMemNeedsGrant	Indicates whether the memory governor must grant memory to one or more memory-intensive query execution operators that are present in this request's execution strategy. See "The memory governor" on page 549.
QueryMemLikelyGrant	The estimated number of pages from the query memory pool that would be granted to this statement if it were executed immediately. This estimate can vary depending on the number of memory-intensive operators in the plan, the database server's multiprogramming level, and the number of concurrently-executing memory-intensive requests. See "The memory governor" on page 549.
QueryMemPages	The total amount of memory in the query memory pool that is available for memory-intensive query execution algorithms for all connections, expressed as a number of pages. See "The memory governor" on page 549.
QueryMemActiveMax	The maximum number of tasks that can actively use query memory at any particular time. See "The memory governor" on page 549.
QueryMemActiveEst	The database server's estimate of the steady state average of the number of tasks actively using query memory. See "The memory governor" on page 549.
isolation_level	The isolation level of the statement. The isolation level of the statement may differ from other statements in the same transaction, and may be further overridden for specific base tables through the use of hints in the FROM clause. See "isolation_level option" [*SQL Anywhere Server - Database Administration*].
optimization_goal	Indicates if query processing is optimized for returning the first row quickly, or minimizing the cost of returning the complete result set. See "optimization_goal option" [*SQL Anywhere Server - Database Administration*].
optimization_level	Controls amount of effort made by the query optimizer to find an access plan. See "optimization_level option" [*SQL Anywhere Server - Database Administration*].

Field	Description
optimization_workload	The **Mixed** or **OLAP** value of the optimization_workload setting. See "optimization_workload option" [*SQL Anywhere Server - Database Administration*].
max_query_tasks	Maximum number of tasks that may be used by a parallel execution plan for a single query. See "max_query_tasks option" [*SQL Anywhere Server - Database Administration*].
user_estimates	Controls whether to respect or ignore user estimates that are specified in individual predicates in the query text. See "user_estimates option" [*SQL Anywhere Server - Database Administration*].

Execution plan abbreviations

Following are the abbreviations that you see in execution plans.

Short text plan	Long text plan	Additional information
	Costed Best Plan	The optimizer generates and costs access plans for a given query. During this process the current best plan maybe replaced by a new best plan found to have a lower cost estimate. The last best plan is the execution plan used to execute the statement. Costed Best Plans indicates the number of times the optimizer found a better plan than the current best plan. A low number indicates that the best plan was determined early in the enumeration process. Since the optimizer starts the enumeration process at least once for each query block in the given statement, Costed Best Plans represents the cumulative count. See "How the optimizer works" on page 540.
	Costed Plans	Many plans generated by the optimizer are found to be too expensive compared to the best plan found so far. Costed Plans represents the number of partial or complete plans the optimizer considered during the enumeration processes for a given statement.
DELETE	Delete	The root node of a DELETE operation. See "DELETE statement" [*SQL Anywhere Server - SQL Reference*].
DistH	HashDistinct	See "HashDistinct algorithm (DistH)" on page 581.
DistO	OrderedDistinct	See "OrderedDistinct algorithm (DistO)" on page 581.

Short text plan	Long text plan	Additional information
DP	Decode-Postings	See “DecodePostings (DP)” on page 588.
DT	DerivedT-able	See “DerivedTable algorithm (DT)” on page 588.
EAH	HashEx-ceptAll	See “Except algorithms (EAH, EAM, EH, EM)” on page 584.
EAM	MergeEx-ceptAll	See “Except algorithms (EAH, EAM, EH, EM)” on page 584.
EH	HashEx-cept	See “Except algorithms (EAH, EAM, EH, EM)” on page 584.
EM	MergeAc-cept	See “Except algorithms (EAH, EAM, EH, EM)” on page 584.
Exchange	Exchange	See “Exchange algorithm (Exchange)” on page 588.
Filter	Filter	See “Filter algorithms (Filter, PreFilter)” on page 589.
GrByH	Hash-GroupBy	See “HashGroupBy algorithm (GrByH)” on page 582.
GrByHClust	Hash-GroupBy-Clustered	See “ClusteredHashGroupBy algorithm (GrByHClust)” on page 582.
GrByHSets	Hash-GroupBy-Sets	See “HashGroupBySets algorithm (GrByHSets)” on page 583.
GrByO	Ordered-GroupBy	See “OrderedGroupBy algorithm (GrByO)” on page 583.
GrByOSets	Ordered-GroupBy-Sets	See “OrderedGroupBySets algorithm (GrByOSets)” on page 583.
GrByS	Single-Row-GroupBy	See “SingleRowGroupBy algorithm (GrByS)” on page 583.

Short text plan	Long text plan	Additional information
GrBySSets	Sorted-GroupBy-Sets	See “SortedGroupBySets algorithm (GrBySSets)” on page 584.
HF	HashFilter	See “Hash filter algorithms (HF, HFP)” on page 589.
HFP	Parallel-HashFilter	See “Hash filter algorithms (HF, HFP)” on page 589.
HTS	HashTa-bleScan	See “HashTableScan method (HTS)” on page 574.
IAH	HashIn-tersectAll	See “Intersect algorithms (IH, IM, IAH, IAM)” on page 585.
IAM	MergeIn-tersectAll	See “Intersect algorithms (IH, IM, IAH, IAM)” on page 585.
IH	HashIn-tersect	See “Intersect algorithms (IH, IM, IAH, IAM)” on page 585.
IM	MergeIn-tersect	See “Intersect algorithms (IH, IM, IAH, IAM)” on page 585.
IN	InList	See “InList algorithm (IN)” on page 590.
table-name<in-dex-name>	Index-Scan, Par-allelIn-dexScan	In a graphical plan, an index scan appears as an index name in a trapezoid. See “IndexScan method” on page 571.
INSENSI-TIVE	Insensi-tive	See “Intersect algorithms (IH, IM, IAH, IAM)” on page 585.
INSERT	Insert	Root node of an insert operation. See “INSERT statement” [*SQL Any-where Server - SQL Reference*].
IO	IndexOn-lyScan, ParallelIn-dexOnly-Scan	See “IndexOnlyScan method (IO)” on page 572, and “ParallelIndexScan method” on page 573.
JH	HashJoin	See “HashJoin algorithms (JH, JHSP, JHFO, JHAP, JHO, JHPO)” on page 577.

Short text plan	Long text plan	Additional information
JHS	HashSemijoin	See “HashSemijoin algorithm (JHS)” on page 578.
JHSP	ParallelHashSemijoin	See “HashJoin algorithms (JH, JHSP, JHFO, JHAP, JHO, JHPO)” on page 577.
JHFO	Full Outer HashJoin	See “HashJoin algorithms (JH, JHSP, JHFO, JHAP, JHO, JHPO)” on page 577.
JHA	HashAntisemijoin	See “HashAntisemijoin algorithm (JHA)” on page 579.
JHAP	ParallelHashAntisemijoin	See “HashJoin algorithms (JH, JHSP, JHFO, JHAP, JHO, JHPO)” on page 577.
JHO	Left Outer HashJoin	See “HashJoin algorithms (JH, JHSP, JHFO, JHAP, JHO, JHPO)” on page 577.
JHP	ParallelHashJoin	See “HashJoin algorithms (JH, JHSP, JHFO, JHAP, JHO, JHPO)” on page 577.
JHPO	ParallelLeftOuterHashJoin	See “HashJoin algorithms (JH, JHSP, JHFO, JHAP, JHO, JHPO)” on page 577.
JHR	RecursiveHashJoin	See “RecursiveHashJoin algorithm (JHR)” on page 578.
JHRO	RecursiveLeftOuterHashJoin	See “RecursiveLeftOuterHashJoin algorithm (JHRO)” on page 578.
JM	MergeJoin	See “MergeJoin algorithms (JM, JMFO, JMO)” on page 579.
JMFO	Full Outer MergeJoin	See “MergeJoin algorithms (JM, JMFO, JMO)” on page 579.

Short text plan	Long text plan	Additional information
JMO	Left Outer Merge-Join	See “MergeJoin algorithms (JM, JMFO, JMO)” on page 579.
JNL	Nested-LoopsJoin	See “NestedLoopsJoin algorithms (JNL, JNLFO, JNLO)” on page 580.
JNLA	Nested-LoopsAn-tisemijoin	See “NestedLoopsAntisemijoin algorithm (JNLA)” on page 580.
JNLFO	Full Outer Nested-LoopsJoin	See “NestedLoopsJoin algorithms (JNL, JNLFO, JNLO)” on page 580.
JNLO	Left Outer Nested-LoopsJoin	See “NestedLoopsJoin algorithms (JNL, JNLFO, JNLO)” on page 580.
JNLS	Nested-LoopsSe-mijoin	See “NestedLoopsSemijoin algorithm (JNLS)” on page 580.
KEYSET	Keyset	Indicates a keyset-driven cursor. See “SQL Anywhere cursors” [*SQL Anywhere Server - Programming*].
LOAD	Load	Root node of a load operation. See “LOAD TABLE statement” [*SQL Anywhere Server - SQL Reference*].
MultiIdx	Multi-pleIndexS-can	See “MultipleIndexScan method (MultIdx)” on page 573.
OpenString	Open-String	See “OpenString algorithm (OpenString)” on page 591.
	Optimiza-tion Time	The total time spent by the optimizer during all enumeration processes for a given statement.
PC	ProcCall	Procedure call (table function). See “ProcCall algorithm (PC)” on page 591.
PreFilter	PreFilter	See “Filter algorithms (Filter, PreFilter)” on page 589.
RL	RowLimit	See “RowLimit algorithm (RL)” on page 592.

Short text plan	Long text plan	Additional information
ROWID	RowIdScan	In a graphical plan, a row ID scan appears as a table name in a rectangle. See "RowIdScan method (ROWID)" on page 575.
ROWS	RowConstructor	See "RowConstructor algorithm (ROWS)" on page 591.
RR	RowReplicate	See "RowReplicate algorithm (RR)" on page 585.
RT	RecursiveTable	See "RecursiveTable algorithm (RT)" on page 585.
RU	RecursiveUnion	See "RecursiveUnion algorithm (RU)" on page 585.
SELECT	Select	Root node of a SELECT operation. See "SELECT statement" [*SQL Anywhere Server - SQL Reference*].
seq	TableScan, ParallelTableScan	In a graphical plan, table scans appear as a table name in a rectangle. See "TableScan method (seq)" on page 573, and "ParallelTableScan method" on page 574.
Sort	Sort	Indexed or merge sort. See "Sort algorithm (Sort)" on page 586.
SrtN	SortTopN	See "SortTopN algorithm (SrtN)" on page 587.
TermBreak	TermBreak	The full text search TermBreaker algorithm. See "How to alter a text index" on page 344.
UA	UnionAll	See "UnionAll algorithm (UA)" on page 586.
UPDATE	Update	The root node of an UPDATE operation. See "UPDATE statement" [*SQL Anywhere Server - SQL Reference*].
Window	Window	See "Window algorithm (Window)" on page 592.
Work	Work table	An internal node that represents an intermediate result.

Common statistics used in the plan

The following statistics are actual, measured amounts.

Statistic	Explanation
Invocations	Number of times a row was requested from the sub tree.
RowsReturned	Number of rows returned for the current node.
RunTime	Time required for execution of the sub-tree, including time for children.
CacheHits	Number of successful reads of the cache.
CacheRead	Number of database pages that have been looked up in the cache.
CacheReadTable	Number of table pages that have been read from the cache.
CacheReadIndLeaf	Number of index leaf pages that have been read from the cache.
CacheReadIndInt	Number of index internal node pages that have been read from the cache.
DiskRead	Number of pages that have been read from disk.
DiskReadTable	Number of table pages that have been read from disk.
DiskReadIndLeaf	Number of index leaf pages that have been read from disk.
DiskReadIndInt	Number of index internal node pages that have been read from disk.
DiskWrite	Number of pages that have been written to disk (work table pages or modified table pages).
IndAdd	Number of entries that have been added to indexes.
IndLookup	Number of entries that have been looked up in indexes.
FullCompare	Number of comparisons that have been performed beyond the hash value in an index.

Common estimates used in the plan

Statistic	Explanation
EstRowCount	Estimated number of rows that the node will return each time it is invoked.
AvgRowCount	Average number of rows returned on each invocation. This is not an estimate, but is calculated as RowsReturned / Invocations. If this value is significantly different from EstRowCount, the selectivity estimates may be poor.
EstRunTime	Estimated time required for execution (sum of EstDiskReadTime, EstDiskWrite-Time, and EstCpuTime).

Statistic	Explanation
AvgRunTime	Average time required for execution (measured).
EstDiskReads	Estimated number of read operations from the disk.
AvgDiskReads	Average number of read operations from the disk (measured).
EstDiskWrites	Estimated number of write operations to the disk.
AvgDiskWrites	Average number of write operations to the disk (measured).
EstDiskReadTime	Estimated time required for reading rows from the disk.
EstDiskWriteTime	Estimated time required for writing rows to the disk.
EstCpuTime	Estimated processor time required for execution.

Items in the plan related to SELECT, INSERT, UPDATE, and DELETE

Item	Explanation
Optimization Goal	Determines whether query processing is optimized towards returning the first row quickly, or minimizing the cost of returning the complete result set. See "optimization_goal option" [*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. See "optimization_workload option" [*SQL Anywhere Server - Database Administration*].
ANSI update constraints	Controls the range of updates that are permitted (options are Off, Cursors, and Strict). See "ansi_update_constraints option" [*SQL Anywhere Server - Database Administration*]
Optimization level	Reserved.
Select list	List of expressions selected by the query.

Item	Explanation
Materialized views	List of materialized views considered by the optimizer. Each entry in the list is a tuple in the following format: *`view-name [ view-matching-outcome ] [ table-list ]`* where *view-matching-outcome* reveals the usage of a materialized view; if the value is COSTED, the view was used during enumeration. The *table-list* is a list of query tables that were potentially replaced by this view. Values for *view-matching-outcome* include: • Base table mismatch • Permissions mismatch • Predicate mismatch • Select list mismatch • Costed • Stale mismatch • Snapshot stale mismatch • Cannot be used by optimizer • Cannot be used internally by optimizer • Cannot build definition • Cannot access • Disabled • Options mismatch • Reached view matching threshold • View used For more information about restrictions and conditions that prevent the optimizer from using a materialized view, see "Improving performance with materialized views" on page 555, and "Restrictions on materialized views" on page 39.

Items in the plan related to locks

Item	Explanation
Locked tables	List of all locked tables and their isolation levels.

Items in the plan related to scans

Item	Explanation
Table name	Actual name of the table.
Correlation name	Alias for the table.
Estimated rows	Estimated number of rows in the table.

Item	Explanation
Estimated pages	Estimated number of pages in the table.
Estimated row size	Estimated row size for the table.
Page maps	YES when a page map is used to read multiple pages.

Items in the plan related to index scans

Item	Explanation
Selectivity	Estimated number of rows that match the range bounds.
Index name	Name of the index.
Key type	Can be one of PRIMARY KEY, FOREIGN KEY, CONSTRAINT (unique constraint), or UNIQUE (unique index). The key type does not appear if the index is a non-unique secondary index.
Depth	Height of the index. See "Table and page sizes" on page 622.
Estimated leaf pages	Estimated number of leaf pages.
Sequential Transitions	Statistics for each physical index indicating how clustered the index is.
Random Transitions	Statistics for each physical index indicating how clustered the index is.
Key Values	The number of unique entries in the index.
Cardinality	Cardinality of the index if it is different from the estimated number of rows. This applies only to SQL Anywhere databases version 6.0.0 and earlier.
Direction	FORWARD or BACKWARD.
Range bounds	Range bounds are shown as a list (col_name=value) or col_name IN [low, high].
Primary Key Table	The primary key table name for a foreign key index scan.

Item	Explanation
Primary Key Table Estimated Rows	The number of rows in the primary key table for a foreign key index scan.
Primary Key Column	The primary key column names for a foreign key index scan.

Items in the plan related to joins, filter, and prefilter

Item	Explanation
Predicate	Search condition that is evaluated in this node, along with selectivity estimates and measurement. See "Viewing selectivity in the graphical plan" on page 601

Items in the plan related to hash filter

Item	Explanation
Build values	Estimated number of distinct values in the input.
Probe values	Estimated number of distinct values in the input when checking the predicate.
Bits	Number of bits selected to build the hash map.
Pages	Number of pages required to store the hash map.

Items in the plan related to Union

Item	Explanation
Union List	Columns involved in a UNION statement.

Items in the plan related to GROUP BY

Item	Explanation
Aggregates	All the aggregate functions.
Group-by list	All the columns in the group by clause.

Items in the plan related to DISTINCT

Item	Explanation
Distinct list	All the columns in the distinct clause.

Items in the plan related to IN LIST

Item	Explanation
In List	All the expressions in the specified set.
Expression SQL	Expressions to compare to the list.

Items in the plan related to SORT

Item	Explanation
Order-by	List of all expressions to sort by.

Items in the plan related to row limits

Item	Explanation
Row limit count	Maximum number of rows returned as specified by FIRST or TOP n.

Improving query performance

Storage allocations for each table or entry have a large impact on the efficiency of queries. The following points are of particular importance because each one influences how fast your queries execute.

Disk allocation for inserted rows

The following section explains how rows in the database are stored on disk.

SQL Anywhere stores rows contiguously, if possible

Every new row that is smaller than the page size of the database file is always stored on a single page. If no present page has enough free space for the new row, SQL Anywhere writes the row to a new page. For example, if the new row requires 600 bytes of space but only 500 bytes are available on a partially-filled page, then SQL Anywhere places the row on a new page.

To make table pages more contiguous on the disk, SQL Anywhere allocates table pages in blocks of eight pages. For example, when it needs to allocate a page it allocates eight pages, inserts the page in the block, and then fills up with the block with the next seven pages. In addition, it uses a free page bitmap to find

contiguous blocks of pages within the dbspace, and performs sequential scans by reading groups of 64 KB, using the bitmap to find relevant pages. This leads to more efficient sequential scans.

SQL Anywhere may store rows in any order

SQL Anywhere locates space on pages and inserts rows in the order it receives them in. It assigns each row to a page, but the locations it chooses in the table may not correspond to the order they were inserted in. For example, the database server may have to start a new page to store a long row contiguously. Should the next row be shorter, it may fit in an empty location on a previous page.

The rows of all tables are unordered. If the order that you receive or process the rows is important, use an ORDER BY clause in your SELECT statement to apply an ordering to the result. Applications that rely on the order of rows in a table can fail without warning.

If you frequently require the rows of a table to be in a particular order, consider creating an index on those columns specified in the query's ORDER BY clause.

Space is not reserved for NULL columns

By default, whenever SQL Anywhere inserts a row, it reserves only the space necessary to store the row with the values it contains at the time of creation. It reserves no space to store values that are NULL or to accommodate fields, such as text strings, which may enlarge.

You can force SQL Anywhere to reserve space by using the PCTFREE option when creating the table. For more information, see “CREATE TABLE statement” [*SQL Anywhere Server - SQL Reference*].

Once inserted, rows identifiers are immutable

Once assigned a home position on a page, a row never moves from that page. If an update changes any of the values in the row so that it no longer fits in its assigned page, then the row splits and the extra information is inserted on another page.

This characteristic deserves special attention, especially since SQL Anywhere allows no extra space when you insert the row. For example, suppose you insert a large number of empty rows into a table, then fill in the values, one column at a time, using UPDATE statements. The result would be that almost every value in a single row is stored on a separate page. To retrieve all the values from one row, the database server may need to read several disk pages. This simple operation would become extremely and unnecessarily slow.

You should consider filling new rows with data at the time of insertion. Once inserted, they then have enough room for the data you expect them to hold.

A database file never shrinks

As you insert and delete rows from the database, SQL Anywhere automatically reuses the space they occupy. So, SQL Anywhere may insert a row into space formerly occupied by another row.

SQL Anywhere keeps a record of the amount of empty space on each page. When you ask it to insert a new row, it first searches its record of space on existing pages. If it finds enough space on an existing page, it places the new row on that page, reorganizing the contents of the page if necessary. If not, it starts a new page.

Over time, if you delete several rows and do not insert new rows small enough to use the empty space, the information in the database may become sparse. You can reload the table, or use the REORGANIZE TABLE statement to defragment the table.

For more information, see "REORGANIZE TABLE statement" [*SQL Anywhere Server - SQL Reference*].

Table and page sizes

The page size you choose for your database can affect the performance of your database. In general, smaller page sizes are likely to benefit operations that retrieve a relatively small number of rows from random locations. By contrast, larger pages tend to benefit queries that perform sequential table scans, particularly when the rows are stored on pages in the order the rows are retrieved via an index. In this situation, reading one page into memory to obtain the values of one row may have the side effect of loading the contents of the next few rows into memory. Often, the physical design of disks permits them to retrieve fewer large blocks more efficiently than many small ones.

For each table, SQL Anywhere creates a bitmap that reflects the position of each table page in the entire dbspace file. The database server uses the bitmap to read large blocks (64 KB) of table pages, instead of single pages at a time. This efficiency, also known as **group reads**, reduces the total number of I/O operations to disk, and improves performance. Users cannot control the database server's criteria for bitmap creation or usage.

Should you choose a larger page size, such as 8 KB, you may want to increase the size of the cache because fewer large pages can fit into a cache of the same size. For example, 1 MB of memory can hold 512 pages that are each 2 KB in size, but only 128 pages that are 8 KB in size. Determining the proper page ratio of page size to cache size depends on your database and the nature of the queries your application performs. You can conduct performance tests with various cache sizes. If your cache cannot hold enough pages, performance suffers as the database server begins swapping frequently-used pages to disk. This is important when using SQL Anywhere on a Windows Mobile device, since larger page sizes may have a greater amount of internal fragmentation.

SQL Anywhere attempts to fill pages as much as possible. Empty space accumulates only when new objects are too large to fit empty space on existing pages. So, adjusting the page size may not significantly affect the overall size of your database.

Page size also affects indexes. Each index lookup requires one page read for each of the levels of the index plus one page read for the table page, and a single query can require several thousand index lookups. Page size can significantly affect fan-out, in turn affecting the depth of index required for a table. A large fan-out often means that fewer index levels are required, which can improve searches considerably. For large databases that have tables with a significant numbers of rows, 8 KB pages may be warranted for the best performance. It is strongly recommended that you test performance (and other behavior aspects) when choosing a page size. Then, choose the smallest page size that gives satisfactory results. It is important to pick the correct and reasonable page size if more than one database is started on the same server.

See also

- "Use an appropriate page size" on page 223
- "Initialization utility (dbinit)" [*SQL Anywhere Server - Database Administration*]
- "CREATE DATABASE statement" [*SQL Anywhere Server - SQL Reference*]

Indexes

Indexes can greatly improve the performance of searches on the indexed column(s). However, indexes take up space within the database and slow down insert, update, and delete operations. This section helps you determine when you should create an index and how to achieve maximum performance from your index.

There are many situations in which creating an index improves the performance of a database. An index provides an ordering of a table's rows based on the values in some or all the columns. An index allows SQL Anywhere to find rows quickly. It permits greater concurrency by limiting the number of database pages accessed. An index also affords SQL Anywhere a convenient means of enforcing a uniqueness constraint on the rows in a table.

When creating indexes, the order in which you specify the columns becomes the order in which the columns appear in the index. Duplicate references to column names in the index definition is not allowed.

The Index Consultant is a tool that assists you in the selection of an appropriate set of indexes for your database. See "Index Consultant" on page 164.

Index sharing using logical indexes

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. When you create a primary key, secondary key, foreign key, or unique constraint, the database server ensures referential integrity by creating a logical index for the constraint. Then, the database server looks to see if a physical index already exists that satisfies the constraint. If a qualifying physical index already exists, the database server points the logical index to it. If one does not exist, the database server creates a new physical index and then points the logical index to it.

For a physical index to satisfy the requirements of a logical index, the columns, column order and the ordering (ascending, descending) of data for each column must be identical.

Information about all logical and physical indexes in the database is recorded in the ISYSIDX and ISYSPHYSIDX system tables, respectively. When you create a logical index, an entry is made in the ISYSIDX system table to hold the index definition. A reference to the physical index used to satisfy the logical index is recorded in the ISYSIDX.phys_id column. The physical index is defined in the ISYSPHYSIDX system table.

For more information about the ISYSIDX and ISYSPHYSIDX system tables, see their corresponding views, "SYSIDX system view" [*SQL Anywhere Server - SQL Reference*] and "SYSPHYSIDX system view" [*SQL Anywhere Server - SQL Reference*].

Using logical indexes means that the database server does not need to create and maintain duplicate physical indexes, since more than one logical index can point to a single physical index.

When you delete a logical index, its definition is removed from the ISYSIDX system table. If it was the only logical index referencing a particular physical index, the physical index is also deleted, and its corresponding entry in the ISYSPHYSIDX system table.

You should always carefully consider whether an index is required before creating it. See "When to create an index" on page 625.

Physical indexes are not created for remote tables. For temporary tables, physical indexes are created, but they are not recorded in ISYSPHYSIDX, and are discarded after use. Also, physical indexes for temporary tables are not shared.

Determining which logical indexes share a physical index

When you drop an index, you are dropping a logical index; however, you are not always dropping the physical index to which it refers. If another logical index refers to the same physical index, the physical index is not deleted. This is important to know, especially if you expect disk space to be freed by dropping the index, or if you are dropping the index with the intent to physically recreate it.

To determine whether an index for a table is sharing a physical index with any other indexes, select the table in Sybase Central, and then click the **Indexes** tab. Note whether the Phys. ID value for the index is also present for other indexes in the list. Matching Phys. ID values mean that those indexes share the same physical index. If you want to recreate a physical index, you can use the ALTER INDEX ... REBUILD statement. Alternatively, you can drop all the indexes, and then recreate them.

Determining tables in which physical indexes are being shared

At any time, you can obtain a list of all tables in which physical indexes are being shared, by executing a query similar to the following:

```
SELECT tab.table_name, idx.table_id, phys.phys_index_id, COUNT(*)
 FROM SYSIDX idx JOIN SYSTAB tab ON (idx.table_id = tab.table_id)
 JOIN SYSPHYSIDX phys ON ( idx.phys_index_id  = phys.phys_index_id
   AND idx.table_id = phys.table_id )
 GROUP BY tab.table_name, idx.table_id, phys.phys_index_id
   HAVING COUNT(*) > 1
ORDER BY tab.table_name;
```

Following is an example result set for the query:

table_name	table_id	phys_index_id	COUNT(*)
ISYSCHECK	57	0	2
ISYSCOLSTAT	50	0	2
ISYSFKEY	6	0	2

table_name	table_id	phys_index_id	COUNT(*)
ISYSSOURCE	58	0	2
MAINLIST	94	0	3
MAINLIST	94	1	2

The number of rows for each table indicates the number of shared physical indexes for the tables. In this example, all the tables have one shared physical index, except for the fictitious table, MAINLIST, which has two. The phys_index_id values identifies the physical index being shared, and the value in the COUNT column tells you how many logical indexes are sharing the physical index.

You can also use Sybase Central to see which indexes for a given table share a physical index. To do this, choose the table in the left pane, click the **Indexes** tab in the right pane, and then look for multiple rows with the same value in the Phys. ID column. Indexes with the same value in Phys. ID share the same physical index.

See also

- "Rebuild indexes" on page 62
- "ALTER INDEX statement" [*SQL Anywhere Server - SQL Reference*]
- "SYSIDX system view" [*SQL Anywhere Server - SQL Reference*]

When to create an index

There is no simple formula to determine whether an index should be created. You must consider the trade-off of the benefits of indexed retrieval versus the maintenance overhead of that index. The following factors may help to determine whether you should create an index:

- **Keys and unique columns** SQL Anywhere automatically creates indexes on primary keys, foreign keys, and unique columns. You should not create additional indexes on these columns. The exception is composite keys, which can sometimes be enhanced with additional indexes.

  For more information, see "Composite indexes" on page 627.

- **Frequency of search** If a particular column is searched frequently, you can achieve performance benefits by creating an index on that column. Creating an index on a column that is rarely searched may not be worthwhile.

- **Size of table** Indexes on relatively large tables with many rows provide greater benefits than indexes on relatively small tables. For example, a table with only 20 rows is unlikely to benefit from an index, since a sequential scan would not take any longer than an index lookup.

- **Number of updates** An index is updated every time a row is inserted or deleted from the table and every time an indexed column is updated. An index on a column slows the performance of inserts,

updates and deletes. A database that is frequently updated should have fewer indexes than one that is read-only.

- **Space considerations** Indexes take up space within the database. If database size is a primary concern, you should create indexes sparingly.

- **Data distribution** If an index lookup returns too many values, it is more costly than a sequential scan. SQL Anywhere does not make use of the index when it recognizes this condition. For example, SQL Anywhere would not make use of an index on a column with only two values, such as Employees.Sex in the SQL Anywhere sample database. For this reason, you should not create an index on a column that has only a few distinct values.

The Index Consultant is a tool that assists you in the selection of an appropriate set of indexes for your database. See "Index Consultant" on page 164.

Temporary tables

You can create indexes on both local and global temporary tables. You may want to consider indexing a temporary table if you expect it will be large and accessed several times in sorted order or in a join. Otherwise, any improvement in performance for queries is likely to be outweighed by the cost of creating and dropping the index.

For more information, see "Working with indexes" on page 56.

Improving index performance

If your index is not performing as expected, you may want to consider the following actions:

- Reorganize composite indexes.
- Increase the page size.

These measures are aimed at increasing index selectivity and index fan-out, as explained below.

Index selectivity

Index selectivity refers to the ability of an index to locate a desired index entry without having to read additional data.

If selectivity is low, additional information must be retrieved from the table page that the index references. These retrievals are called **full compares**, and they have a negative effect on index performance.

The FullCompare property function keeps track of the number of full compares that have occurred. You can also monitor this statistic using the Sybase Central Performance Monitor or the Windows Performance Monitor.

> **Note**
> The Windows Performance Monitor may not be available on Windows Mobile.

In addition, the number of full compares is provided in the graphical plan with statistics. For more information, see "Common statistics used in the plan" on page 614.

For more information about the FullCompare function, see "Database properties" [*SQL Anywhere Server - Database Administration*].

Index structure and index fan-out

Indexes are organized in several levels, like a tree. The first page of an index, called the root page, branches into one or more pages at the next level, and each of those pages branch again, until the lowest level of the index is reached. These lowest level index pages are called leaf pages. To locate a specific row, an index with *n* levels requires *n* reads for index pages and one read for the data page containing the actual row. In general, fewer than *n* reads from disk are needed, since index pages that are used frequently tend to be stored in cache.

The **index fan-out** is the number of index entries stored on a page. An index with a higher fan-out may have fewer levels than an index with a lower fan-out. Therefore, higher index fan-out generally means better index performance. Choosing the correct page size for your database can improve index fan-out. See "Table and page sizes" on page 622.

You can see the number of levels in an index by using the sa_index_levels system procedure. See "sa_index_levels system procedure" [*SQL Anywhere Server - SQL Reference*].

Composite indexes

An index can contain one, two, or more columns. An index on two or more columns is called a **composite index**. For example, the following statement creates a two-column composite index:

```
CREATE INDEX name
ON Employees ( Surname, GivenName );
```

A composite index is useful if the first column alone does not provide high selectivity. For example, a composite index on Surname and GivenName is useful when many employees have the same surname. A composite index on EmployeeID and Surname would not be useful because each employee has a unique ID, so the column Surname does not provide any additional selectivity.

Additional columns in an index can allow you to narrow down your search, but having a two-column index is not the same as having two separate indexes. A composite index is structured like a telephone book, which first sorts people by their surnames, and then all the people with the same surname by their given names. A telephone book is useful if you know the surname, even more useful if you know both the given name and the surname, but worthless if you only know the given name and not the surname.

Column order

When you create composite indexes, you should think carefully about the order of the columns. Composite indexes are useful for doing searches on all the columns in the index or on the first columns only; they are not useful for doing searches on any of the later columns alone.

If you are likely to do many searches on one column only, that column should be the first column in the composite index. If you are likely to do individual searches on both columns of a two-column index, you may want to consider creating a second index that contains the second column only.

For example, suppose you create a composite index on two columns. One column contains employee's given names, the other their surnames. You could create an index that contains their given name, then their surname. Alternatively, you could index the surname, then the given name. Although these two indexes organize the information in both columns, they have different functions.

```
CREATE INDEX IX_GivenName_Surname
   ON Employees ( GivenName, Surname );
CREATE INDEX IX_Surname_GivenName
   ON Employees ( Surname, GivenName );
```

Suppose you then want to search for the given name John. The only useful index is the one containing the given name in the first column of the index. The index organized by surname then given name is of no use because someone with the given name John could appear anywhere in the index.

If you are more likely to look up people by given name only or surname only, then you should consider creating both of these indexes.

Alternatively, you could make two indexes, each containing only one of the columns. Remember, however, that SQL Anywhere only uses one index to access any one table while processing a single query. Even if you know both names, it is likely that SQL Anywhere needs to read extra rows, looking for those with the correct second name.

When you create an index using the CREATE INDEX command, as in the example above, the columns appear in the order shown in your command.

Composite indexes and ORDER BY

By default, the columns of an index are sorted in ascending order, but they can optionally be sorted in descending order by specifying DESC in the CREATE INDEX statement.

SQL Anywhere can choose to use an index to optimize an ORDER BY query as long as the ORDER BY clause contains only columns included in that index. In addition, the columns in the index must be ordered in exactly the same way, or in exactly the opposite way, as the ORDER BY clause. For single-column indexes, the ordering is always such that it can be optimized, but composite indexes require slightly more thought. The table below shows the possibilities for a two-column index.

Index columns	Optimizable ORDER BY queries	Not optimizable ORDER BY queries
ASC, ASC	ASC, ASC or DESC, DESC	ASC, DESC or DESC, ASC
ASC, DESC	ASC, DESC or DESC, ASC	ASC, ASC or DESC, DESC
DESC, ASC	DESC, ASC or ASC, DESC	ASC, ASC or DESC, DESC
DESC, DESC	DESC, DESC or ASC, ASC	ASC, DESC or DESC, ASC

An index with more than two columns follows the same general rule as above. For example, suppose you have the following index:


```
CREATE INDEX idx_example
ON table1 ( col1 ASC, col2 DESC, col3 ASC );
```

In this case, the following queries can be optimized:

```
SELECT col1, col2, col3 FROM table1
ORDER BY col1 ASC, col2 DESC, col3 ASC;

SELECT col1, col2, col3 FROM example
ORDER BY col1 DESC, col2 ASC, col3 DESC;
```

The index is not used to optimize a query with any other pattern of ASC and DESC in the ORDER BY clause. For example, the following statement is not optimized:

```
SELECT col1, col2, col3 FROM table1
ORDER BY col1 ASC, col2 ASC, col3 ASC;
```

Other uses for indexes

SQL Anywhere uses indexes to achieve other performance benefits. Having an index allows SQL Anywhere to enforce column uniqueness, to reduce the number of rows and pages that must be locked, and to better estimate the selectivity of a predicate.

- **Enforce column uniqueness** Without an index, SQL Anywhere has to scan the entire table every time that a value is inserted to ensure that it is unique. For this reason, SQL Anywhere automatically builds an index on every column with a uniqueness constraint.
- **Reduce locks** Indexes reduce the number of rows and pages that must be locked during inserts, updates, and deletes. This reduction is a result of the ordering that indexes impose on a table.

  For more information about indexes and locking, see "How locking works" on page 111.
- **Estimate selectivity** Because an index is ordered, the optimizer can estimate the percentage of values that satisfy a given query by scanning the upper levels of the index. This action is called a partial index scan.

B-link indexes

B-link indexes are a variant of B- and B+- tree indexes in which each index page, non-leaf and leaf, contains the page number of (or a link to) its right sibling. Further, index pages need not appear immediately in a parent page. The primary advantage of B-link indexes is improved concurrency.

Indexes can be declared as either clustered or unclustered. Only one index on a table can be clustered. If you determine that an index should be clustered, you do not need to drop and recreate the index: the clustering characteristic of an index can be removed or added by issuing an ALTER INDEX statement. Clustered indexes may assist performance by allowing the query optimizer to make more accurate decisions about the cost of index scans.

To improve fanout, SQL Anywhere stores a compressed form of each indexed value in which the prefix shared with the immediately preceding value is not stored. To reduce the CPU time when searching

within a page, a small look-aside map of complete index keys (subject to data length restrictions) is also stored. In particular, SQL Anywhere indexes efficiently handle index values that are identical (or nearly so), so common prefixes within the indexed values have negligible impact on storage requirements and performance.

See also

- "Using clustered indexes" on page 58
- "ALTER INDEX statement" [*SQL Anywhere Server - SQL Reference*]

SQL dialects and compatibility

This section describes Transact-SQL compatibility and those features of SQL Anywhere that are not commonly found in other SQL implementations.

SQL Anywhere complies with the SQL-92-based United States Federal Information Processing Standard Publication (FIPS PUB) 127. With minor exceptions, SQL Anywhere is compliant with the ISO/ANSI SQL/2008 core specification as documented in the 9 parts of ISO/IEC JTC 1/SC 32 9075-2008. Information about compliance is provided in the reference documentation for each feature of SQL Anywhere.

Testing SQL compliance using the SQL Flagger

In SQL Anywhere, the database server and the SQL preprocessor (sqlpp) can identify SQL statements that are vendor extensions, are not compliant with specific ISO/ANSI SQL standards, or are not supported by UltraLite. This functionality is called the SQL Flagger, first introduced as optional SQL language feature F812 of the ISO/ANSI 9075-1999 SQL standard, referred to as SQL/1999 in this document. The SQL Flagger helps an application developer to identify SQL language constructs that violate a specified subset of the SQL language. The SQL Flagger can also be used to ensure compliance with core features of a SQL standard, or compliance with a combination of core and optional features. The SQL Flagger can also be used when prototyping an UltraLite application with SQL Anywhere, to ensure that the SQL being used is supported by UltraLite.

As spatial data support is standardized as Part 3 of the SQL/MM standard (ISO/IEC 13249-3), spatial functions, operations, and syntax are not supported by the SQL Flagger and are flagged as vendor extensions.

The SQL Flagger is intended to provide static, compile-time checking of compliance, although both syntactic and semantic elements of a SQL statement are candidates for analysis by the SQL Flagger. An example test of syntactic compliance is the lack of the optional INTO keyword in an INSERT statement (for example, `INSERT Products VALUES( ... )`), which is a SQL Anywhere grammar extension to the SQL language. The use of an INSERT statement without the INTO keyword is flagged as a vendor extension because the ANSI SQL/2008 standard mandates the use of the INTO keyword. Note, however, that the INTO keyword is optional for UltraLite applications.

Key joins are also flagged as a vendor extension. A key join is used by default when the JOIN keyword is used without an ON clause. A key join uses existing foreign key relationships to join the tables. Key joins are not supported by UltraLite. For example, the following query specifies an implicit join condition between the Products and SalesOrderItems tables. This query is flagged by the SQL Flagger as a vendor extension.

```
SELECT * FROM Products JOIN SalesOrderItems;
```

SQL Flagger functionality is not dependent on the execution of a SQL statement; all flagging logic is done only as a static, compile-time process.

See also

- "SQLFLAGGER function [Miscellaneous]" [*SQL Anywhere Server - SQL Reference*]
- "Compliance with spatial standards" [*SQL Anywhere Server - Spatial Data Support*]
- "Running the SQL preprocessor" [*SQL Anywhere Server - Programming*]
- "INSERT statement" [*SQL Anywhere Server - SQL Reference*]
- "Key joins" on page 417

Invoking the SQL Flagger

SQL Anywhere offers several ways to invoke the SQL Flagger to check a SQL statement, or a batch of SQL statements:

- **SQLFLAGGER function** The SQLFLAGGER function analyzes a single SQL statement, or batch, passed as a string argument, for compliance with a given SQL standard. The statement or batch is parsed, but not executed. See "SQLFLAGGER function [Miscellaneous]" [*SQL Anywhere Server - SQL Reference*].

- **sa_ansi_standard_packages system procedure** The sa_ansi_standard_packages system procedure analyzes a statement, or batch, for the use of optional SQL language features, or packages, from the ANSI SQL/2008, SQL/2003 or SQL/1999 international standards. The statement or batch is parsed, but not executed. See "sa_ansi_standard_packages system procedure" [*SQL Anywhere Server - SQL Reference*].

- **sql_flagger_error_level and sql_flagger_warning_level options** The sql_flagger_error_level and sql_flagger_warning_level options invoke the SQL Flagger for any statement prepared or executed for the connection. If the statement does not comply with the option setting, which is a specific ANSI standard or UltraLite, the statement either terminates with an error (SQLSTATE 0AW03), or returns a warning (SQLSTATE 01W07), depending upon the option setting. If the statement complies, statement execution proceeds normally. See "sql_flagger_error_level option" [*SQL Anywhere Server - Database Administration*] and "sql_flagger_warning_level option" [*SQL Anywhere Server - Database Administration*].

- **SQL preprocessor (sqlpp)** The SQL preprocessor (sqlpp) has the ability to flag static SQL statements in an embedded SQL application at compile time. This feature can be especially useful when developing an UltraLite application, to verify SQL statements for UltraLite compatibility. See "SQL preprocessor" [*SQL Anywhere Server - Programming*], and "Running the SQL preprocessor" [*SQL Anywhere Server - Programming*].

See also

- "Introduction to batches" on page 812

Standards and compatibility

The flagging functionality used in the database server and in the SQL preprocessor follows the SQL Flagger functionality defined in Part 1 (Framework) of the ANSI/ISO SQL/2008 International Standard.

The SQL Flagger supports the following ANSI SQL standards when determining the compliance of SQL language constructions :

- SQL/1992 Entry level, Intermediate level, and Full level
- SQL/1999 Core, and SQL/1999 optional packages
- SQL/2003 Core, and SQL/2003 optional packages
- SQL/2008 Core, and SQL/2008 optional packages

Note
SQL Flagger support for SQL/1992 (all levels) is deprecated.

In addition, the SQL Flagger can identify statements that are not compliant with UltraLite SQL. For example, UltraLite has only limited abilities to CREATE and ALTER schema objects.

All SQL statements can be analyzed by the SQL Flagger. However, most statements that create or alter schema objects, including statements that create tables, indexes, materialized views, publications, subscriptions, and proxy tables, are vendor extensions to the ANSI SQL standards, and are flagged as non-conforming.

The SET OPTION statement, including its optional components, is never flagged for non-compliance with any SQL standard, or for compatibility with UltraLite.

See also

- "UltraLite SQL elements" [*UltraLite - Database Management and Reference*]
- "SET OPTION statement" [*SQL Anywhere Server - SQL Reference*]

SQL Anywhere features that differ from other SQL implementations

SQL Anywhere offers rich SQL functionality, including: per-row, per-statement, and INSTEAD OF triggers; SQL stored procedures and user-defined functions; RECURSIVE UNION queries; common table expressions; table functions; LATERAL derived tables; integrated full-text search; WINDOW aggregate functions; regular-expression searching; XML support; materialized views; snapshot isolation; and referential integrity. This section describes some specific features supported by SQL Anywhere that differ from other SQL database implementations.

Dates

SQL Anywhere has date, time and timestamp types that include a year, month and day, hour, minutes, seconds, and fraction of a second. For insertions or updates to date fields, or comparisons with date fields, a free format date is supported.

In addition, the following operations are allowed on dates:

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

- **date - integer** Subtract the specified number of days from a date.
- **date - date** Compute the number of days between two dates.
- **date + time** Make a timestamp out of a date and time.

SQL Anywhere does not support an INTERVAL data type, which is SQL language feature F052 of the SQL/2008 standard. However, SQL Anywhere provides many functions, such as DATEADD, for manipulating dates and times. See "Date and time functions" [*SQL Anywhere Server - SQL Reference*].

Entity and referential integrity

SQL Anywhere supports both entity and referential integrity via the PRIMARY KEY and FOREIGN KEY clauses of the CREATE TABLE and ALTER TABLE statements.

PRIMARY KEY [**CLUSTERED**] (*column-name* [**ASC** | **DESC**], ...)
[**NOT NULL**] **FOREIGN KEY** [*role-name*]
[(*column-name* [**ASC** | **DESC**], ...)]
REFERENCES *table-name* [(*column-name*, ...)]
[**MATCH** [**UNIQUE** | **SIMPLE** | **FULL**]]
[**ON UPDATE** [**CASCADE** | **RESTRICT** | **SET DEFAULT** | **SET NULL**]]
[**ON DELETE** [**CASCADE** | **RESTRICT** | **SET DEFAULT** | **SET NULL**]]
[**CHECK ON COMMIT**] [**CLUSTERED**]

The PRIMARY KEY clause declares the primary key for the table. SQL Anywhere then enforces the uniqueness of the primary key by creating a unique index over the primary key column(s). Two SQL Anywhere extensions permit the customization of this index:

- **CLUSTERED** The CLUSTERED keyword signifies that the primary key index is a clustered index, and therefore adjacent index entries in the index point to physically-adjacent rows in the table.
- **ASC | DESC** The sortedness—ascending or descending—of each indexed column in the primary key index can be customized. This customization can be used to ensure that the sortedness of the primary key index matches the sortedness required by specific SQL queries, as specified in those statements' ORDER BY clauses.

The FOREIGN KEY clause defines a relationship between two tables. This relationship is represented by a column (or columns) in this table that must contain values in the primary key of another table. SQL Anywhere automatically constructs an index for each FOREIGN KEY defined to enforce the referential constraint. The semantics of the constraint, and physical characteristics of this index, can be customized as follows:

- **CLUSTERED** The CLUSTERED keyword signifies that the foreign key index is a clustered index, and therefore adjacent index entries in the index point to physically-adjacent rows in the foreign table.
- **ASC | DESC** The sortedness—ascending or descending—of each indexed column in the foreign key index can be customized. The sortedness of the foreign key index may differ from that of the primary key index. Sortedness customization can be used to ensure that the sortedness of the foreign key index matches the sortedness required by specific SQL queries in your application, as specified in those statements' ORDER BY clauses.

- **MATCH clause** SQL Anywhere supports the MATCH clause, which is SQL language feature F741 of the SQL/2008 standard. In addition, SQL Anywhere supports MATCH UNIQUE, which enforces a one-to-one relationship between the primary and foreign tables without the need for an additional UNIQUE index.

For more information, see "CREATE TABLE statement" [*SQL Anywhere Server - SQL Reference*].

Unique indexes

SQL Anywhere supports the creation of unique indexes, sometimes called unique secondary indexes, over nullable columns. By default, each index key must be unique or contain a NULL in at least one column. For example, two index entries ('a', NULL) and ('a', NULL) are each considered unique index values. SQL Anywhere also supports unique secondary indexes where NULL values are treated as special values in each domain. This is accomplished using the WITH NULLS NOT DISTINCT clause. With such an index, the two pairs of values ('a', NULL) and ('a', NULL) are considered duplicates.

For more information, see "CREATE INDEX statement" [*SQL Anywhere Server - SQL Reference*].

Joins

SQL Anywhere supports INNER, LEFT OUTER, RIGHT OUTER, and FULL OUTER joins. In addition to explicit join predicates, SQL Anywhere supports NATURAL joins and a vendor extension known as KEY joins, which specifies an implicit join predicate based on the tables' foreign key relationships.

For more information about key joins, see "Key joins" on page 417.

CHAR, NCHAR, and BINARY data types

SQL Anywhere internals do not distinguish between fixed- and varying-length string types (CHAR, NCHAR, or BINARY). SQL Anywhere does not truncate trailing blanks from string types when such values are inserted to the database. SQL Anywhere does distinguish between the NULL value and the empty string. By default, SQL Anywhere databases utilize a case-insensitive collation to support case-insensitive string comparisons. In SQL Anywhere, fixed-length string types are never blank-padded; rather, blank-padding semantics are simulated during the execution of each string comparison. These semantics may differ subtly from string comparisons with other SQL implementations.

For more information, see "Character data types" [*SQL Anywhere Server - SQL Reference*].

UPDATE statements

SQL Anywhere partially supports optional SQL language feature T111 that permits an UPDATE statement to refer to a view that contains a join. In addition, the UPDATE and UPDATE WHERE CURRENT OF statements permit more than one table to be referenced in the statement's SET clause, and the FROM clause of an UPDATE statement can be comprised of an arbitrary table expression containing joins and derived tables.

SQL Anywhere also allows the UPDATE, INSERT, MERGE, and DELETE statements to be embedded within another SQL statement as a derived table. One of the benefits of this support is that you can construct a query that returns the set of rows that has been modified by an UPDATE statement in a straightforward way.

For more information about DML derived tables, see "FROM clause" [*SQL Anywhere Server - SQL Reference*].

Table functions

SQL Anywhere lets you refer to the result set of a stored procedure as a table in a statement's FROM clause, a feature commonly referred to as table functions. Table functions are SQL language feature T326 of the SQL/2008 standard. In the standard, table functions are specified using the TABLE keyword. In SQL Anywhere, use of the TABLE keyword is unnecessary; a stored procedure can be referenced directly in the FROM clause, optionally with a correlation name and a specification of schema of the result set returned by the procedure.

The following example joins the result of the stored procedure ShowCustomerProducts with the base table Products. Accompanying the stored procedure reference is an explicit declaration of the schema of the procedure's result, using 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
```

For more information about table functions, see "FROM clause" [*SQL Anywhere Server - SQL Reference*].

Materialized views

SQL Anywhere supports materialized views, which are precomputed result sets that can be referenced directly or indirectly from within a SQL query. In SQL Anywhere, both immediately-maintained and manually-maintained views can be created using the CREATE MATERIALIZED VIEW statement. Other database products may use different terms to describe this functionality.

For more information on materialized views, see "Working with materialized views" on page 33.

Cursors

SQL Anywhere supports optional SQL language feature F431 of the SQL/2008 standard. In SQL Anywhere, all cursors are bi-directionally scrollable unless they are explicitly declared FORWARD ONLY, and applications can scroll through a cursor using either relative or absolute positioning with the FETCH statement or its equivalent with other application programming interfaces, such as ODBC.

SQL Anywhere supports value-sensitive and row-membership sensitive cursors. Commonly-supported cursor types, including INSENSITIVE, KEYSET-DRIVEN, and SENSITIVE cursors, are supported. When using embedded SQL, cursor positions can be moved arbitrarily on the FETCH statement. Cursors can be moved forward or backward relative to the current position or a given number of records from the beginning or end of the cursor.

For more information about cursors and cursor sensitivity, see "SQL Anywhere cursors" [*SQL Anywhere Server - Programming*].

By default, cursors in embedded SQL and SQL procedures, user-defined functions, and triggers are updatable. They can be made explicitly updatable by using the FOR UPDATE clause. However, specifying the FOR UPDATE clause alone does not acquire any locks on the rows in the cursor's result set. To ensure that rows in the result set cannot be modified by other transactions, you can specify either:

- **FOR UPDATE BY LOCK** This clause causes the database server to acquire intent row locks on fetched rows of the result set. These are long-term locks that are held until the transaction is committed or rolled back.

- **FOR UPDATE BY { VALUES | TIMESTAMP }** The SQL Anywhere database server uses a keyset-driven cursor to enable the application to be informed when rows have been modified or deleted as the result set is scrolled.

For more information about cursor updatability, see "DECLARE CURSOR statement [ESQL] [SP]" [*SQL Anywhere Server - SQL Reference*].

Alias references

SQL Anywhere permits aliased expressions in the select list of a query to be referenced in other parts of the query. Most other SQL implementations and the SQL/2008 standard do not allow this behavior. For example, you can specify the SQL query:

```
SELECT column-or-expression AS alias-name
FROM table-reference
WHERE alias-name = expression
```

Aliases can be used anywhere in the SELECT block, including other select list expressions that in turn define additional aliases. Cyclic alias references are not permitted. If the alias specified for an expression is identical to the name of a column or variable in the name space of the SELECT block, the alias definition occludes the column or variable. Column names, however, can be explicitly qualified by table name in such cases.

Snapshot isolation

SQL Anywhere supports snapshot isolation, which is also known as Multi-Version Concurrency Control, or MVCC. In other SQL implementations that support snapshot isolation, writer-writer conflicts - that is, concurrent updates by two or more transactions to the same row - are made apparent only at the time of COMMIT. In such cases, usually the first COMMIT wins, and the other transactions involved in the conflict must abort.

In SQL Anywhere, write operations to rows cause write row locks to be acquired so that snapshot transactions can co-exist with transactions executing at ANSI isolation levels. Consequently, a writer-writer conflict in SQL Anywhere will result in blocking, though the precise behavior can be controlled through the BLOCKING and BLOCKING_TIMEOUT connection options.

For more information on snapshot isolation, see "Snapshot isolation" on page 94.

Watcom SQL

The dialect of SQL supported by SQL Anywhere is referred to as Watcom SQL. The original version of SQL Anywhere was called Watcom SQL when it was introduced in 1992. The term Watcom SQL is still used to identify the dialect of SQL supported by SQL Anywhere.

SQL Anywhere also supports a large subset of Transact-SQL, the dialect of SQL supported by Sybase Adaptive Server Enterprise. See "Transact-SQL compatibility" on page 638.

Transact-SQL compatibility

SQL Anywhere supports a large subset of Transact-SQL, the dialect of SQL supported by Sybase Adaptive Server Enterprise. This section describes compatibility of SQL between SQL Anywhere and Adaptive Server Enterprise.

Goals

The goals of Transact-SQL support in SQL Anywhere are as follows:

- **Application portability** Many applications, stored procedures, and batch files can be written for use with both Adaptive Server Enterprise and SQL Anywhere databases.
- **Data portability** SQL Anywhere and Adaptive Server Enterprise databases can exchange and replicate data between each other with minimum effort.

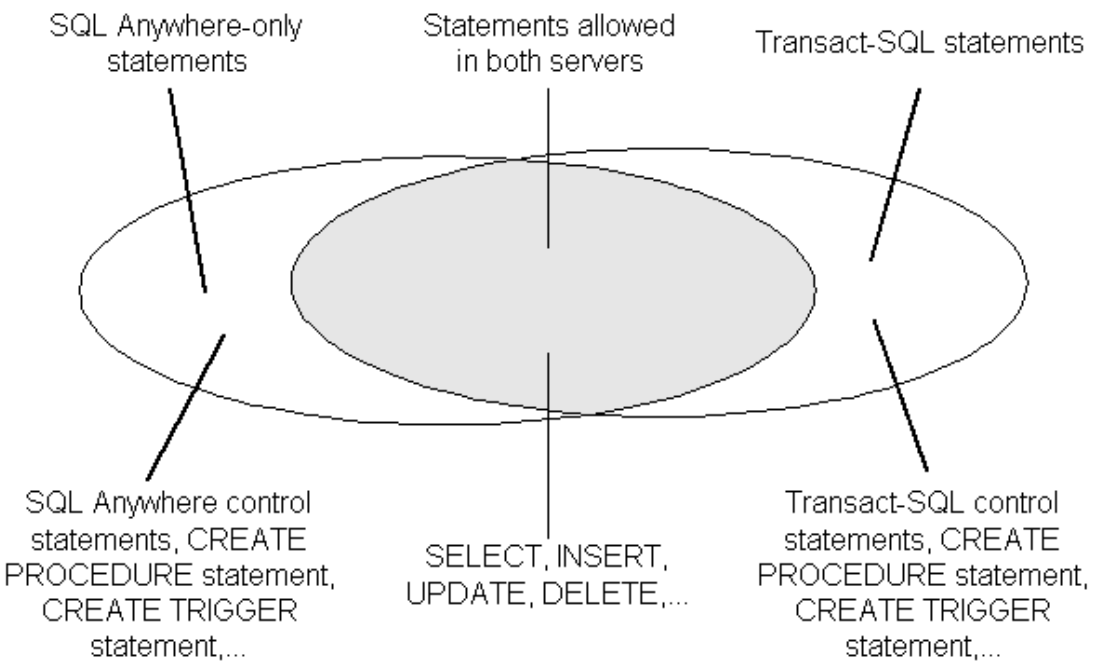
The aim is to write applications to work with both Adaptive Server Enterprise and SQL Anywhere. Existing Adaptive Server Enterprise applications generally require some changes to run on a SQL Anywhere database.

How Transact-SQL is supported

Transact-SQL support in SQL Anywhere takes the following form:

- Many SQL statements are compatible between SQL Anywhere and Adaptive Server Enterprise.
- For some statements, particularly in the procedure language used in procedures, triggers, and batches, a separate Transact-SQL statement is supported together with the syntax supported in previous versions of SQL Anywhere. For these statements, SQL Anywhere supports two dialects of SQL. Those dialects are called Transact-SQL — the dialect of Adaptive Server Enterprise, and Watcom SQL — the dialect of SQL Anywhere.
- A procedure, trigger, or batch is executed in either the Transact-SQL or Watcom SQL dialect. You must use control statements from one dialect only throughout the batch or procedure. For example, each dialect has different flow control statements.

The following diagram illustrates how the two dialects overlap.



Similarities and differences

SQL Anywhere supports a high percentage of Transact-SQL language elements, functions, and statements for working with existing data. For example, SQL Anywhere supports all numeric, aggregate, and date and time functions, and all but one string function. As another example, SQL Anywhere supports extended DELETE and UPDATE statements using joins.

Further, SQL Anywhere supports a high percentage of the Transact-SQL stored procedure language (CREATE PROCEDURE and CREATE TRIGGER syntax, control statements, and so on) and many, but not all, aspects of Transact-SQL data definition language statements.

There are design differences in the architectural and configuration facilities supported by each product. Device management, user management, and maintenance tasks such as backups tend to be system-specific. Even here, SQL Anywhere provides Transact-SQL system tables as views, where the tables that are not meaningful in SQL Anywhere have no rows. Also, SQL Anywhere provides a set of system procedures for some common administrative tasks.

This section looks first at some system-level issues where differences are most noticeable, before discussing data manipulation and data definition language aspects of the dialects where compatibility is high.

Transact-SQL only

Some SQL statements supported by SQL Anywhere are part of one dialect, but not the other. You cannot mix the two dialects within a procedure, trigger, or batch. For example, SQL Anywhere supports the following statements, but as part of the Transact-SQL dialect only:

- Transact-SQL control statements IF and WHILE

- Transact-SQL EXECUTE statement
- Transact-SQL CREATE PROCEDURE and CREATE TRIGGER statements
- Transact-SQL BEGIN TRANSACTION statement
- SQL statements *not* separated by semicolons are part of a Transact-SQL procedure or batch

SQL Anywhere only

Adaptive Server Enterprise does not support the following statements:

- LOOP and FOR control statements
- SQL Anywhere versions of IF and WHILE
- CALL statement
- SIGNAL statement
- SQL Anywhere versions of the CREATE PROCEDURE, CREATE FUNCTION, and CREATE TRIGGER statements
- SQL statements separated by semicolons

Notes

The two dialects cannot be mixed within a procedure, trigger, or batch. This means that:

- You can include Transact-SQL-only statements together with statements that are part of both dialects in a batch, procedure, or trigger.
- You can include statements not supported by Adaptive Server Enterprise together with statements that are supported by both servers in a batch, procedure, or trigger.
- You cannot include Transact-SQL-only statements together with SQL Anywhere-only statements in a batch, procedure, or trigger.

Adaptive Server Enterprise architectures

Adaptive Server Enterprise and SQL Anywhere are complementary products, with architectures designed to suit their distinct purposes.

This section describes architectural differences between Adaptive Server Enterprise and SQL Anywhere. It also describes the Adaptive Server Enterprise-like tools that SQL Anywhere includes for compatible database management.

Servers and databases

The relationship between servers and databases is different in Adaptive Server Enterprise and SQL Anywhere.

In Adaptive Server Enterprise, each database exists inside a server, and each server can contain several databases. Users can have login rights to the server, and can connect to the server. They can then use each database on that server for which they have permissions. System-wide system tables, held in a master database, contain information common to all databases on the server.

No master database in SQL Anywhere

In SQL Anywhere, there is no level corresponding to the Adaptive Server Enterprise master database. Instead, each database is an independent entity, containing all of its system tables. Users can have connection rights to a database, not to the server. When a user connects, they connect to an individual database. There is no system-wide set of system tables maintained at a master database level. Each SQL Anywhere database server can dynamically load and unload multiple databases, and users can maintain independent connections on each.

SQL Anywhere provides tools in its Transact-SQL support and in its Open Server support to allow some tasks to be performed in a manner similar to Adaptive Server Enterprise. For example, SQL Anywhere provides an implementation of the Adaptive Server Enterprise sp_addlogin system procedure that performs the nearest equivalent action: adding a user to a database. See "Using SQL Anywhere as an Open Server" [*SQL Anywhere Server - Database Administration*].

File manipulation statements

SQL Anywhere does not support the Transact-SQL statements DUMP DATABASE and LOAD DATABASE for backing up and restoring. Instead, SQL Anywhere has its own BACKUP DATABASE and RESTORE DATABASE statements with different syntax.

Device management

SQL Anywhere and Adaptive Server Enterprise use different models for managing devices and disk space, reflecting the different uses for the two products. While Adaptive Server Enterprise sets out a comprehensive resource management scheme using a variety of Transact-SQL statements, SQL Anywhere manages its own resources automatically, and its databases are regular operating system files.

SQL Anywhere does not support Transact-SQL DISK statements, such as DISK INIT, DISK MIRROR, DISK REFIT, DISK REINIT, DISK REMIRROR, and DISK UNMIRROR.

For information about disk management, see "Working with database files" [*SQL Anywhere Server - Database Administration*].

Defaults and rules

SQL Anywhere does not support the Transact-SQL CREATE DEFAULT statement or CREATE RULE statement. The CREATE DOMAIN statement allows you to incorporate a default and a rule (called a

CHECK condition) into the definition of a domain, and so provides similar functionality to the Transact-SQL CREATE DEFAULT and CREATE RULE statements.

In SQL Anywhere, a domain can have a default value and a CHECK condition associated with it, which are applied to all columns defined on that data type. You create the domain using the CREATE DOMAIN statement.

You can define default values and rules, or CHECK conditions, for individual columns using the CREATE TABLE statement or the ALTER TABLE statement.

In Adaptive Server Enterprise, the CREATE DEFAULT statement creates a named default. This default can be used as a default value for columns by binding the default to a particular column or as a default value for all columns of a domain by binding the default to the data type using the sp_bindefault system procedure. The CREATE RULE statement creates a named rule that can be used to define the domain for columns by binding the rule to a particular column or as a rule for all columns of a domain by binding the rule to the data type. A rule is bound to a data type or column using the sp_bindrule system procedure.

See also

- "CREATE DOMAIN statement" [*SQL Anywhere Server - SQL Reference*]
- "CREATE TABLE statement" [*SQL Anywhere Server - SQL Reference*]
- "ALTER TABLE statement" [*SQL Anywhere Server - SQL Reference*]
- "Search conditions" [*SQL Anywhere Server - SQL Reference*]

System tables

In addition to its own system tables, SQL Anywhere provides a set of system views that mimic relevant parts of the Adaptive Server Enterprise system tables.

For a list and individual descriptions, including descriptions of the system catalogs of the two products, see "Views for Transact-SQL compatibility" [*SQL Anywhere Server - SQL Reference*].

The SQL Anywhere system tables rest entirely within each database, while the Adaptive Server Enterprise system tables rest partly inside each database and partly in the master database. The SQL Anywhere architecture does not include a master database.

In Adaptive Server Enterprise, the database owner (user dbo) owns the system tables. In SQL Anywhere, the system owner (user SYS) owns the system tables. The user dbo owns the Adaptive Server Enterprise-compatible system views provided by SQL Anywhere.

Administrative roles

Adaptive Server Enterprise has a more elaborate set of administrative roles than SQL Anywhere. In Adaptive Server Enterprise there is a set of distinct roles, although more than one login account on an Adaptive Server Enterprise can be granted any role, and one account can possess more than one role.

Adaptive Server Enterprise roles

In Adaptive Server Enterprise distinct roles include:

- **System Administrator** Responsible for general administrative tasks unrelated to specific applications; can access any database object.
- **System Security Officer** Responsible for security-sensitive tasks in Adaptive Server Enterprise, but has no special permissions on database objects.
- **Database Owner** Has full permissions on objects inside the database he or she owns, can add users to a database and grant other users the permission to create objects and execute commands within the database.
- **Data definition statements** Permissions can be granted to users for specific data definition statements, such as CREATE TABLE or CREATE VIEW, enabling the user to create database objects.
- **Object owner** Each database object has an owner who may grant permissions to other users to access the object. The owner of an object automatically has all permissions on the object.

In SQL Anywhere, the following database-wide permissions have administrative roles:

- The Database Administrator (DBA authority) has, like the Adaptive Server Enterprise database owner, full permissions on all objects inside the database (other than objects owned by SYS) and can grant other users the permission to create objects and execute commands within the database. The default database administrator is user DBA.
- The RESOURCE authority allows a user to create any kind of object within a database. This is instead of the Adaptive Server Enterprise scheme of granting permissions on individual CREATE statements.
- SQL Anywhere has object owners in the same way that Adaptive Server Enterprise does. The owner of an object automatically has all permissions on the object, including the right to grant permissions.

For seamless access to data held in both Adaptive Server Enterprise and SQL Anywhere, you should create user IDs with appropriate permissions in the database (RESOURCE in SQL Anywhere, or permission on individual CREATE statements in Adaptive Server Enterprise) and create objects from that user ID. If you use the same user ID in each environment, object names and qualifiers can be identical in the two databases, ensuring compatible access.

See also

- “Database permissions and authorities” [*SQL Anywhere Server - Database Administration*]
- “DBA authority” [*SQL Anywhere Server - Database Administration*]
- “RESOURCE authority” [*SQL Anywhere Server - Database Administration*]

Users and groups

There are some differences between the Adaptive Server Enterprise and SQL Anywhere models of users and groups.

In Adaptive Server Enterprise, users connect to a server. Each user requires a login ID and password to the server and a user ID for each database they want to access on that server. Each user of a database can only be a member of one group.

In SQL Anywhere, users connect directly to a database and do not require a separate login ID to the database server. Instead, each user receives a user ID and password on a database so they can use that database. Users can be members of many groups, and group hierarchies are allowed.

Both servers support groups, so you can grant permissions to many users at one time. However, there are differences in the specifics of groups in the two servers. For example, Adaptive Server Enterprise allows each user to be a member of only one group, while SQL Anywhere has no such restriction. You should compare the documentation on users and groups in the two products for specific information.

Both Adaptive Server Enterprise and SQL Anywhere have a public group, for defining default permissions. Every user automatically becomes a member of the public group.

SQL Anywhere supports the following Adaptive Server Enterprise system procedures for managing users and groups. See "Adaptive Server Enterprise system and catalog procedures" [*SQL Anywhere Server - SQL Reference*].

System procedure	Description
sp_addlogin	In Adaptive Server Enterprise, this adds a user to the server. In SQL Anywhere, this adds a user to a database.
sp_adduser	In Adaptive Server Enterprise and SQL Anywhere, this adds a user to a database. While this is a distinct task from sp_addlogin in Adaptive Server Enterprise, in SQL Anywhere, they are the same.
sp_addgroup	Adds a group to a database.
sp_changegroup	Adds a user to a group, or moves a user from one group to another.
sp_droplogin	In Adaptive Server Enterprise, removes a user from the server. In SQL Anywhere, removes a user from the database.
sp_dropuser	Removes a user from the database.
sp_dropgroup	Removes a group from the database.

In Adaptive Server Enterprise, login IDs are server-wide. In SQL Anywhere, users belong to individual databases.

Database object permissions

The Adaptive Server Enterprise and SQL Anywhere GRANT and REVOKE statements for granting permissions on individual database objects are very similar. Both allow SELECT, INSERT, DELETE, UPDATE, and REFERENCES permissions on database tables and views, and UPDATE permissions on selected columns of database tables. Both allow EXECUTE permissions to be granted on stored procedures.

For example, the following statement is valid in both Adaptive Server Enterprise and SQL Anywhere:

```
GRANT INSERT, DELETE
ON Employees
TO MARY, SALES;
```

This statement grants permission to use the INSERT and DELETE statements on the Employees table to user MARY and to the SALES group.

Both SQL Anywhere and Adaptive Server Enterprise support the WITH GRANT OPTION clause, allowing the recipient of permissions to grant them in turn, although SQL Anywhere does not permit WITH GRANT OPTION to be used on a GRANT EXECUTE statement. In SQL Anywhere, you can only specify WITH GRANT OPTION for users. Members of groups do not inherit the WITH GRANT OPTION if it is granted to a group.

Database-wide permissions

Adaptive Server Enterprise and SQL Anywhere use different models for database-wide user permissions. SQL Anywhere employs DBA permissions to allow a user full authority within a database. The System Administrator in Adaptive Server Enterprise enjoys this permission for all databases on a server. However, DBA authority on a SQL Anywhere database is different from the permissions of an Adaptive Server Enterprise Database Owner, who must use the Adaptive Server Enterprise SETUSER statement to gain permissions on objects owned by other users. See “Users and groups” on page 643.

SQL Anywhere employs RESOURCE permissions to allow a user the right to create objects in a database. A closely corresponding Adaptive Server Enterprise permission is GRANT ALL, used by a Database Owner.

Configuring databases for Transact-SQL compatibility

You can eliminate some differences in behavior between SQL Anywhere and Adaptive Server Enterprise by selecting appropriate options when creating a database or, if you are working on an existing database, when rebuilding the database. You can control other differences by connection level options using the SET TEMPORARY OPTION statement in SQL Anywhere or the SET statement in Adaptive Server Enterprise.

Creating a Transact-SQL-compatible database

This section describes choices you must make when creating or rebuilding a database.

Quick start

Here are the steps you need to take to create a Transact-SQL-compatible database. The remainder of the section describes which options you need to set.

To create a Transact-SQL compatible database (Sybase Central)

1. Use the SQL Anywhere 12 plug-in to connect to the database.
2. Choose **Tools** » **SQL Anywhere 12** » **Create Database**.
3. Follow the instructions in the wizard.
4. Click **Emulate Adaptive Server Enterprise**. Click **Next**.
5. Follow the remaining instructions in the wizard.

To create a Transact-SQL compatible database (command line)

- Run the following dbinit command:

  ```
  dbinit -b -c -k db-name.db
  ```

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

To create a Transact-SQL compatible database (SQL)

1. Connect to any SQL Anywhere database.
2. Enter the following statement, for example, in Interactive SQL:

   ```
   CREATE DATABASE 'db-name.db'
   ASE COMPATIBLE
   CASE RESPECT
   BLANK PADDING ON;
   ```

   In this statement, ASE COMPATIBLE means compatible with Adaptive Server Enterprise. It prevents the SYS.SYSCOLUMNS and SYS.SYSINDEXES views from being created.

Make the database case sensitive

By default, string comparisons in Adaptive Server Enterprise databases are case sensitive, while those in SQL Anywhere are case insensitive.

When building an Adaptive Server Enterprise-compatible database using SQL Anywhere, choose the case sensitive option.

- If you are using Sybase Central, this option is in the **Create Database Wizard**.
- If you are using the dbinit utility, specify the -c option.

Ignore trailing blanks in comparisons

When building an Adaptive Server Enterprise-compatible database using SQL Anywhere, choose the option to ignore trailing blanks in comparisons.

- If you are using Sybase Central, this option is in the **Create Database Wizard**.

- If you are using the dbinit utility, specify the -b option.

When you choose this option, Adaptive Server Enterprise and SQL Anywhere considers the following two strings equal:

```
'ignore the trailing blanks   '
'ignore the trailing blanks'
```

If you do not choose this option, SQL Anywhere considers the two strings above different.

A side effect of choosing this option is that strings are padded with blanks when fetched by a client application.

Remove historical system views

Older versions of SQL Anywhere employed two system views whose names conflict with the Adaptive Server Enterprise system views provided for compatibility. These views include SYSCOLUMNS and SYSINDEXES. If you are using Open Client or JDBC interfaces, create your database excluding these views. You can do this with the dbinit -k option.

If you do not use this option when creating your database, executing the statement `SELECT * FROM SYSCOLUMNS;` results in the error, `Table name 'SYSCOLUMNS' is ambiguous`.

Setting options for Transact-SQL compatibility

You set SQL Anywhere database options using the SET OPTION statement. Several database option settings are relevant to Transact-SQL behavior.

Set the allow_nulls_by_default option

By default, Adaptive Server Enterprise disallows NULLs on new columns unless you explicitly define the column to allow NULLs. SQL Anywhere permits NULL in new columns by default, which is compatible with the SQL/2008 ISO standard.

To make Adaptive Server Enterprise behave in a SQL/2008-compatible manner, use the sp_dboption system procedure to set the allow_nulls_by_default option to true.

To make SQL Anywhere behave in a Transact-SQL-compatible manner, set the allow_nulls_by_default option to Off. You can do this using the SET OPTION statement as follows:

```
SET OPTION PUBLIC.allow_nulls_by_default = 'Off';
```

Set the quoted_identifier option

By default, Adaptive Server Enterprise treats identifiers and strings differently than SQL Anywhere, which matches the SQL/2008 ISO standard.

The quoted_identifier option is available in both Adaptive Server Enterprise and SQL Anywhere. Ensure the option is set to the same value in both databases, for identifiers and strings to be treated in a compatible manner. See “quoted_identifier option” [*SQL Anywhere Server - Database Administration*].

For SQL/2008 behavior, set the quoted_identifier option to On in both Adaptive Server Enterprise and SQL Anywhere.

For Transact-SQL behavior, set the quoted_identifier option to Off in both Adaptive Server Enterprise and SQL Anywhere. If you choose this, you can no longer use identifiers that are the same as keywords, enclosed in double quotes. As an alternative to setting quoted_identifier to Off, ensure that all strings used in SQL statements in your application are enclosed in single quotes, not double quotes.

Set the string_rtruncation option

Both Adaptive Server Enterprise and SQL Anywhere support the string_rtruncation option, which affects error message reporting when an INSERT or UPDATE string is truncated. Ensure that each database has the option set to the same value. See "string_rtruncation option" [*SQL Anywhere Server - Database Administration*].

See "Compatibility options" [*SQL Anywhere Server - Database Administration*].

Case sensitivity

Case sensitivity in databases refers to:

- **Data** The case sensitivity of the data is reflected in indexes and so on.
- **Identifiers** Identifiers include table names, column names, and so on.
- **Passwords** Passwords are always case sensitive in SQL Anywhere databases.

Case sensitivity of data

You decide the case-sensitivity of SQL Anywhere data in comparisons when you create the database. By default, SQL Anywhere databases are case-insensitive in comparisons, although data is always held in the case in which you enter it.

Adaptive Server Enterprise's sensitivity to case depends on the sort order installed on the Adaptive Server Enterprise system. Case sensitivity can be changed for single-byte character sets by reconfiguring the Adaptive Server Enterprise sort order.

Case sensitivity of identifiers

SQL Anywhere does not support case sensitive identifiers. In Adaptive Server Enterprise, the case sensitivity of identifiers follows the case sensitivity of the data. The default user ID for SQL Anywhere databases is DBA.

In Adaptive Server Enterprise, domain names are case sensitive. In SQL Anywhere, they are case insensitive, with the exception of Java data types.

Case sensitivity of passwords

In SQL Anywhere, passwords are always case sensitive. The default password for the DBA user ID is **sql** in lowercase letters.

In Adaptive Server Enterprise, the case sensitivity of user IDs and passwords follows the case sensitivity of the server.

Ensuring compatible object names

Each database object must have a unique name within a certain **name space**. Outside this name space, duplicate names are allowed. Some database objects occupy different name spaces in Adaptive Server Enterprise and SQL Anywhere.

Adaptive Server Enterprise has a more restrictive name space on trigger names than SQL Anywhere. Trigger names must be unique in the database. For compatible SQL, you should stay within the Adaptive Server Enterprise restriction and make your trigger names unique in the database.

The special Transact-SQL timestamp column and data type

SQL Anywhere supports the Transact-SQL special timestamp column. The timestamp column, together with the TSEQUAL system function, checks whether a row has been updated.

Two meanings of timestamp
SQL Anywhere has a TIMESTAMP data type, which holds accurate date and time information. It is distinct from the special Transact-SQL TIMESTAMP column and data type.

Creating a Transact-SQL timestamp column in SQL Anywhere

To create a Transact-SQL timestamp column, create a column that has the (SQL Anywhere) data type TIMESTAMP and a default setting of timestamp. The column can have any name, although the name timestamp is common.

For example, the following CREATE TABLE statement includes a Transact-SQL timestamp column:

```
CREATE TABLE tablename (
   column_1 INTEGER,
   column_2 TIMESTAMP DEFAULT TIMESTAMP
);
```

The following ALTER TABLE statement adds a Transact-SQL timestamp column to the SalesOrders table:

```
ALTER TABLE SalesOrders
ADD timestamp TIMESTAMP DEFAULT TIMESTAMP;
```

In Adaptive Server Enterprise a column with the name timestamp and no data type specified automatically receives a TIMESTAMP data type. In SQL Anywhere you must explicitly assign the data type yourself.

The data type of a timestamp column

Adaptive Server Enterprise treats a timestamp column as a domain that is VARBINARY(8), allowing NULL, while SQL Anywhere treats a timestamp column as the TIMESTAMP data type, which consists of the date and time, with fractions of a second held to six decimal places.

When fetching from the table for later updates, the variable into which the timestamp value is fetched should correspond to the column description.

In Interactive SQL, you may need to set the timestamp_format option to see the differences in values for the rows. The following statement sets the timestamp_format option to display all six digits in the fractions of a second:

```
SET OPTION timestamp_format='YYYY-MM-DD HH:NN:SS.SSSSSS';
```

If all six digits are not shown, some timestamp column values may appear to be equal: they are not.

Using TSEQUAL for updates

With the TSEQUAL system function you can tell whether a timestamp column has been updated or not.

For example, an application may SELECT a timestamp column into a variable. When an UPDATE of one of the selected rows is submitted, it can use the TSEQUAL function to check whether the row has been modified. The TSEQUAL function compares the timestamp value in the table with the timestamp value obtained in the SELECT. Identical timestamps means there are no changes. If the timestamps differ, the row has been changed since the SELECT was performed.

An application may SELECT a timestamp column into a variable. When an UPDATE of one of the selected rows is submitted, it can use the TSEQUAL function to check whether the row has been modified. The TSEQUAL function compares the timestamp value in the table with the timestamp value obtained in the SELECT. Identical timestamps means there are no changes. If the timestamps differ, the row has been changed since the SELECT was performed. For example:

```
CREATE VARIABLE old_ts_value TIMESTAMP;

SELECT timestamp into old_ts_value
FROM publishers
WHERE pub_id = '0736';

UPDATE publishers
SET city = 'Springfield'
WHERE pub_id = '0736'
AND TSEQUAL(timestamp, old_ts_value);
```

The special IDENTITY column

The IDENTITY column stores sequential numbers, such as invoice numbers or employee numbers, which are automatically generated. The value of the IDENTITY column uniquely identifies each row in a table.

In Adaptive Server Enterprise, each table in a database can have one IDENTITY column. The data type must be numeric with scale zero, and the IDENTITY column should not allow nulls.

In SQL Anywhere, the IDENTITY column is a column default setting. You can explicitly insert values that are not part of the sequence into the column with an INSERT statement. Adaptive Server Enterprise does not allow INSERTs into identity columns unless the identity_insert option is *on*. In SQL Anywhere, you need to set the NOT NULL property yourself and ensure that only one column is an IDENTITY column. SQL Anywhere allows any numeric data type to be an IDENTITY column. The use of integer data types is recommended for better performance.

In SQL Anywhere, the IDENTITY column and the AUTOINCREMENT default setting for a column are identical.

To create an IDENTITY column, use the following CREATE TABLE syntax, where *n* is large enough to hold the value of the maximum number of rows that may be inserted into the table.:

```
CREATE TABLE table-name (
   ...
   column-name numeric(n,0) IDENTITY NOT NULL,
   ...
)
```

Retrieving IDENTITY column values with @@identity

The first time you insert a row into the table, an IDENTITY column has a value of 1 assigned to it. On each subsequent insert, the value of the column increases by one. The value most recently inserted into an identity column is available in the @@identity global variable.

For more information about the behavior of @@identity, see "@@identity global variable" [*SQL Anywhere Server - SQL Reference*].

Writing compatible SQL statements

This section describes general guidelines for writing SQL for use on more than one database management system, and discusses compatibility issues between Adaptive Server Enterprise and SQL Anywhere at the SQL statement level.

General guidelines for writing portable SQL

When writing SQL for use on more than one database management system, make your SQL statements as explicit as possible. Even if more than one server supports a given SQL statement, it may be a mistake to assume that default behavior is the same on each system.

In SQL Anywhere, the database server and the SQL preprocessor (sqlpp) can identify SQL statements that are vendor extensions, are not compliant with specific ISO/ANSI SQL standards, or are not supported by UltraLite. This functionality is called the SQL Flagger. See "Testing SQL compliance using the SQL Flagger" on page 631.

General guidelines applicable to writing compatible SQL include:

- Include all the available options, rather than using default behavior.
- Use parentheses to make the order of execution within statements explicit, rather than assuming identical default order of precedence for operators.

- Use the Transact-SQL convention of an @ sign preceding variable names for Adaptive Server Enterprise portability.
- Declare variables and cursors in procedures, triggers, and batches immediately following a BEGIN statement. SQL Anywhere requires this, although Adaptive Server Enterprise allows declarations to be made anywhere in a procedure, trigger, or batch.
- Avoid using reserved words from either Adaptive Server Enterprise or SQL Anywhere as identifiers in your databases.
- Assume large namespaces. For example, ensure that each index should have a unique name.

Creating compatible tables

SQL Anywhere supports domains which allow constraint and default definitions to be encapsulated in the data type definition. It also supports explicit defaults and CHECK conditions in the CREATE TABLE statement. It does not, however, support named defaults.

NULL

SQL Anywhere and Adaptive Server Enterprise differ in some respects in their treatment of NULL. In Adaptive Server Enterprise, NULL is sometimes treated as if it were a value.

For example, a unique index in Adaptive Server Enterprise cannot contain rows that hold null and are otherwise identical. In SQL Anywhere, a unique index can contain such rows.

By default, columns in Adaptive Server Enterprise default to NOT NULL, whereas in SQL Anywhere the default setting is NULL. You can control this setting using the allow_nulls_by_default option. Specify explicitly NULL or NOT NULL to make your data definition statements transferable.

For information about this option, see "Setting options for Transact-SQL compatibility" on page 647.

Temporary tables

You can create a temporary table by placing a pound sign (#) in front of the table name in a CREATE TABLE statement. These temporary tables are SQL Anywhere declared temporary tables, and are available only in the current connection.

Physical placement of a table is performed differently in Adaptive Server Enterprise and in SQL Anywhere. SQL Anywhere supports the **ON** *segment-name* clause, but *segment-name* refers to a SQL Anywhere dbspace.

See also

- "Testing SQL compliance using the SQL Flagger" on page 631
- "DECLARE LOCAL TEMPORARY TABLE statement" [*SQL Anywhere Server - SQL Reference*]
- "CREATE TABLE statement" [*SQL Anywhere Server - SQL Reference*]

Writing compatible queries

There are two criteria for writing a query that runs on both SQL Anywhere and Adaptive Server Enterprise databases:

- The data types, expressions, and search conditions in the query must be compatible.
- The syntax of the statement itself must be compatible.

This section assumes compatible data types, expressions, and search conditions. The examples assume the quoted_identifier option is set to OFF, which is the default Adaptive Server Enterprise setting, but not the default SQL Anywhere setting.

SQL Anywhere's implementation of the Transact-SQL dialect supports much of the query expression syntax from the Watcom SQL dialect, even though some of these SQL constructions are not supported by Adaptive Server Enterprise. In a Transact-SQL query, SQL Anywhere supports the following SQL constructions:

- the back-quote character `, the double-quote character ", and square parentheses [] to denote identifiers
- UNION, EXCEPT, and INTERSECT query expressions
- derived tables
- table functions
- CONTAINS table expressions for full text search
- REGEXP, SIMILAR, IS DISTINCT FROM, and CONTAINS predicates
- user-defined SQL or external functions
- LEFT, RIGHT and FULL outer joins
- GROUP BY ROLLUP, CUBE, and GROUPING SETS
- TOP N START AT M
- window aggregate functions and other analytic functions including statistical analysis and linear regression functions

To summarize, SQL Anywhere's Transact-SQL dialect supports the following:

Syntax

query-expression:
{ *query-expression* **EXCEPT** [**ALL**] *query-expression*
| *query-expression* **INTERSECT** [**ALL**] *query-expression*
| *query-expression* **UNION** [**ALL**] *query-expression*
| *query-specification* }
[**ORDER BY** { *expression* | *integer* }
    [**ASC** | **DESC**], ...]
[**FOR READ ONLY** | *for-update-clause*]
[**FOR XML** *xml-mode*]

query-specification:
SELECT [**ALL** | **DISTINCT**] [*cursor-range*] *select-list*
[**INTO** *#temporary-table-name*]
[**FROM** *table-expression*, ...]
[**WHERE** *search-condition*]
[**GROUP BY** *group-by-term*, ...]

[**HAVING** *search-condition*]
[**WINDOW** *window-specification*, ...]

Parameters

select-list:
*table-name.**
| ***
| *expression*
| *alias-name* = *expression*
| *expression* as *identifier*
| *expression* as *string*

table-expression: See "FROM clause" [*SQL Anywhere Server - SQL Reference*].

group-by-term: See "GROUP BY clause" [*SQL Anywhere Server - SQL Reference*].

for-update-clause: See "FOR UPDATE or FOR READ ONLY clause, SELECT statement" [*SQL Anywhere Server - SQL Reference*].

xml-mode: See "SELECT statement" [*SQL Anywhere Server - SQL Reference*].

alias-name:
identifier | '*string*' | "*string*" | `*string*`

cursor-range:
{ **FIRST** | **TOP***constant-or-variable* } [**START AT** *constant-or-variable*]

Transact-SQL-table-reference:
[*owner* .]*table-name* [[**AS**] *correlation-name*]
[**(** **INDEX** *index_name* [**PREFETCH** *size*][**LRU** | **MRU**] **)**]

Notes

- In addition to the Watcom SQL syntax for the FROM clause, SQL Anywhere supports Transact-SQL syntax for specific Adaptive Server Enterprise table hints. For a table reference, *Transact-SQL-table-reference* supports the INDEX hint keyword, along with the PREFETCH, MRU and LRU caching hints. PREFETCH, MRU and LRU are ignored in SQL Anywhere.

- SQL Anywhere does not support the Transact-SQL extension to the GROUP BY clause allowing references to columns that are not included in the GROUP BY clause. See "GROUP BY and the SQL/2008 standard" on page 375. SQL Anywhere also does not support the Transact-SQL GROUP BY ALL construction.

- SQL Anywhere supports a subset of Transact-SQL outer join constructions using the comparison operators *= and =*. For more information about Transact-SQL outer joins, see "Transact-SQL outer joins (*= or =*)" on page 403.

- SQL Anywhere's Transact-SQL dialect does not support common table expressions except when embedded within a derived table. Consequently SQL Anywhere's Transact-SQL dialect does not support recursive UNION queries. Use the Watcom SQL dialect if you require this functionality.

- The performance parameters part of the table specification is parsed, but has no effect.

- The HOLDLOCK keyword is supported by SQL Anywhere. With HOLDLOCK, a shared lock on a specified table or view is more restrictive because the shared lock is not released when the data page is no longer needed. The query is performed at isolation level 3 on a table on which the HOLDLOCK is specified.

- The HOLDLOCK option applies only to the table or view for which it is specified, and only for the duration of the transaction defined by the statement in which it is used. Setting the isolation level to 3 applies a holdlock for each select within a transaction. You cannot specify both a HOLDLOCK and NOHOLDLOCK option in a query.

- The NOHOLDLOCK keyword is recognized by SQL Anywhere, but has no effect.

- Transact-SQL uses the SELECT statement to assign values to local variables:

  ```
  SELECT @localvar = 42;
  ```

  The corresponding statement in SQL Anywhere is the SET statement:

  ```
  SET @localvar = 42;
  ```

- Adaptive Server Enterprise does not support the following:

  - SELECT ... INTO *host-variable-list*
  - SELECT ... INTO *variable-list*
  - EXCEPT [ALL] or INTERSECT [ALL]
  - START AT clause
  - SQL Anywhere-defined table hints - see "FROM clause" [*SQL Anywhere Server - SQL Reference*]
  - table functions
  - FULL OUTER JOIN
  - FOR UPDATE BY { LOCK | TIMESTAMP }
  - window aggregate functions and linear regression functions

- SQL Anywhere does not support the following keywords and clauses of the Adaptive Server Enterprise Transact-SQL SELECT statement syntax:

  - SHARED keyword
  - PARTITION keyword
  - COMPUTE clause
  - FOR BROWSE clause
  - GROUP BY ALL clause
  - PLAN clause
  - ISOLATION clause

See also

- "Testing SQL compliance using the SQL Flagger" on page 631
- "SELECT statement" [*SQL Anywhere Server - SQL Reference*]
- "OLAP support" on page 445

Compatibility of joins

In SQL Anywhere's implementation of Transact-SQL, you can specify join syntax from the SQL/2008 standard using the keywords JOIN, LEFT OUTER JOIN, and RIGHT OUTER JOIN, and FULL OUTER JOIN, along with legacy Transact-SQL outer join syntax that uses the specialty comparison operators *= and =* in the statement's WHERE clause.

Note
Support for Transact-SQL outer join operators *= and =* is deprecated and will be removed in a future release.

For information about joins in SQL Anywhere and in the ANSI/ISO SQL standards, see "Joins: Retrieving data from several tables" on page 386, and "FROM clause" [*SQL Anywhere Server - SQL Reference*].

For more information about Transact-SQL outer joins, see "Transact-SQL outer joins (*= or =*)" on page 403.

See also

- "Testing SQL compliance using the SQL Flagger" on page 631

Transact-SQL procedure language overview

The **stored procedure language** is the part of SQL used in stored procedures, triggers, and batches.

SQL Anywhere supports a large part of the Transact-SQL stored procedure language in addition to the Watcom SQL dialect based on SQL/2008.

Transact-SQL stored procedure overview

The native SQL Anywhere dialect, Watcom-SQL, is based on the ISO/ANSI SQL/2008 standard. Consequently, the Watcom-SQL stored procedure dialect differs from the Transact-SQL dialect in many ways. Many of the concepts and features are similar, but the syntax is different. SQL Anywhere support for Transact-SQL takes advantage of the similar concepts by providing automatic translation between dialects. However, a procedure must be written exclusively in one of the two dialects, not in a mixture of the two.

SQL Anywhere support for Transact-SQL stored procedures

There are a variety of aspects to SQL Anywhere support for Transact-SQL stored procedures, including:

- Passing parameters
- Returning result sets
- Returning status information
- Providing default values for parameters
- Control statements
- Error handling
- User-defined functions

Transact-SQL trigger overview

Trigger compatibility requires compatibility of trigger features and syntax. This section provides an overview of the feature compatibility of Transact-SQL and SQL Anywhere triggers.

Adaptive Server Enterprise supports statement-level AFTER triggers; that is, triggers that execute after the triggering statement has completed. The Watcom-SQL dialect supported by SQL Anywhere supports row-level BEFORE, AFTER, and INSTEAD OF triggers, and statement-level AFTER and INSTEAD OF triggers. See "Introduction to triggers" on page 803.

Row-level triggers are not part of the Transact-SQL compatibility features, and are discussed in "Using procedures, triggers, and batches" on page 793.

Description of unsupported or different Transact-SQL triggers

Features of Transact-SQL triggers that are either unsupported or different in SQL Anywhere include:

- **Triggers firing other triggers** Suppose a trigger performs an action that would, if performed directly by a user, fire another trigger. SQL Anywhere and Adaptive Server Enterprise respond slightly differently to this situation. By default in Adaptive Server Enterprise, triggers fire other triggers up to a configurable nesting level, which has the default value of 16. You can control the nesting level with the Adaptive Server Enterprise nested triggers option. In SQL Anywhere, triggers fire other triggers without limit unless there is insufficient memory.

- **Triggers firing themselves** Suppose a trigger performs an action that would, if performed directly by a user, fire the same trigger. SQL Anywhere and Adaptive Server Enterprise respond slightly differently to this situation. By default, in SQL Anywhere, non-Transact-SQL triggers fire themselves recursively, whereas Transact-SQL dialect triggers do not fire themselves recursively. However, for Transact-SQL dialect triggers, you can use the self_recursion option of the SET statement [T-SQL] to allow a trigger to call itself recursively. See "SET statement [T-SQL]" [*SQL Anywhere Server - SQL Reference*].

  By default in Adaptive Server Enterprise, a trigger does not call itself recursively, but you can use the self_recursion option to allow recursion to occur.

- **ROLLBACK statement in triggers in triggers not supported** Adaptive Server Enterprise permits the ROLLBACK TRANSACTION statement within triggers, to roll back the entire

transaction of which the trigger is a part. SQL Anywhere does not permit ROLLBACK (or ROLLBACK TRANSACTION) statements in triggers because a triggering action and its trigger together form an atomic statement.

SQL Anywhere does provide the Adaptive Server Enterprise-compatible ROLLBACK TRIGGER statement to undo actions within triggers. See "ROLLBACK TRIGGER statement" [*SQL Anywhere Server - SQL Reference*].

- **ORDER clause not supported** Transact-SQL triggers do not permit an ORDER *nn* clause; the value of trigger_order is automatically set to 1. This can cause an error to be returned creating a T-SQL trigger if there is already a statement level trigger. This is because the SYSTRIGGER system table has a unique index on table_id, event, trigger_time, trigger_order. For a particular event (insert, update, delete) statement-level triggers are always AFTER and trigger_order cannot be set, so there can be only one per table, assuming any other triggers do not set an order other than 1.

Transact-SQL batch overview

In Transact-SQL, a batch is a set of SQL statements submitted together and executed as a group, one after the other. Batches can be stored in command files. Interactive SQL in SQL Anywhere and the Interactive SQL utility in Adaptive Server Enterprise provide similar capabilities for executing batches interactively.

The control statements used in procedures can also be used in batches. SQL Anywhere supports the use of control statements in batches and the Transact-SQL-like use of non-delimited groups of statements terminated with a go statement to signify the end of a batch.

For batches stored in command files, SQL Anywhere supports the use of parameters in command files. See "PARAMETERS statement [Interactive SQL]" [*SQL Anywhere Server - SQL Reference*].

Automatic translation of stored procedures

In addition to supporting Transact-SQL alternative syntax, SQL Anywhere provides aids for translating statements between the Watcom SQL and Transact-SQL dialects. SQL language built-in functions returning information about SQL statements and enabling automatic translation of SQL statements include:

- **SQLDialect(statement)** Returns Watcom-SQL or Transact-SQL.
- **Watcom SQL(statement)** Returns the Watcom-SQL syntax for the statement.
- **TransactSQL(statement)** Returns the Transact-SQL syntax for the statement.

These are functions, and so can be accessed using a select statement from Interactive SQL. For example, the following statement returns the value Watcom-SQL:

```
SELECT SQLDialect( 'SELECT * FROM Employees' );
```


Using Sybase Central to translate stored procedures

Sybase Central has facilities for creating, viewing, and altering procedures and triggers. You can also export the text to a file for editing outside Sybase Central.

To translate a stored procedure (Sybase Central)

1. Use the SQL Anywhere 12 plug-in to connect to the database as a user with DBA authority or as the owner of the procedure.
2. Open the **Procedures & Functions** folder.
3. In the right pane, click the **SQL** tab and then click in the editor.
4. Choose **File** and select one of the **Translate To** commands.

   The procedure appears in the right pane in the selected dialect. If the selected dialect is not the one in which the procedure is stored, the server translates it to that dialect. Any untranslated lines appear as comments.
5. Rewrite any untranslated lines.
6. Choose **File** » **Save**.

Returning result sets from Transact-SQL procedures

SQL Anywhere 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.

Example of a Transact-SQL procedure

The following Transact-SQL procedure illustrates how Transact-SQL stored procedures returns result sets:

```
CREATE PROCEDURE ShowDepartment (@deptname VARCHAR(30))
AS
   SELECT Employees.Surname, Employees.GivenName
   FROM Departments, Employees
   WHERE Departments.DepartmentName = @deptname
   AND Departments.DepartmentID = Employees.DepartmentID;
```

Example of a Watcom SQL procedure

The following is the corresponding SQL Anywhere procedure:

```
CREATE PROCEDURE ShowDepartment(in deptname VARCHAR(30))
RESULT ( LastName CHAR(20), FirstName CHAR(20))
BEGIN
   SELECT Employees.Surname, Employees.GivenName
   FROM Departments, Employees
   WHERE Departments.DepartmentName = deptname
   AND Departments.DepartmentID = Employees.DepartmentID
END;
```

For more information about procedures and results, see "Returning results from procedures" on page 820.

Variables in Transact-SQL procedures

SQL Anywhere uses the SET statement to assign values to variables in a procedure. In Transact-SQL, values are assigned using either the SELECT statement with an empty table-list, or the SET statement. The following simple procedure illustrates how the Transact-SQL syntax works:

```
CREATE PROCEDURE multiply
                @mult1 int,
                @mult2 int,
                @result int output
AS
SELECT @result = @mult1 * @mult2;
```

This procedure can be called as follows:

```
CREATE VARIABLE @product int
go
EXECUTE multiply 5, 6, @product OUTPUT
go
```

The variable @product has a value of 30 after the procedure executes.

For more information about using the SELECT statement to assign variables, see "Writing compatible queries" on page 653. For more information about using the SET statement to assign variables, see "SET statement" [*SQL Anywhere Server - SQL Reference*].

Error handling in Transact-SQL procedures

Default procedure error handling is different in the Watcom SQL and Transact-SQL dialects. By default, Watcom SQL dialect procedures exit when they encounter an error, returning SQLSTATE and SQLCODE values to the calling environment.

Explicit error handling can be built into Watcom SQL stored procedures using the EXCEPTION statement, or you can instruct the procedure to continue execution at the next statement when it encounters an error, using the ON EXCEPTION RESUME statement.

When a Transact-SQL dialect procedure encounters an error, execution continues at the following statement. The global variable @@error holds the error status of the most recently executed statement. You can check this variable following a statement to force return from a procedure. For example, the following statement causes an exit if an error occurs.

```
IF @@error != 0 RETURN
```

When the procedure completes execution, a return value indicates the success or failure of the procedure. This return status is an integer, and can be accessed as follows:


```
DECLARE @Status INT
EXECUTE @Status = proc_sample
IF @Status = 0
   PRINT 'procedure succeeded'
ELSE
   PRINT 'procedure failed'
```

The following table describes the built-in procedure return values and their meanings:

Value	Definition	SQL Anywhere SQLSTATE
0	Procedure executed without error	
-1	Missing object	42W33, 52W02, 52003, 52W07, 42W05
-2	Data type error	53018
-3	Process was chosen as deadlock victim	40001, 40W06
-4	Permission error	42501
-5	Syntax error	42W04
-6	Miscellaneous user error	
-7	Resource error, such as out of space	08W26
-10	Fatal internal inconsistency	40W01
-11	Fatal internal inconsistency	40000
-13	Database is corrupt	WI004
-14	Hardware error	08W17, 40W03, 40W04

When a SQL Anywhere SQLSTATE is not applicable, the default value -6 is returned.

The RETURN statement can be used to return other integers, with their own user-defined meanings.

Using the RAISERROR statement in procedures

You can use the RAISERROR statement to generate user-defined errors. The RAISERROR statement functions similar to the SIGNAL statement. See "RAISERROR statement" [*SQL Anywhere Server - SQL Reference*].

By itself, the RAISERROR statement does not cause an exit from the procedure, but it can be combined with a RETURN statement or a test of the @@error global variable to control execution following a user-defined error.

If you set the on_tsql_error database option to Continue, the RAISERROR statement no longer signals an execution-ending error. Instead, the procedure completes and stores the RAISERROR status code and message, and returns the most recent RAISERROR. If the procedure causing the RAISERROR was called from another procedure, the RAISERROR returns after the outermost calling procedure terminates. If you set the on_tsql_error option to the default (Conditional), the continue_after_raiserror option controls the behavior following the execution of a RAISERROR statement. If you set the on_tsql_error option to Stop or Continue, the on_tsql_error setting takes precedence over the continue_after_raiserror setting.

You lose intermediate RAISERROR statuses and codes after the procedure terminates. If, at return time, an error occurs along with the RAISERROR, then the error information is returned and you lose the RAISERROR information. The application can query intermediate RAISERROR statuses by examining @@error global variable at different execution points.

Transact-SQL-like error handling in the Watcom SQL dialect

You can make a Watcom SQL dialect procedure handle errors in a Transact-SQL-like manner by supplying the ON EXCEPTION RESUME clause to the CREATE PROCEDURE statement:

```
CREATE PROCEDURE sample_proc()
ON EXCEPTION RESUME
BEGIN
   ...
END
```

The presence of an ON EXCEPTION RESUME clause prevents explicit exception handling code from being executed, so avoid this clause with explicit error handling See “CREATE PROCEDURE statement” [*SQL Anywhere Server - SQL Reference*].

Using XML in the database

Extensible Markup Language (XML) represents structured data in text format. XML was designed specifically to meet the challenges of large-scale electronic publishing.

XML is a simple markup language, like HTML, but is also flexible, like SGML. XML is hierarchical, and its main purpose is to describe the structure of data for both humans and computer software to author and read.

Rather than providing a static set of elements which describe various forms of data, XML lets you define elements. As a result, many types of structured data can be described with XML. XML documents can optionally use a document type definition (DTD) or XML schema to define the structure, elements, and attributes that are used in an XML file.

There are several ways you can use XML with SQL Anywhere:

- Storing XML documents in the database
- Exporting relational data as XML
- Importing XML into the database
- Querying relational data as XML

For more details about XML, see http://www.w3.org/XML/.

Storing XML documents in relational databases

SQL Anywhere supports two data types that can be used to store XML documents in your database: the XML data type and the LONG VARCHAR data type. Both of these data types store the XML document as a string in the database.

The XML data type uses the character set encoding of the database server. The XML encoding attribute should match the encoding used by the database server. The XML encoding attribute does not specify how the automatic character set conversion is completed.

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.

When you generate elements from relational data, any characters that are invalid in XML are escaped unless the data is of type XML. For example, suppose you want to generate a <product> element with the following content so that the element content contains less than and greater than signs:

```
<hat>bowler</hat>
```

If you write a query that specifies that the element content is of type XML, then the greater than and less than signs are not quoted, as follows:

```
SELECT XMLFOREST( CAST( '<hat>bowler</hat>' AS XML )
AS product );
```

You get the following result:

```
<product><hat>bowler</hat></product>
```

However, if the query does not specify that the element content is of type XML, for example:

```
SELECT XMLFOREST( '<hat>bowler</hat>' AS product );
```

In this case, the less than and greater than signs are replaced with entity references as follows:

```
<product>&lt;hat&gt;bowler&lt;/hat&gt;</product>
```

Note that attributes are always quoted, regardless of the data type.

For more information about how element content is escaped, see "Encoding illegal XML names" on page 675.

For more information about the XML data type, see "XML data type" [*SQL Anywhere Server - SQL Reference*].

Exporting relational data as XML

SQL Anywhere provides two ways to export your relational data as XML: the Interactive SQL OUTPUT statement and the ADO.NET DataSet object.

The FOR XML clause and SQL/XML functions allow you to generate a result set as XML from the relational data in your database. You can then export the generated XML to a file using the UNLOAD statement or the xp_write_file system procedure.

Exporting relational data as XML from Interactive SQL

The Interactive SQL OUTPUT statement supports an XML format that outputs query results to a generated XML file.

This generated XML file is encoded in UTF-8 and contains an embedded DTD. In the XML file, binary values are encoded in character data (CDATA) blocks with the binary data rendered as 2-hex-digit strings.

For more information about exporting XML with the OUTPUT statement, see "OUTPUT statement [Interactive SQL]" [*SQL Anywhere Server - SQL Reference*].

The INPUT statement does not accept XML as a file format. However, you can import XML using the openxml procedure or the ADO.NET DataSet object.

For more information about importing XML, see "Importing XML documents as relational data" on page 665.

Exporting relational data as XML using the DataSet object

The ADO.NET DataSet object allows you to save the contents of the DataSet in an XML document. Once you have filled the DataSet (for example, with the results of a query on your database) you can save either the schema or both the schema and data from the DataSet in an XML file. The WriteXml method saves both the schema and data in an XML file, while the WriteXmlSchema method saves only the schema in an XML file. You can fill a DataSet object using the SQL Anywhere ADO.NET data provider.

For information about exporting relational data as XML using a DataSet, see "Inserting, updating, and deleting rows using the SACommand object" [*SQL Anywhere Server - Programming*].

Importing XML documents as relational data

SQL Anywhere supports two different ways to import XML into your database:

- using the openxml procedure to generate a result set from an XML document
- using the ADO.NET DataSet object to read the data and/or schema from an XML document into a DataSet

Importing XML using openxml

The openxml procedure is used in the FROM clause of a query to generate a result set from an XML document. openxml uses a subset of the XPath query language to select nodes from an XML document.

Using XPath expressions

When you use openxml, the XML document is parsed and the result is modeled as a tree. The tree is made up of nodes. XPath expressions are used to select nodes in the tree. The following list describes some commonly-used XPath expressions:

- ***/*** indicates the root node of the XML document
- ***//*** indicates all descendants of the root, including the root node
- **. (single period)** indicates the current node of the XML document
- ***.//*** indicates all descendants of the current node, including the current node
- **..** indicates the parent node of the current node
- ***./@attributename*** indicates the attribute of the current node having the name *attributename*
- ***./childname*** indicates the children of the current node that are elements having the name *childname*

Consider the following XML document:

```
<inventory>
  <product ID="301" size="Medium">Tee Shirt
```

```
    <quantity>54</quantity>
  </product>
  <product ID="302" size="One Size fits all">Tee Shirt
    <quantity>75</quantity>
  </product>
  <product ID="400" size="One Size fits all">Baseball Cap
    <quantity>112</quantity>
  </product>
</inventory>
```

The <inventory> element is the root node. You can refer to it using the following XPath expression:

```
/inventory
```

Suppose that the current node is a <quantity> element. You can refer to this node using the following XPath expression:

```
.
```

To find all the <product> elements that are children of the <inventory> element, use the following XPath expression:

```
/inventory/product
```

If the current node is a <product> element and you want to refer to the size attribute, use the following XPath expression:

```
./@size
```

For a complete list of XPath syntax supported by openxml, see "openxml system procedure" [*SQL Anywhere Server - SQL Reference*].

For information about the XPath query language, see http://www.w3.org/TR/xpath.

Generating a result set using openxml

Each match for the first *xpath-query* argument to openxml generates one row in the result set. The WITH clause specifies the schema of the result set and how the value is found for each column in the result set. For example, consider the following query:

```
SELECT * FROM openxml( '<inventory>
                          <product>Tee Shirt
                            <quantity>54</quantity>
                            <color>Orange</color>
                          </product>
                          <product>Baseball Cap
                  <quantity>112</quantity>
                            <color>Black</color>
                          </product>
                        </inventory>',
                       '/inventory/product' )
WITH ( Name CHAR (25) './text()',
       Quantity CHAR(3) 'quantity',
       Color CHAR(20) 'color');
```

The first *xpath-query* argument is /inventory/product, and there are two <product> elements in the XML, so two rows are generated by this query.

The WITH clause specifies that there are three columns: Name, Quantity, and Color. The values for these columns are taken from the <product>, <quantity> and <color> elements. The query above generates the following result:

Name	Quantity	Color
Tee Shirt	54	Orange
Baseball Cap	112	Black

For more information, see "openxml system procedure" [*SQL Anywhere Server - SQL Reference*].

Using openxml to generate an edge table

The openxml procedure can be used to generate an edge table, a table that contains a row for every element in the XML document. You may want to generate an edge table so that you can query the data in the result set using SQL.

The following SQL statement creates a variable, x, that contains an XML document. The XML generated by the query has a root element called <root>, which is generated using the XMLELEMENT function, and elements are generated for each column in the Employees, SalesOrders, and Customers tables using FOR XML AUTO with the ELEMENTS modifier specified.

For information about the XMLELEMENT function, see "XMLELEMENT function [String]" [*SQL Anywhere Server - SQL Reference*].

For information about FOR XML AUTO, see "Using FOR XML AUTO" on page 678.

```
CREATE VARIABLE x XML;
SET x=(SELECT XMLELEMENT( NAME root,
         (SELECT * FROM Employees
         KEY JOIN SalesOrders
         KEY JOIN Customers
         FOR XML AUTO, ELEMENTS)));
SELECT x;
```

The generated XML looks as follows (the result has been formatted to make it easier to read—the result returned by the query is one continuous string):

```
<root>
 <Employees>
  <EmployeeID>299</EmployeeID>
  <ManagerID>902</ManagerID>
  <Surname>Overbey</Surname>
  <GivenName>Rollin</GivenName>
  <DepartmentID>200</DepartmentID>
  <Street>191 Companion Ct.</Street>
  <City>Kanata</City>
  <State>CA</State>
  <Country>USA</Country>
  <PostalCode>94608</PostalCode>
  <Phone>5105557255</Phone>
  <Status>A</Status>
  <SocialSecurityNumber>025487133</SocialSecurityNumber>
  <Salary>39300.000</Salary>
  <StartDate>1987-02-19</StartDate>
```

```
  <BirthDate>1964-03-15</BirthDate>
  <BenefitHealthInsurance>Y</BenefitHealthInsurance>
  <BenefitLifeInsurance>Y</BenefitLifeInsurance>
  <BenefitDayCare>N</BenefitDayCare>
  <Sex>M</Sex>
  <SalesOrders>
  <ID>2001</ID>
  <CustomerID>101</CustomerID>
  <OrderDate>2000-03-16</OrderDate>
  <FinancialCode>r1</FinancialCode>
  <Region>Eastern</Region>
  <SalesRepresentative>299</SalesRepresentative>
    <Customers>
    <ID>101</ID>
    <Surname>Devlin</Surname>
    <GivenName>Michael</GivenName>
    <Street>114 Pioneer Avenue</Street>
    <City>Kingston</City>
    <State>NJ</State>
    <PostalCode>07070</PostalCode>
    <Phone>2015558966</Phone>
    <CompanyName>The Power Group</CompanyName>
    </Customers>
  </SalesOrders>
</Employees>
...
```

The following query uses the descendant-or-self (//*) XPath expression to match every element in the above XML document, and for each element the id metaproperty is used to obtain an ID for the node, and the parent (../) XPath expression is used with the ID metaproperty to get the parent node. The localname metaproperty is used to obtain the name of each element. Metaproperty names are case sensitive, so ID or LOCALNAME cannot be used as metaproperty names.

```
SELECT * FROM openxml( x, '//*' )
 WITH (ID INT '@mp:id',
       parent INT '../@mp:id',
       name CHAR(25) '@mp:localname',
       text LONG VARCHAR 'text()' )
ORDER BY ID;
```

The result set generated by this query shows the ID of each node, the ID of the parent node, and the name and content for each element in the XML document.

ID	parent	name	text
5	(NULL)	root	(NULL)
16	5	Employees	(NULL)
28	16	EmployeeID	299
55	16	ManagerID	902
79	16	Surname	Overbey
...	...	...	...

Using openxml with xp_read_file

So far, XML that was generated with a procedure like XMLELEMENT has been used. You can also read XML from a file and parse it using the xp_read_file procedure. Suppose the file *c:\inventory.xml* has the following contents:

```
<inventory>
   <product>Tee Shirt
      <quantity>54</quantity>
      <color>Orange</color>
   </product>
   <product>Baseball Cap
      <quantity>112</quantity>
      <color>Black</color>
   </product>
</inventory>
```

You can use the following statement to read and parse the XML in the file:

```
CREATE VARIABLE x XML;
SELECT xp_read_file( 'c:\\inventory.xml' )
 INTO x;
SELECT * FROM openxml( x, '//*' )
 WITH (ID INT '@mp:id',
       parent INT '../@mp:id',
       name CHAR(128) '@mp:localname',
       text LONG VARCHAR 'text()' )
ORDER BY ID;
```

Querying XML in a column

If you have a table with a column that contains XML, you can use openxml to query all the XML values in the column at once. This can be done using a lateral derived table.

The following statements create a table with two columns, ManagerID and Reports. The Reports column contains XML data generated from the Employees table.

```
CREATE TABLE test (ManagerID INT, Reports XML);
INSERT INTO test
SELECT ManagerID, XMLELEMENT( NAME reports,
              XMLAGG( XMLELEMENT( NAME e, EmployeeID)))
FROM Employees
GROUP BY ManagerID;
```

Execute the following query to view the data in the test table:

```
SELECT * FROM test
ORDER BY ManagerID;
```

This query produces the following result:

ManagerID	Reports
501	<reports> <e>102</e> <e>105</e> <e>160</e> <e>243</e> ... </reports>
703	<reports> <e>191</e> <e>750</e> <e>868</e> <e>921</e> ... </reports>
902	<reports> <e>129</e> <e>195</e> <e>299</e> <e>467</e> ... </reports>
1293	<reports> <e>148</e> <e>390</e> <e>586</e> <e>757</e> ... </reports>
...	...

The following query uses a lateral derived table to generate a result set with two columns: one that lists the ID for each manager, and one that lists the ID for each employee that reports to that manager:

```
SELECT ManagerID, EmployeeID
FROM test, LATERAL( openxml( test.Reports, '//e' )
WITH (EmployeeID INT '.') ) DerivedTable
ORDER BY ManagerID, EmployeeID;
```

This query generates the following result:

ManagerID	EmployeeID
501	102
501	105
501	160
501	243

ManagerID	EmployeeID
...	...

For more information about lateral derived tables, see "FROM clause" [*SQL Anywhere Server - SQL Reference*].

Importing XML using the DataSet object

The ADO.NET DataSet object allows you to read the data and/or schema from an XML document into a DataSet.

- The ReadXml method populates a DataSet from an XML document that contains both a schema and data.
- The ReadXmlSchema method reads only the schema from an XML document. Once the DataSet is filled with data from the XML document, you can update the tables in your database with the changes from the DataSet.

DataSet objects can also be manipulated using the SQL Anywhere ADO.NET data provider.

For information about using a DataSet to read the data and/or schema from an XML document using the SQL Anywhere .NET Data Provider, see "Getting data using the SADataAdapter object" [*SQL Anywhere Server - Programming*].

Defining default XML namespaces

You define a default namespace in an element of an XML document with an attribute of the form `xmlns="URI"`. In the following example, a document has a default namespace bound to the URI http://www.iAnywhere.com/EmployeeDemo:

```
<x xmlns="http://www.iAnywhere.com/EmployeeDemo"/>
```

If the element does not have a prefix in its name, a default namespace applies to the element and to any descendant of that element where it is defined. A colon separates a prefix from the rest of the element name. For example, <x/> does not have a prefix, while <p:x/> has the prefix p. You define a namespace that is bound to a prefix with an attribute of the form `xmlns:prefix="URI"`. In the following example, a document binds the prefix p to the same URI as the previous example:

```
<x xmlns:p="http://www.iAnywhere.com/EmployeeDemo"/>
```

Default namespaces are never applied to attributes. Unless it has a prefix, an attribute is always bound to the NULL namespace URI. In the following example, the root and child elements have the iAnywhere1 namespace while the x attribute has the NULL namespace URI and the y element has the iAnywhere2 namespace:

```
<root xmlns="iAnywhere1" xmlns:p="iAnywhere2">
<child x='1' p:y='2' />
</root>
```

The namespaces defined in the root element of the document are applied in the query when you pass an XML document as the *namespace-declaration* argument of an openxml query. All parts of the document after the root element are ignored. In the following example, p1 is bound to iAnywhere1 in the document and bound to p2 in the *namespace-declaration* argument, and the query is able to use the prefix p2:

```
SELECT *
FROM openxml( '<p1:x xmlns:p1="iAnywhere1"> 1 </x>', '/p2:x', 1, '<root
xmlns:p2="iAnywhere1"/>' )
WITH ( c1 int '.' );
```

When matching an element, you must correctly specify the URI that a prefix is bound to. In the example above, the x name in the xpath query matches the x element in the document because they both have the iAnywhere1 namespace.

Do not use a default namespace in the *namespace-declaration* of the openxml system procedure. Use a wildcard query of the form /*:x which matches an x element bound to any URI including the NULL namespace, or bind the URI you want to a specific prefix and use that in the query. For more information about generating result sets from an XML document, see "openxml system procedure" [*SQL Anywhere Server - SQL Reference*].

Obtaining query results as XML

SQL Anywhere supports two different ways to obtain query results from your relational data as XML:

- **FOR XML clause** The FOR XML clause can be used in a SELECT statement to generate an XML document.

  For information about using the FOR XML clause, see "Using the FOR XML clause to retrieve query results as XML" on page 673 and "SELECT statement" [*SQL Anywhere Server - SQL Reference*].

- **SQL/XML** SQL Anywhere supports functions based on the draft SQL/XML standard that generate XML documents from relational data.

  For information about using one or more of these functions in a query, see "Using SQL/XML to obtain query results as XML" on page 689.

The FOR XML clause and the SQL/XML functions supported by SQL Anywhere give you two alternatives for generating XML from your relational data. You can usually use one or the other to generate the same XML.

For example, this query uses FOR XML AUTO to generate XML:

```
SELECT ID, Name
FROM Products
WHERE Color='black'
FOR XML AUTO;
```


The following query uses the XMLELEMENT function to generate XML:

```
SELECT XMLELEMENT(NAME product,
          XMLATTRIBUTES(ID, Name))
FROM Products
WHERE Color='black';
```

Both queries generate the following XML (the result set has been formatted to make it easier to read):

```
<product ID="302" Name="Tee Shirt"/>
<product ID="400" Name="Baseball Cap"/>
<product ID="501" Name="Visor"/>
<product ID="700" Name="Shorts"/>
```

Tip

If you are generating deeply-nested documents, a FOR XML EXPLICIT query will likely be more efficient than a SQL/XML query because EXPLICIT mode queries normally use a UNION to generate nesting, while SQL/XML uses subqueries to generate the required nesting.

Using the FOR XML clause to retrieve query results as XML

SQL Anywhere allows you to execute a SQL query against your database and return the results as an XML document by using the FOR XML clause in your SELECT statement. The XML document is of type XML.

For information about the XML data type, see "XML data type" [*SQL Anywhere Server - SQL Reference*].

The FOR XML clause can be used in any SELECT statement, including subqueries, queries with a GROUP BY clause or aggregate functions, and view definitions.

For examples of how the FOR XML clause can be used, see "FOR XML examples" on page 676.

SQL Anywhere does not generate a schema for XML documents generated by the FOR XML clause.

Within the FOR XML clause, you specify one of three XML modes that control the format of the XML that is generated:

- **RAW** represents each row that matches the query as an XML <row> element, and each column as an attribute.

  For more information, see "Using FOR XML RAW" on page 676.

- **AUTO** returns 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 of the tables in the *select-list*.

  For more information, see "Using FOR XML AUTO" on page 678.

- **EXPLICIT** allows you to write queries that contain information about the expected nesting so you can control the form of the resulting XML.

For more information, see "Using FOR XML EXPLICIT" on page 681.

The following sections describe the behavior of all three modes of the FOR XML clause regarding binary data, NULL values, and invalid XML names. The section also includes examples of how you can use the FOR XML clause.

FOR XML and binary data

When you use the FOR XML clause in a SELECT statement, regardless of the mode used, any BINARY, LONG BINARY, IMAGE, or VARBINARY columns are output as attributes or elements that are automatically represented in base64-encoded format.

If you are using openxml to generate a result set from XML, openxml assumes that the types BINARY, LONG BINARY, IMAGE, and VARBINARY, are base64-encoded and decodes them automatically.

For more information about openxml, see "openxml system procedure" [*SQL Anywhere Server - SQL Reference*].

FOR XML and NULL values

By default, elements and attributes that contain NULL values are omitted from the result. This behavior is controlled by the for_xml_null_treatment option.

Consider an entry in the Customers table that contains a NULL company name.

```
INSERT INTO
      Customers( ID, Surname, GivenName, Street, City, Phone)
VALUES (100,'Robert','Michael',
       '100 Anywhere Lane','Smallville','519-555-3344');
```

If you execute the following query with the for_xml_null_treatment option set to Omit (the default), then no attribute is generated for a NULL column value.

```
SELECT ID, GivenName, Surname, CompanyName
FROM Customers
WHERE GivenName LIKE 'Michael%'
ORDER BY ID
FOR XML RAW;
```

In this case, no CompanyName attribute is generated for Michael Robert.

```
<row ID="100" GivenName="Michael" Surname="Robert"/>
<row ID="101" GivenName="Michaels" Surname="Devlin" CompanyName="The Power
Group"/>
<row ID="110" GivenName="Michael" Surname="Agliori" CompanyName="The Pep
Squad"/>
```

If the for_xml_null_treatment option is set to Empty, then an empty attribute is included in the result:

```
<row ID="100" GivenName="Michael" Surname="Robert" CompanyName=""/>
<row ID="101" GivenName="Michaels" Surname="Devlin" CompanyName="The Power
```



```
Group"/>
<row ID="110" GivenName="Michael" Surname="Agliori" CompanyName="The Pep
Squad"/>
```

In this case, an empty CompanyName attribute is generated for Michael Robert.

For information about the for_xml_null_treatment option, see "for_xml_null_treatment option" [*SQL Anywhere Server - Database Administration*].

Encoding illegal XML names

SQL Anywhere uses the following rules for encoding names that are not legal XML names (for example, column names that include spaces):

XML has rules for names that differ from rules for SQL names. For example, spaces are not allowed in XML names. When a SQL name, such as a column name, is converted to an XML name, characters that are not valid characters for XML names are encoded or escaped.

For each encoded character, the encoding is based on the character's Unicode code point value, expressed as a hexadecimal number.

- For most characters, the code point value can be represented with 16 bits or four hex digits, using the encoding _xHHHH_. These characters correspond to Unicode characters whose UTF-16 value is one 16-bit word.
- For characters whose code point value requires more than 16 bits, eight hex digits are used in the encoding _xHHHHHHHH_. These characters correspond to Unicode characters whose UTF-16 value is two 16-bit words. However, the Unicode code point value, which is typically 5 or 6 hex digits, is used for the encoding, not the UTF-16 value.

  For example, the following query contains a column name with a space:

  ```
  SELECT EmployeeID AS "Employee ID"
  FROM Employees
  FOR XML RAW;
  ```

  and returns the following result:

  ```
  <row Employee_x0020_ID="102"/>
  <row Employee_x0020_ID="105"/>
  <row Employee_x0020_ID="129"/>
  <row Employee_x0020_ID="148"/>
  ...
  ```

- Underscores (_) are escaped if they are followed by the character x. For example, the name Linu_x is encoded as Linu_x005F_x.
- Colons (:) are not escaped so that namespace declarations and qualified element and attribute names can be generated using a FOR XML query.

For information about the syntax of the FOR XML clause, see "SELECT statement" [*SQL Anywhere Server - SQL Reference*].

Tip
When executing queries that contain a FOR XML clause in Interactive SQL, you may want to increase the column length by setting the truncation_length option.

For information about setting the truncation length, see "truncation_length option [Interactive SQL]" [*SQL Anywhere Server - Database Administration*].

FOR XML examples

The following examples show how the FOR XML clause can be used in a SELECT statement.

- The following example shows how the FOR XML clause can be used in a subquery:

```
SELECT XMLELEMENT(
   NAME root,
      (SELECT * FROM Employees
       FOR XML RAW));
```

- The following example shows how the FOR XML clause can be used in a query with a GROUP BY clause and aggregate function:

```
SELECT Name, AVG(UnitPrice) AS Price
FROM Products
GROUP BY Name
FOR XML RAW;
```

- The following example shows how the FOR XML clause can be used in a view definition:

```
CREATE VIEW EmployeesDepartments
AS SELECT Surname, GivenName, DepartmentName
FROM Employees JOIN Departments
ON Employees.DepartmentID = Departments.DepartmentID
FOR XML AUTO;
```

Using FOR XML RAW

When you specify FOR XML RAW in a query, each row is represented as a <row> element, and each column is an attribute of the <row> element.

Syntax

FOR XML RAW[, **ELEMENTS**]

Parameters

ELEMENTS tells FOR XML RAW to generate an XML element, instead of an attribute, for each column in the result. If there are NULL values, the element is omitted from the generated XML document. The following query generates <EmployeeID> and <DepartmentName> elements:

```
SELECT Employees.EmployeeID, Departments.DepartmentName
FROM Employees JOIN Departments
   ON Employees.DepartmentID=Departments.DepartmentID
FOR XML RAW, ELEMENTS;
```


This query gives the following result:

```
<row>
   <EmployeeID>102</EmployeeID>
   <DepartmentName>R &amp;amp; D</DepartmentName>
</row>
<row>
   <EmployeeID>105</EmployeeID>
   <DepartmentName>R &amp;amp; D</DepartmentName>
</row>
<row>
   <EmployeeID>160</EmployeeID>
   <DepartmentName>R &amp;amp; D</DepartmentName>
</row>
<row>
   <EmployeeID>243</EmployeeID>
   <DepartmentName>R &amp;amp; D</DepartmentName>
</row>
...
```

Usage

Data in BINARY, LONG BINARY, IMAGE, and VARBINARY columns is automatically returned in base64-encoded format when you execute a query that contains FOR XML RAW.

By default, NULL values are omitted from the result. This behavior is controlled by the for_xml_null_treatment option.

For information about how NULL values are returned in queries that contain a FOR XML clause, see "FOR XML and NULL values" on page 674.

FOR XML RAW does not return a well-formed XML document because the document does not have a single root node. If a <root> element is required, one way to insert one is to use the XMLELEMENT function. For example,

```
SELECT XMLELEMENT( NAME root,
                   (SELECT EmployeeID AS id, GivenName AS name
                   FROM Employees FOR XML RAW));
```

For more information about the XMLELEMENT function, see "XMLELEMENT function [String]" [*SQL Anywhere Server - SQL Reference*].

The attribute or element names used in the XML document can be changed by specifying aliases. The following query renames the ID attribute to product_ID:

```
SELECT ID AS product_ID
FROM Products
WHERE Color='black'
FOR XML RAW;
```

This query gives the following result:

```
<row product_ID="302"/>
<row product_ID="400"/>
<row product_ID="501"/>
<row product_ID="700"/>
```

The order of the results depend on the plan chosen by the optimizer, unless you request otherwise. If you want the results to appear in a particular order, you must include an ORDER BY clause in the query, for example:

```
SELECT Employees.EmployeeID, Departments.DepartmentName
FROM Employees JOIN Departments
   ON Employees.DepartmentID=Departments.DepartmentID
ORDER BY EmployeeID
FOR XML RAW;
```

Example

Suppose you want to retrieve information about which department an employee belongs to, as follows:

```
SELECT Employees.EmployeeID, Departments.DepartmentName
FROM Employees JOIN Departments
   ON Employees.DepartmentID=Departments.DepartmentID
FOR XML RAW;
```

The following XML document is returned:

```
<row EmployeeID="102" DepartmentName="R &amp;amp; D"/>
<row EmployeeID="105" DepartmentName="R &amp;amp; D"/>
<row EmployeeID="160" DepartmentName="R &amp;amp; D"/>
<row EmployeeID="243" DepartmentName="R &amp;amp; D"/>
...
```

Using FOR XML AUTO

AUTO mode generates nested elements within the XML document. Each table referenced in the select list is represented as an element in the generated XML. The order of nesting is based on the order in which tables are referenced in the select list. When you specify AUTO mode, an element is created for each table in the select list, and each column in that table is a separate attribute.

Syntax

FOR XML AUTO[, **ELEMENTS**]

Parameters

ELEMENTS tells FOR XML AUTO to generate an XML element, instead of an attribute, for each column in the result. For example,

```
SELECT Employees.EmployeeID, Departments.DepartmentName
FROM Employees JOIN Departments
   ON Employees.DepartmentID=Departments.DepartmentID
ORDER BY EmployeeID
FOR XML AUTO, ELEMENTS;
```

In this case, each column in the result set is returned as a separate element, rather than as an attribute of the <Employees> element. If there are NULL values, the element is omitted from the generated XML document.

```
<Employees>
   <EmployeeID>102</EmployeeID>
   <Departments>
```



```
      <DepartmentName>R &amp; D</DepartmentName>
   </Departments>
</Employees>
<Employees>
   <EmployeeID>105</EmployeeID>
   <Departments>
      <DepartmentName>R &amp; D</DepartmentName>
   </Departments>
</Employees>
<Employees>
   <EmployeeID>129</EmployeeID>
   <Departments>
   <DepartmentName>Sales</DepartmentName>
   </Departments>
</Employees>
...
```

Usage

When you execute a query using FOR XML AUTO, data in BINARY, LONG BINARY, IMAGE, and VARBINARY columns is automatically returned in base64-encoded format. By default, NULL values are omitted from the result. You can return NULL values as empty attributes by setting the for_xml_null_treatment option to EMPTY.

For information about setting the for_xml_null_treatment option, see "for_xml_null_treatment option" [*SQL Anywhere Server - Database Administration*].

Unless otherwise requested, the database server returns the rows of a table in an order that has no meaning. If you want the results to appear in a particular order, or for a parent element to have multiple children, you must include an ORDER BY clause in the query so that all children are adjacent. If you do not specify an ORDER BY clause, the nesting of the results depends on the plan chosen by the optimizer and you may not get the nesting you want.

FOR XML AUTO does not return a well-formed XML document because the document does not have a single root node. If a <root> element is required, one way to insert one is to use the XMLELEMENT function. For example,

```
SELECT XMLELEMENT( NAME root,
                   (SELECT EmployeeID AS id, GivenName AS name
                   FROM Employees FOR XML AUTO ) );
```

For more information about the XMLELEMENT function, see "XMLELEMENT function [String]" [*SQL Anywhere Server - SQL Reference*].

You can change the attribute or element names used in the XML document by specifying aliases. The following query renames the ID attribute to product_ID:

```
SELECT ID AS product_ID
FROM Products
WHERE Color='Black'
FOR XML AUTO;
```

The following XML is generated:

```
<Products product_ID="302"/>
<Products product_ID="400"/>
<Products product_ID="501"/>
<Products product_ID="700"/>
```

You can also rename the table with an alias. The following query renames the table to product_info:

```
SELECT ID AS product_ID
FROM Products AS product_info
WHERE Color='Black'
FOR XML AUTO;
```

The following XML is generated:

```
<product_info product_ID="302"/>
<product_info product_ID="400"/>
<product_info product_ID="501"/>
<product_info product_ID="700"/>
```

Example

The following query generates XML that contains both <employee> and <department> elements, and the <employee> element (the table listed first in the select list) is the parent of the <department> element.

```
SELECT EmployeeID, DepartmentName
FROM Employees AS employee JOIN Departments AS department
   ON Employees.DepartmentID=Departments.DepartmentID
ORDER BY EmployeeID
FOR XML AUTO;
```

The following XML is generated by the above query:

```
<employee EmployeeID="102">
   <department DepartmentName="R &amp; D"/>
</employee>
<employee EmployeeID="105">
   <department DepartmentName="R &amp; D"/>
</employee>
<employee EmployeeID="129">
   <department DepartmentName="Sales;"/>
</employee>
<employee EmployeeID="148">
   <department DepartmentName="Finance;"/>
</employee>
...
```

If you change the order of the columns in the select list as follows:

```
SELECT DepartmentName, EmployeeID
FROM Employees AS employee JOIN Departments AS department
   ON Employees.DepartmentID=Departments.DepartmentID
ORDER BY 1, 2
FOR XML AUTO;
```

The result is nested as follows:

```
<department DepartmentName="Finance">
   <employee EmployeeID="148"/>
   <employee EmployeeID="390"/>
   <employee EmployeeID="586"/>
   ...
</department>
<Department name="Marketing">
   <employee EmployeeID="184"/>
   <employee EmployeeID="207"/>
   <employee EmployeeID="318"/>
```



```
   ...
</department>
...
```

Again, the XML generated for the query contains both <employee> and <department> elements, but in this case the <department> element is the parent of the <employee> element.

Using FOR XML EXPLICIT

FOR XML EXPLICIT allows you to control the structure of the XML document returned by the query. The query must be written in a particular way so that information about the nesting you want is specified within the query result. The optional directives supported by FOR XML EXPLICIT allow you to configure the treatment of individual columns. For example, you can control whether a column appears as element or attribute content, or whether a column is used only to order the result, rather than appearing in the generated XML.

For an example of how to write a query using FOR XML EXPLICIT, see "Writing an EXPLICIT mode query" on page 683.

Parameters

In EXPLICIT mode, the first two columns in the SELECT statement must be named **Tag** and **Parent**, respectively. Tag and Parent are metadata columns, and their values are used to determine the parent-child relationship, or nesting, of the elements in the XML document that is returned by the query.

- **Tag column** This is the first column specified in the select list. The Tag column stores the tag number of the current element. Permitted values for tag numbers are 1 to 255.
- **Parent column** This column stores the tag number for the parent of the current element. If the value in this column is NULL, the row is placed at the top level of the XML hierarchy.

For example, consider a query that returns the following result set when FOR XML EXPLICIT is not specified. (The purpose of the GivenName!1 and ID!2 data columns is discussed in the following section, "Adding data columns to the query" on page 682.)

Tag	Parent	GivenName!1	ID!2
1	NULL	'Beth'	NULL
2	NULL	NULL	'102'

In this example, the values in the Tag column are the tag numbers for each element in the result set. The Parent column for both rows contains the value NULL. This means that both elements are generated at the top level of the hierarchy, giving the following result when the query includes the FOR XML EXPLICIT clause:

```
<GivenName>Beth</GivenName>
<ID>102</ID>
```

However, if the second row had the value 1 in the Parent column, the result would look as follows:

```
<GivenName>Beth
   <ID>102</ID>
</GivenName>
```

For an example of how to write a query using FOR XML EXPLICIT, see "Writing an EXPLICIT mode query" on page 683.

Adding data columns to the query

In addition to the Tag and Parent columns, the query must also contain one or more data columns. The names of these data columns control how the columns are interpreted during tagging. Each column name is split into fields separated by an exclamation mark (!). The following fields can be specified for data columns:

ElementName!*TagNumber*!*AttributeName*!*Directive*

ElementName the name of the element. For a given row, the name of the element generated for the row is taken from the *ElementName* field of the first column with a matching tag number. If there are multiple columns with the same *TagNumber*, the *ElementName* is ignored for subsequent columns with the same *TagNumber*. In the example above, the first row generates an element called <GivenName>.

TagNumber the tag number of the element. For a row with a given tag value, all columns with the same value in their *TagNumber* field will contribute content to the element that corresponds to that row.

AttributeName specifies that the column value is an attribute of the *ElementName* element. For example, if a data column had the name productID!1!Color, then Color would appear as an attribute of the <productID> element.

Directive this optional field allows you to control the format of the XML document further. You can specify any one of the following values for *Directive*:

- **hide** indicates that this column is ignored when generating the result. This directive can be used to include columns that are only used to order the table. The attribute name is ignored and does not appear in the result.

  For an example using the hide directive, see "Using the hide directive" on page 687.

- **element** indicates that the column value is inserted as a nested element with the name *AttributeName*, rather than as an attribute.

  For an example using the element directive, see "Using the element directive" on page 686.

- **xml** indicates that the column value is inserted with no quoting. If the *AttributeName* is specified, the value is inserted as an element with that name. Otherwise, it is inserted with no wrapping element. If this directive is not used, then markup characters are escaped unless the column is of type XML. For example, the value <a/> would be inserted as <a/>.

  For an example using the xml directive, see "Using the xml directive" on page 688.

- **cdata** indicates that the column value is to be inserted as a CDATA section. The *AttributeName* is ignored.

For an example using the cdata directive, see "Using the cdata directive" on page 689.

Usage

Data in BINARY, LONG BINARY, IMAGE, and VARBINARY columns is automatically returned in base64-encoded format when you execute a query that contains FOR XML EXPLICIT. By default, any NULL values in the result set are omitted. You can change this behavior by changing the setting of the for_xml_null_treatment option.

For more information about the for_xml_null_treatment option, see "for_xml_null_treatment option" [*SQL Anywhere Server - Database Administration*] and "FOR XML and NULL values" on page 674.

Writing an EXPLICIT mode query

Suppose you want to write a query using FOR XML EXPLICIT that generates the following XML document:

```
<employee EmployeeID='129'>
   <customer CustomerID='107' Region='Eastern'/>
   <customer CustomerID='119' Region='Western'/>
   <customer CustomerID='131' Region='Eastern'/>
</employee>
<employee EmployeeID='195'>
   <customer CustomerID='109' Region='Eastern'/>
   <customer CustomerID='121' Region='Central'/>
</employee>
```

You do this by writing a SELECT statement that returns the following result set in the exact order specified, and then appending FOR XML EXPLICIT to the query.

Tag	Parent	employee!1!EmployeeID	customer!2!CustomerID	customer!2!Region
1	NULL	129	NULL	NULL
2	1	129	107	Eastern
2	1	129	119	Western
2	1	129	131	Central
1	NULL	195	NULL	NULL
2	1	195	109	Eastern
2	1	195	121	Central

When you write your query, only some of the columns for a given row become part of the generated XML document. A column is included in the XML document only if the value in the *TagNumber* field (the second field in the column name) matches the value in the Tag column.

In the example, the third column is used for the two rows that have the value 1 in their Tag column. In the fourth and fifth columns, the values are used for the rows that have the value 2 in their Tag column. The element names are taken from the first field in the column name. In this case, <employee> and <customer> elements are created.

The attribute names come from the third field in the column name, so an EmployeeID attribute is created for <employee> elements, while CustomerID and Region attributes are generated for <customer> elements.

The following steps explain how to construct the FOR XML EXPLICIT query that generates an XML document similar to the one found at the beginning of this section using the SQL Anywhere sample database.

To write a FOR XML EXPLICIT query

1. Write a SELECT statement to generate the top-level elements.

   In this example, the first SELECT statement in the query generates the <employee> elements. The first two values in the query must be the Tag and Parent column values. The <employee> element is at the top of the hierarchy, so it is assigned a Tag value of 1, and a Parent value of NULL.

   Note
   If you are writing an EXPLICIT mode query that uses a UNION, then only the column names specified in the first SELECT statement are used. Column names that are to be used as element or attribute names must be specified in the first SELECT statement because column names specified in subsequent SELECT statements are ignored.

   To generate the <employee> elements for the table above, your first SELECT statement is as follows:

   ```
   SELECT
         1          AS tag,
         NULL       AS parent,
         EmployeeID AS [employee!1!EmployeeID],
         NULL       AS [customer!2!CustomerID],
         NULL       AS [customer!2!Region]
   FROM Employees;
   ```

2. Write a SELECT statement to generate the child elements.

   The second query generates the <customer> elements. Because this is an EXPLICIT mode query, the first two values specified in all the SELECT statements must be the Tag and Parent values. The <customer> element is given the tag number 2, and because it is a child of the <employee> element, it has a Parent value of 1. The first SELECT statement has already specified that EmployeeID, CustomerID, and Region are attributes.

   ```
   SELECT
         2,
         1,
         EmployeeID,
         CustomerID,
         Region
   FROM Employees KEY JOIN SalesOrders
   ```

3. Add a UNION ALL to the query to combine the two SELECT statements together:

   ```
   SELECT
         1          AS tag,
         NULL       AS parent,
         EmployeeID AS [employee!1!EmployeeID],
         NULL       AS [customer!2!CustomerID],
         NULL       AS [customer!2!Region]
   FROM Employees
   ```



```
UNION ALL
SELECT
      2,
      1,
      EmployeeID,
      CustomerID,
      Region
FROM Employees KEY JOIN SalesOrders
```

4. Add an ORDER BY clause to specify the order of the rows in the result. The order of the rows is the order that is used in the resulting document.

```
SELECT
      1          AS tag,
      NULL       AS parent,
      EmployeeID AS [employee!1!EmployeeID],
      NULL       AS [customer!2!CustomerID],
      NULL       AS [customer!2!Region]
FROM Employees
UNION ALL
SELECT
      2,
      1,
      EmployeeID,
      CustomerID,
      Region
FROM Employees KEY JOIN SalesOrders
ORDER BY 3, 1
FOR XML EXPLICIT;
```

For information about the syntax of EXPLICIT mode, see "Parameters" on page 681.

FOR XML EXPLICIT examples

The following example query retrieves information about the orders placed by employees. In this example, there are three types of elements: <employee>, <order>, and <department>. The <employee> element has ID and name attributes, the <order> element has a date attribute, and the <department> element has a name attribute.

```
SELECT
       1           tag,
       NULL        parent,
       EmployeeID  [employee!1!ID],
       GivenName   [employee!1!name],
       NULL        [order!2!date],
       NULL        [department!3!name]
FROM Employees
UNION ALL
SELECT
       2,
       1,
       EmployeeID,
       NULL,
       OrderDate,
       NULL
FROM Employees KEY JOIN SalesOrders
UNION ALL
SELECT
       3,
       1,
       EmployeeID,
```

```
        NULL,
        NULL,
        DepartmentName
FROM Employees e JOIN Departments d
   ON e.DepartmentID=d.DepartmentID
ORDER BY 3, 1
FOR XML EXPLICIT;
```

You get the following result from this query:

```
<employee ID="102" name="Fran">
   <department name="R &amp; D"/>
</employee>
<employee ID="105" name="Matthew">
   <department name="R &amp; D"/>
</employee>
<employee ID="129" name="Philip">
   <order date="2000-07-24"/>
   <order date="2000-07-13"/>
   <order date="2000-06-24"/>
   <order date="2000-06-08"/>
   ...
   <department name="Sales"/>
</employee>
<employee ID="148" name="Julie">
   <department name="Finance"/>
</employee>
...
```

Using the element directive

If you want to generate sub-elements rather than attributes, you can add the element directive to the query, as follows:

```
SELECT
        1           tag,
        NULL        parent,
        EmployeeID  [employee!1!id!element],
        GivenName   [employee!1!name!element],
        NULL        [order!2!date!element],
        NULL        [department!3!name!element]
FROM Employees
UNION ALL
SELECT
        2,
        1,
        EmployeeID,
        NULL,
        OrderDate,
        NULL
FROM Employees KEY JOIN SalesOrders
UNION ALL
SELECT
        3,
        1,
        EmployeeID,
        NULL,
        NULL,
        DepartmentName
FROM Employees e JOIN Departments d
   ON e.DepartmentID=d.DepartmentID
ORDER BY 3, 1
FOR XML EXPLICIT;
```


You get the following result from this query:

```
<employee>
   <id>102</id>
   <name>Fran</name>
   <department>
      <name>R &amp;amp; D</name>
   </department>
</employee>
<employee>
   <id>105</id>
   <name>Matthew</name>
   <department>
      <name>R &amp;amp; D</name>
   </department>
</employee>
<employee>
   <id>129</id>
   <name>Philip</name>
   <order>
      <date>2000-07-24</date>
   </order>
   <order>
      <date>2000-07-13</date>
   </order>
   <order>
      <date>2000-06-24</date>
   </order>
   ...
   <department>
      <name>Sales</name>
   </department>
</employee>
...
```

Using the hide directive

In the following query, the employee ID is used to order the result, but the employee ID does not appear in the result because the hide directive is specified:

```
SELECT
        1           tag,
        NULL        parent,
        EmployeeID  [employee!1!id!hide],
        GivenName   [employee!1!name],
        NULL        [order!2!date],
        NULL        [department!3!name]
FROM Employees
UNION ALL
SELECT
        2,
        1,
        EmployeeID,
        NULL,
        OrderDate,
        NULL
FROM Employees KEY JOIN SalesOrders
UNION ALL
SELECT
        3,
        1,
        EmployeeID,
        NULL,
```

```
        NULL,
        DepartmentName
FROM Employees e JOIN Departments d
   ON e.DepartmentID=d.DepartmentID
ORDER BY 3, 1
FOR XML EXPLICIT;
```

This query returns the following result:

```
<employee name="Fran">
   <department name="R &amp; D"/>
</employee>
<employee name="Matthew">
   <department name="R &amp; D"/>
</employee>
<employee name="Philip">
   <order date="2000-04-21"/>
   <order date="2001-07-23"/>
   <order date="2000-12-30"/>
   <order date="2000-12-20"/>
   ...
   <department name="Sales"/>
</employee>
<employee name="Julie">
   <department name="Finance"/>
</employee>
...
```

Using the xml directive

By default, when the result of a FOR XML EXPLICIT query contains characters that are not valid XML characters, the invalid characters are escaped unless the column is of type XML. For information, see "Encoding illegal XML names" on page 675.

For example, the following query generates XML that contains an ampersand (&):

```
SELECT
        1               AS tag,
        NULL            AS parent,
        ID              AS [customer!1!ID!element],
        CompanyName     AS [customer!1!CompanyName]
FROM Customers
WHERE ID = '115'
FOR XML EXPLICIT;
```

In the result generated by this query, the ampersand is escaped because the column is not of type XML:

```
<Customers CompanyName="Sterling &amp; Co.">
   <ID>115</ID>
</Customers>
```

The xml directive indicates that the column value is inserted into the generated XML with no quoting. If you execute the same query as above with the xml directive:

```
SELECT
        1               AS tag,
        NULL            AS parent,
        ID              AS [customer!1!ID!element],
        CompanyName     AS [customer!1!CompanyName!xml]
FROM Customers
```



```
WHERE ID = '115'
FOR XML EXPLICIT;
```

The ampersand is not quoted in the result:

```
<customer>
 <ID>115</ID>
 <CompanyName>Sterling & Co.</CompanyName>
</customer>
```

Note that this XML is not well-formed because it contains an ampersand, which is a special character in XML. When XML is generated by a query, it is your responsibility to ensure that the XML is well-formed and valid: SQL Anywhere does not check whether the XML being generated is well-formed or valid.

When you specify the xml directive, the *AttributeName* field is ignored, and elements are generated rather than attributes.

Using the cdata directive

The following query uses the cdata directive to return the customer name in a CDATA section:

```
SELECT
        1               AS tag,
        NULL            AS parent,
        ID              AS [product!1!ID],
        Description     AS [product!1!!cdata]
FROM Products
FOR XML EXPLICIT;
```

The result produced by this query lists the description for each product in a CDATA section. Data contained in the CDATA section is not quoted:

```
<product ID="300">
   <![CDATA[Tank Top]]>
</product>
<product ID="301">
   <![CDATA[V-neck]]>
</product>
<product ID="302">
   <![CDATA[Crew Neck]]>
</product>
<product ID="400">
   <![CDATA[Cotton Cap]]>
</product>
...
```

Using SQL/XML to obtain query results as XML

SQL/XML is a draft standard that describes a functional integration of XML into the SQL language: it describes the ways that SQL can be used in conjunction with XML. The supported functions allow you to write queries that construct XML documents from relational data.

Invalid names and SQL/XML

In SQL/XML, expressions that are not legal XML names, for example expressions that include spaces, are escaped in the same manner as the FOR XML clause. Element content of type XML is not quoted.

For more information about quoting invalid expressions, see "Encoding illegal XML names" on page 675.

For information about using the XML data type, see "Storing XML documents in relational databases" on page 663.

Using the XMLAGG function

The XMLAGG function is used to produce a forest of XML elements from a collection of XML elements. XMLAGG is an aggregate function, and produces a single aggregated XML result for all the rows in the query.

In the following query, XMLAGG is used to generate a <name> element for each row, and the <name> elements are ordered by employee name. The ORDER BY clause is specified to order the XML elements:

```
SELECT XMLELEMENT( NAME Departments,
                   XMLATTRIBUTES ( DepartmentID ),
                   XMLAGG( XMLELEMENT( NAME name,
                                       Surname )
                                       ORDER BY Surname )
                 ) AS department_list
FROM Employees
GROUP BY DepartmentID
ORDER BY DepartmentID;
```

This query produces the following result:

department_list
<Departments DepartmentID="100"> <name>Breault</name> <name>Cobb</name> <name>Diaz</name> <name>Driscoll</name> ... </Departments>
<Departments DepartmentID="200"> <name>Chao</name> <name>Chin</name> <name>Clark</name> <name>Dill</name> ... </Departments>
<Departments DepartmentID="300"> <name>Bigelow</name> <name>Coe</name> <name>Coleman</name> <name>Davidson</name> ... </Departments>
...

For more information about the XMLAGG function, see "XMLAGG function [Aggregate]" [*SQL Anywhere Server - SQL Reference*].

Using the XMLCONCAT function

The XMLCONCAT function creates a forest of XML elements by concatenating all the XML values passed in. For example, the following query concatenates the <given_name> and <surname> elements for each employee in the Employees table:

```
SELECT XMLCONCAT( XMLELEMENT( NAME given_name, GivenName ),
                  XMLELEMENT( NAME surname, Surname )
                ) AS "Employee_Name"
FROM Employees;
```

This query returns the following result:

Employee_Name
<given_name>Fran</given_name> <surname>Whitney</surname>
<given_name>Matthew</given_name> <surname>Cobb</surname>
<given_name>Philip</given_name> <surname>Chin</surname>
<given_name>Julie</given_name> <surname>Jordan</surname>
...

For more information, see "XMLCONCAT function [String]" [*SQL Anywhere Server - SQL Reference*].

Using the XMLELEMENT function

The XMLELEMENT function constructs an XML element from relational data. You can specify the content of the generated element and if you want, you can also specify attributes and attribute content for the element.

Generating nested elements

The following query generates nested XML, producing a <product_info> element for each product, with elements that provide the name, quantity, and description of each product:

```
SELECT ID,
XMLELEMENT( NAME product_info,
            XMLELEMENT( NAME item_name, Products.name ),
            XMLELEMENT( NAME quantity_left, Products.Quantity ),
            XMLELEMENT( NAME description, Products.Size || ' ' ||
                        Products.Color || ' ' || Products.name )
```

```
          ) AS results
FROM Products
WHERE Quantity > 30;
```

This query produces the following result:

ID	results
301	<product_info> <item_name>Tee Shirt </item_name> <quantity_left>54 </quantity_left> <description>Medium Orange Tee Shirt</description> </product_info>
302	<product_info> <item_name>Tee Shirt </item_name> <quantity_left>75 </quantity_left> <description>One Size fits all Black Tee Shirt </description> </product_info>
400	<product_info> <item_name>Baseball Cap </item_name> <quantity_left>112 </quantity_left> <description>One Size fits all Black Baseball Cap </description> </product_info>
...	...

Specifying element content

The XMLELEMENT function allows you to specify the content of an element. The following statement produces an XML element with the content **hat**.

```
SELECT ID, XMLELEMENT( NAME product_type, 'hat' )
FROM Products
WHERE Name IN ( 'Baseball Cap', 'Visor' );
```

Generating elements with attributes

You can add attributes to the elements by including the XMLATTRIBUTES argument in your query. This argument specifies the attribute name and content. The following statement produces an attribute for the name, Color, and UnitPrice of each item.

```
SELECT ID, XMLELEMENT( NAME item_description,
                       XMLATTRIBUTES( Name,
                                      Color,
                                      UnitPrice )
                     ) AS item_description_element
```



```
FROM Products
WHERE ID > 400;
```

Attributes can be named by specifying the AS clause:

```
SELECT ID, XMLELEMENT( NAME item_description,
                       XMLATTRIBUTES ( UnitPrice AS
                                       price ),
                       Products.name
                     ) AS products
FROM Products
WHERE ID > 400;
```

For more information, see "XMLELEMENT function [String]" [*SQL Anywhere Server - SQL Reference*].

Example

The following example uses XMLELEMENT with an HTTP web service.

```
ALTER PROCEDURE "DBA"."http_header_example_with_table_proc"()
RESULT ( res LONG VARCHAR )
BEGIN
   DECLARE var LONG VARCHAR;
   DECLARE varval LONG VARCHAR;
   DECLARE I INT;
 DECLARE res LONG VARCHAR;
 DECLARE tabl XML;
   SET var  = NULL;
loop_h:
   LOOP
      SET var = NEXT_HTTP_HEADER( var );
      IF var IS NULL THEN LEAVE leave loop_h END IF;
      SET varval = http_header( var );
      -- ... do some action for <var,varval> pair...
        SET tabl = tabl ||
                        XMLELEMENT( name "tr",
                        XMLATTRIBUTES( 'left' AS "align", 'top' AS
"valign" ),
                        XMLELEMENT( name "td", var ),
                        XMLELEMENT( name "td", varval ) )  ;

   END LOOP;

    SET res = XMLELEMENT( NAME "table",
                  XMLATTRIBUTES( '' AS "BORDER", '10' as "CELLPADDING", '0' AS
"CELLSPACING" ),

                  XMLELEMENT( NAME "th",
                      XMLATTRIBUTES( 'left' AS "align", 'top' AS "valign" ),
                      'Header Name' ),

                  XMLELEMENT( NAME "th",
                      XMLATTRIBUTES( 'left' AS "align", 'top' AS "valign" ),
                      'Header Value' ),

                   tabl ) ;
    SELECT res;
END
```

Using the XMLFOREST function

XMLFOREST constructs a forest of XML elements. An element is produced for each XMLFOREST argument.

The following query produces an <item_description> element, with <name>, <color>, and <price> elements:

```
SELECT ID, XMLELEMENT( NAME item_description,
                       XMLFOREST( Name as name,
                                  Color as color,
                                  UnitPrice AS price )
                     ) AS product_info
FROM Products
WHERE ID > 400;
```

The following result is generated by this query:

ID	product_info
401	<item_description> <name>Baseball Cap</name> <color>White</color> <price>10.00</price> </item_description>
500	<item_description> <name>Visor</name> <color>White</color> <price>7.00</price> </item_description>
501	<item_description> <name>Visor</name> <color>Black</color> <price>7.00</price> </item_description>
...	...

For more information, see "XMLFOREST function [String]" [*SQL Anywhere Server - SQL Reference*].

Using the XMLGEN function

The XMLGEN function is used to generate an XML value based on an XQuery constructor.

The XML generated by the following query provides information about customer orders in the SQL Anywhere sample database. It uses the following variable references:

- **{$ID}** Generates content for the <ID> element using values from the ID column in the SalesOrders table.

- **{$OrderDate}** Generates content for the <date> element using values from the OrderDate column in the SalesOrders table.
- **{$Customers}** Generates content for the <customer> element from the CompanyName column in the Customers table.

```
SELECT XMLGEN ( '<order>
                <ID>{$ID}</ID>
                <date>{$OrderDate}</date>
                <customer>{$Customers}</customer>
                </order>',
                SalesOrders.ID,
                SalesOrders.OrderDate,
                Customers.CompanyName AS Customers
                ) AS order_info
FROM SalesOrders JOIN Customers
ON Customers.ID = SalesOrders.CustomerID
ORDER BY SalesOrders.CustomerID;
```

This query generates the following result:

order_info
<order> <ID>2001</ID> <date>2000-03-16</date> <customer>The Power Group</customer> </order>
<order> <ID>2005</ID> <date>2001-03-26</date> <customer>The Power Group</customer> </order>
<order> <ID>2125</ID> <date>2001-06-24</date> <customer>The Power Group</customer> </order>
<order> <ID>2206</ID> <date>2000-04-16</date> <customer>The Power Group</customer> </order>
...

Generating attributes

If you want the order ID number to appear as an attribute of the <order> element, you would write query as follows (note that the variable reference is contained in double quotes because it specifies an attribute value):

```
SELECT XMLGEN ( '<order ID="{$ID}">
                  <date>{$OrderDate}</date>
                  <customer>{$Customers}</customer>
                  </order>',
```

```
                SalesOrders.ID,
                SalesOrders.OrderDate,
                Customers.CompanyName AS Customers
              ) AS order_info
FROM SalesOrders JOIN Customers
ON Customers.ID = SalesOrders.CustomerID
ORDER BY SalesOrders.OrderDate;
```

This query generates the following result:

order_info
<order ID="2131"> <date>2000-01-02</date> <customer>BoSox Club</customer> </order>
<order ID="2065"> <date>2000-01-03</date> <customer>Bloomfield's</customer> </order>
<order ID="2126"> <date>2000-01-03</date> <customer>Leisure Time</customer> </order>
<order ID="2127"> <date>2000-01-06</date> <customer>Creative Customs Inc.</customer> </order>
...

In both result sets, the customer name Bloomfield's is quoted as Bloomfield's because the apostrophe is a special character in XML and the column the <customer> element was generated from was not of type XML.

For more information about quoting of illegal characters in XMLGEN, see “Invalid names and SQL/XML” on page 689.

Specifying header information for XML documents

The FOR XML clause and the SQL/XML functions supported by SQL Anywhere do not include version declaration information in the XML documents they generate. You can use the XMLGEN function to generate header information.

```
SELECT XMLGEN( '<?xml version="1.0"
                encoding="ISO-8859-1" ?>
                <r>{$x}</r>',
                (SELECT GivenName, Surname
      FROM Customers FOR XML RAW) AS x );
```

This produces the following result:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<r>
 <row GivenName="Michaels" Surname="Devlin"/>
```



```
  <row GivenName="Beth" Surname="Reiser"/>
  <row GivenName="Erin" Surname="Niedringhaus"/>
  <row GivenName="Meghan" Surname="Mason"/>
  ...
</r>
```

For more information about the XMLGEN function, see "XMLGEN function [String]" [*SQL Anywhere Server - SQL Reference*].

Remote data and bulk operations

This section describes how to load and unload your database, and how to access remote data.

Importing and exporting data

The term **bulk operations** is used to describe the process of importing and exporting data. Bulk operations must be executed by a user with DBA authority and are not part of typical end-user applications. Bulk operations may affect concurrency and transaction logs and should be performed when users are not connected to the database.

The following are typical situations in which data is imported or exported:

- Importing an initial set of data into a new database
- Building new copies of a database, perhaps with a modified structure
- Exporting data from your database for use with other applications, such as spreadsheets
- Creating extractions of a database for replication or synchronization
- Repairing a corrupt database
- Rebuilding a database to improve its performance
- Obtaining a newer version of database software and completing software upgrades

Performance aspects of bulk operations

The performance of bulk operations depends on several factors, including whether the operation is internal or external to the database server.

Internal bulk operations

Internal bulk operations, also referred to as *server-side* bulk operations, are import and export operations performed by the database server using the LOAD TABLE, and UNLOAD statements.

When performing internal bulk operations, you can load from, and unload to, ASCII text files, or Adaptive Server Enterprise BCP files. These files can exist on the same computer as the database server, or on a client computer. The specified path to the file being written or read is relative to the database server. Internal bulk operations are the fastest method of importing and exporting data into the database.

External bulk operations

External bulk operations, also referred to as *client-side* bulk operations, are import and export operations performed by a client such as Interactive SQL, using INPUT and OUTPUT statements. When the client

issues an INPUT statement, an INSERT statement is recorded in the transaction log for each row that is read when processing the file specified in the INPUT statement. As a result, client-side loading is considerably slower than server-side loading. As well, INSERT triggers fire during an INPUT.

The OUTPUT statement allows you to write the result set of a SELECT statement to many different file formats.

For external bulk operations, the specified path to the file being read or written is relative to the computer on which the client application is running.

See also

- "LOAD TABLE statement" [*SQL Anywhere Server - SQL Reference*]
- "UNLOAD statement" [*SQL Anywhere Server - SQL Reference*]
- "INPUT statement [Interactive SQL]" [*SQL Anywhere Server - SQL Reference*]
- "OUTPUT statement [Interactive SQL]" [*SQL Anywhere Server - SQL Reference*]
- "Performance tips for importing data" on page 701
- "-b dbeng12/dbsrv12 server option" [*SQL Anywhere Server - Database Administration*]

Data recovery issues for bulk operations

You can run the database server in bulk operations mode (the -b server option). When you use this option, the database server does not perform certain important functions. Specifically:

Function	Implication
Maintain a transaction log	There is no record of the changes. Each COMMIT causes a checkpoint.
Lock any records	There are no serious implications.

Alternatively, you may also need to ensure that data from bulk loading is still available in the event of recovery. You can do so by keeping the original data sources intact, and in their original location. You can also use some of the logging options available for the LOAD TABLE statement that allow bulk-loaded data to be recorded in the transaction log. See "LOAD TABLE statement" [*SQL Anywhere Server - SQL Reference*].

Caution
You should back up the database before and after using bulk operations mode because your database is not protected against media failure in this mode.

See also

- "-b dbeng12/dbsrv12 server option" [*SQL Anywhere Server - Database Administration*]

Importing data

Importing data is an administrative task that involves reading data into your database as a bulk operation. Use SQL Anywhere to:

- import entire tables or portions of tables from text files
- import data from a variable
- import several tables consecutively by automating the import procedure with a script
- insert or add data into tables
- replace data in tables
- create a table before the import or during the import
- load data from a file on a client computer
- transfer files between SQL Anywhere and Adaptive Server Enterprise using the BCP format clause

If you are trying to create an entirely new database, consider loading the data using LOAD TABLE for the best performance.

For more information about unloading and reloading complete databases, see "Rebuilding databases" on page 731.

See also

- "LOAD TABLE statement" [*SQL Anywhere Server - SQL Reference*]
- "UNLOAD statement" [*SQL Anywhere Server - SQL Reference*]
- "INPUT statement [Interactive SQL]" [*SQL Anywhere Server - SQL Reference*]
- "OUTPUT statement [Interactive SQL]" [*SQL Anywhere Server - SQL Reference*]
- "Performance tips for importing data" on page 701
- "Performance aspects of bulk operations" on page 699
- "-b dbeng12/dbsrv12 server option" [*SQL Anywhere Server - Database Administration*]
- "Table structures for import" on page 716
- "Accessing data on client computers" on page 728

Performance tips for importing data

Importing large volumes of data can be time consuming. To save time you can:

- Place data files on a separate physical disk drive from the database. This could avoid excessive disk head movement during the load.
- Extend the size of the database. This command allows a database to be extended in large amounts before the space is required, rather than in smaller amounts when the space is needed. It also improves performance when loading large amounts of data, and keeps the database more contiguous within the file system. See "ALTER DBSPACE statement" [*SQL Anywhere Server - SQL Reference*].

- Use temporary tables to load data. Local or global temporary tables are useful when you need to load a set of data repeatedly, or when you need to merge tables with different structures.
- Start the database server without the -b option (bulk operations mode) when using the LOAD TABLE statement.
- Run Interactive SQL or the client application on the same computer as the database server if you are using the INPUT or OUTPUT statement. Loading data over the network adds extra communication overhead. You may want to load new data at a time when the database server is not busy.

See also

- "LOAD TABLE statement" [*SQL Anywhere Server - SQL Reference*]
- "INPUT statement [Interactive SQL]" [*SQL Anywhere Server - SQL Reference*]
- "OUTPUT statement [Interactive SQL]" [*SQL Anywhere Server - SQL Reference*]
- "-b dbeng12/dbsrv12 server option" [*SQL Anywhere Server - Database Administration*]

Import data with the Import Wizard

Use the Interactive SQL **Import Wizard** to select a source, format, and destination table for the data. You can import data from TEXT and FIXED format files. You can import data into an existing table or a new table. You can also use the **Import Wizard** to import data between:

- databases of different types, such as between a SQL Anywhere database and an UltraLite database.
- databases of different versions (as long as you have an ODBC driver for each database), such as between a SQL Anywhere 12 database and a SQL Anywhere 11 database.

Use the Interactive SQL **Import Wizard** when you:

- want to create a table at the same time you import the data
- prefer using a point-and click interface to import data in a format other than text

To import data

1. In Interactive SQL, choose **Data** » **Import**.
2. Follow the instructions in the **Import Wizard**.

To import data from a file into the SQL Anywhere sample database

1. Create and save a text file named *import.txt* with the following values (on a single line):

```
100,500,'Chan','Julia',100,'300 Royal Drive',
'Springfield','OR','USA','97015','6175553985',
'A','017239033',55700,'1984-09-29',,'1968-05-05',
1,1,0,'F'
```

2. In Interactive SQL, choose **Data** » **Import**.

3. Select **In A Text File** and click **Next**.
4. In the **File Name** field, type **import.txt**.
5. Select **In An Existing Table**.
6. Select **Employees** and click **Next**.
7. In the **Field Separator** list, select **Comma(,)**. Click **Next**.
8. Click **Import**.
9. Click **Close**.

   The SQL statement created by the wizard is stored in the command history when the import finishes.

   You can view the generated SQL statement; from the **SQL** menu, choose **Previous SQL**.

   The IMPORT statement generated by the **Import Wizard** appears in the **SQL Statements** pane:

```
--  Generated by the Import Wizard
INPUT INTO "GROUPO"."Employees" from 'C:\\Tobedeleted\\import.txt'
FORMAT TEXT ESCAPES ON ESCAPE CHARACTER '\\' DELIMITED BY ',' ENCODING
'Cp1252'
```

To import data from the SQL Anywhere sample database into an UltraLite database

1. Connect to an UltraLite database, such as, *C:\Documents and Settings\All Users\Documents\SQL Anywhere 12\Samples\UltraLite\CustDB\custdb.udb*.
2. In Interactive SQL, choose **Data** » **Import**.
3. Click **In A Database**. Click **Next**.
4. In the **Database Type** list, choose **SQL Anywhere**.
5. From the **Action** dropdown list, choose **Connect With An ODBC Data Source**.
6. Click **ODBC Data Source Name**, and then type in the box below, **SQL Anywhere 12 Demo**.
7. Click **Next**.
8. In the **Table Name** list, select **Customers**. Click **Next**.
9. Click **In A New Table**.
10. In the **Owner** field, type **dba**.
11. In the **Table Name** field, type **SQLAnyCustomers**.
12. Click **Import**.

13. Click **Close**.

14. To view the generated SQL statement, choose **SQL** » **Previous SQL**.

    The IMPORT statement created and used by the **Import Wizard** appears in the **SQL Statements** pane.

```
--  Generated by the Import Wizard
INPUT USING 'DSN=SQL Anywhere 12 Demo;CON='''''
FROM "GROUPO.Customers" INTO "dba"."SQLAnyCustomers"
CREATE TABLE ON
```

Import data with the INPUT statement

Use the INPUT statement to import data in different file formats into existing or new tables. If you have the ODBC drivers for the databases, use the USING clause to import data from different types of databases, and from different versions of SQL Anywhere databases.

With the INPUT statement, you can import data from TEXT and FIXED formats. To import data from another file format, use the USING clause with an ODBC data source.

You can use the default input format, or you can specify the file format for each INPUT statement. Because the INPUT statement is an Interactive SQL command, you cannot use it in any compound statement (such as an IF statement) or in a stored procedure.

Use the INPUT statement to import data when you want to import data from a file, or from another database.

For more information, see "INPUT statement [Interactive SQL]" [*SQL Anywhere Server - SQL Reference*].

Considerations for materialized views

For immediate views, an error is returned when you attempt to bulk load data into an underlying table. You must truncate the data in the view first, and then perform the bulk load operation.

For manual views, you can bulk load data into an underlying table. However, the data in the view remains stale until the next refresh.

Consider truncating data in dependent materialized views before attempting a bulk load operation such as INPUT on a table. After you have loaded the data, refresh the view. See "TRUNCATE statement" [*SQL Anywhere Server - SQL Reference*], and "REFRESH MATERIALIZED VIEW statement" [*SQL Anywhere Server - SQL Reference*].

Considerations for text indexes

For immediate text indexes, updating the text index after performing a bulk load operation such as INPUT on the underlying table can take a while even though the update is automatic. For manual text indexes, even a refresh can take a while.

Consider dropping dependent text indexes before performing a bulk load operation such as INPUT on a table. After you have loaded the data, recreate the text index. See "DROP TEXT INDEX statement" [*SQL Anywhere Server - SQL Reference*], and "CREATE TEXT INDEX statement" [*SQL Anywhere Server - SQL Reference*].

Impact on the database

Changes are recorded in the transaction log when you use the INPUT statement. In the event of a media failure, there is a detailed record of the changes. However, there are performance impacts associated with importing large amounts of data with this method since all rows are written to the transaction log.

In comparison, the LOAD TABLE statement does not save each row to the transaction log and so it can be faster than the INPUT statement. However, the INPUT statement supports more databases and file formats.

To import data (INPUT statement)

1. Create and save a text file named *new_employees.txt* with the following values (on a single line):

```
101,500,'Chan','Julia',100,'300 Royal Drive',
'Springfield','OR','USA','97015','6175553985',
'A','017239033',55700,'1984-09-29',,'1968-05-05',
1,1,0,'F'
```

2. Open Interactive SQL and connect to the SQL Anywhere 12 sample database.

3. Enter an INPUT statement in the **SQL Statements** pane.

```
INPUT INTO Employees
FROM c:\new_employees.txt
FORMAT TEXT;
SELECT * FROM Employees;
```

   In this statement, the name of the destination table in the SQL Anywhere 12 sample database is Employees, and *new_employees.txt* is the name of the source file.

4. Execute the statement.

   If the import is successful, the **Messages** tab displays the amount of time it to took to import the data. If the import is unsuccessful, a message appears indicating why the import was unsuccessful.

To input data from an Excel CSV file using the INPUT statement

1. In Excel, save the data from your Excel file into a comma delimited (CSV) file. For example name the file *c:\test\finance_comma_delimited.csv*

2. In Interactive SQL, connect to a SQL Anywhere database such as the demo12 database.

3. Create a table named **imported_sales** and add the required columns. You cannot use the CREATE TABLE clause to create a table when inputting from a csv file.

4. Execute an INPUT statement using the SKIP clause to skip over the column names that Excel places in the first line in the CSV file.

```
INPUT INTO "imported_sales" FROM 'c:\\test\\finances.csv' SKIP 1
```

Import data with the LOAD TABLE statement

Use the LOAD TABLE statement to import data residing on a database server or a client computer into an existing table in text/ASCII format.

You can also use the LOAD TABLE statement to import data from a column from another table, or from a value expression (for example, from the results of a function or system procedure).

The LOAD TABLE statement adds rows into a table; it doesn't replace them.

Loading data using the LOAD TABLE statement (without the WITH ROW LOGGING and WITH CONTENT LOGGING options) is considerably faster than using the INPUT statement.

Triggers do not fire for data loaded using the LOAD TABLE statement.

Considerations for materialized views

For immediate views, an error is returned when you attempt to bulk load data into an underlying table. You must truncate the data in the view first, and then perform the bulk load operation.

For manual views, you can bulk load data into an underlying table; however, the data in the view becomes stale until the next refresh.

Consider truncating data in dependent materialized views before attempting a bulk load operation such as LOAD TABLE on a table. After you have loaded the data, refresh the view. See "TRUNCATE statement" [*SQL Anywhere Server - SQL Reference*], and "REFRESH MATERIALIZED VIEW statement" [*SQL Anywhere Server - SQL Reference*].

Considerations for text indexes

For immediate text indexes, updating the text index after performing a bulk load operation such as LOAD TABLE on the underlying table can take a while even though the update is automatic. For manual text indexes, even a refresh can take a while.

Consider dropping dependent text indexes before performing a bulk load operation such as LOAD TABLE on a table. After you have loaded the data, recreate the text index. See "DROP TEXT INDEX statement" [*SQL Anywhere Server - SQL Reference*], and "CREATE TEXT INDEX statement" [*SQL Anywhere Server - SQL Reference*].

Considerations for database recovery and synchronization

By default, when data is loaded from a file (for example, `LOAD TABLE` *`table-name`* `FROM` *`filename`*`;`), only the LOAD TABLE statement is recorded in the transaction log, not the actual rows of data that are being loaded. This presents a problem when trying to recover the database using the transaction log if the original load file has been changed, moved, or deleted. It also means that databases involved in synchronization or replication do not get the new data.

To address the recovery and synchronization considerations, two logging options are available for the LOAD TABLE statement: WITH ROW LOGGING, which creates INSERT statements in the transaction log for every row that is loaded, and WITH CONTENT LOGGING, which groups the loaded rows into chunks and records the chunks in the transaction log. These options allow a load operation to be repeated,

even when the source of the loaded data is no longer available. See "LOAD TABLE statement" [*SQL Anywhere Server - SQL Reference*].

Considerations for database mirroring

If your database is involved in mirroring, use the LOAD TABLE statement carefully. For example, if you are loading data from a file, consider whether the file will be available for loading on the mirror server, or whether data in the source you are loading from will change by the time the mirror database processes the load. If either of these risks exists, consider specifying either WITH ROW LOGGING or WITH CONTENT LOGGING as the logging level in the LOAD TABLE statement. That way, the data loaded into the mirror database is identical to what was loaded in the mirrored database. See "LOAD TABLE statement" [*SQL Anywhere Server - SQL Reference*].

See also

- "Accessing data on client computers" on page 728
- "Introduction to database mirroring" [*SQL Anywhere Server - Database Administration*]
- "INPUT statement [Interactive SQL]" [*SQL Anywhere Server - SQL Reference*]
- "LOAD TABLE statement" [*SQL Anywhere Server - SQL Reference*]

Import data with the INSERT statement

Use the INSERT statement to add rows to the database. Because the import data for your destination table is included in the INSERT statement, it is considered interactive input. You can also use the INSERT statement with remote data access to import data from another database rather than a file.

Use the INSERT statement to import data when you:

- want to import small amounts of data into a single table
- are flexible with your file formats
- want to import remote data from an external database rather than from a file

The INSERT statement provides an ON EXISTING clause to specify the action to take if a row you are inserting is already found in the destination table. However, 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" [*SQL Anywhere Server - SQL Reference*].

Considerations for materialized views

For immediate views, an error is returned when you attempt to bulk load data into an underlying table. You must truncate the data in the view first, and then perform the bulk load operation.

For manual views, you can bulk load data into an underlying table; however, the data in the view becomes stale until the next refresh.

Consider truncating data in dependent materialized views before attempting a bulk load operation such as INSERT on a table. After you have loaded the data, refresh the view. See "TRUNCATE statement" [*SQL*

Anywhere Server - SQL Reference], and "REFRESH MATERIALIZED VIEW statement" [*SQL Anywhere Server - SQL Reference*].

Considerations for text indexes

For immediate text indexes, updating the text index after performing a bulk load operation such as INSERT on the underlying table can take a while even though the update is automatic. For manual text indexes, even a refresh can take a while.

Consider dropping dependent text indexes before performing a bulk load operation such as INSERT on a table. After you have loaded the data, recreate the text index. See "DROP TEXT INDEX statement" [*SQL Anywhere Server - SQL Reference*], and "CREATE TEXT INDEX statement" [*SQL Anywhere Server - SQL Reference*].

Impact on the database

Changes are recorded in the transaction log when you use the INSERT statement. This means that if there is a media failure involving the database file, you can recover information about the changes you made from the transaction log.

To import data (INSERT statement)

1. Ensure that the destination table exists.
2. Execute an INSERT statement. For example,

   The following example inserts a new row into the Departments table in the SQL Anywhere sample database.

   ```
   INSERT
   INTO Departments ( DepartmentID, DepartmentName, DepartmentHeadID )
   VALUES ( 700, 'Training', 501)
   SELECT * FROM Departments;
   ```

   Inserting values adds the new data to the existing table.

See also

- "The transaction log" [*SQL Anywhere Server - Database Administration*]
- "INSERT statement" [*SQL Anywhere Server - SQL Reference*]
- "LOAD TABLE statement" [*SQL Anywhere Server - SQL Reference*]
- "INPUT statement [Interactive SQL]" [*SQL Anywhere Server - SQL Reference*]

Import data with the MERGE statement

Use the MERGE statement to perform an update operation and update large amounts of table data. When you merge data, you can specify what actions to take when rows from the source data match or do not match the rows in the target data.

Defining the merge behavior

The following is an abbreviated version of the MERGE statement syntax. For the full MERGE statement syntax, see "MERGE statement" [*SQL Anywhere Server - SQL Reference*].

MERGE INTO *target-object*
USING *source-object*
ON *merge-search-condition*
{ **WHEN MATCHED** | **WHEN NOT MATCHED** } [...]

When the database performs a merge operation, it compares rows in *source-object* to rows in *target-object* to find rows that either match or do not match according to the definition contained in 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.

source-object can be a base table, view, materialized view, derived table, or the results of a procedure. *target-object* can be any of these objects except for materialized views and procedures. For further restrictions on these object types, see "MERGE statement" [*SQL Anywhere Server - SQL Reference*].

The ANSI SQL/2008 standard does not allow rows in *target-object* to be updated by more than one row in *source-object* during a merge operation.

Once a row in *source-object* is considered matching or non-matching, it is evaluated against the respective matching or non-matching WHEN clauses (WHEN MATCHED or WHEN NOT MATCHED). A WHEN MATCHED clause defines an action to perform on the row in *target-object* (for example, WHEN MATCHED ... UPDATE specifies to update the row in *target-object*). A WHEN NOT MATCHED clause defines an action to perform on the *target-object* using non-matching rows of the *source-object*.

You can specify unlimited WHEN clauses; they are processed in the order in which you specify them. You can also use the AND clause within a WHEN clause to specify actions against a subset of rows. For example, the following WHEN clauses define different actions to perform depending on the value of the Quantity column for matching rows:

```
WHEN MATCHED AND myTargetTable.Quantity<=500 THEN SKIP
WHEN MATCHED AND myTargetTable.Quantity>500 THEN UPDATE SET
myTargetTable.Quantity=500
```

Branches in a merge operation

The grouping of matched and non-matched rows by action is referred to as **branching**, and each group is referred to as a **branch**. A **branch** is equivalent to a single WHEN MATCHED or WHEN NOT MATCHED clause. For example, one branch might contain the set of non-matching rows from *source-object* that must be inserted. Execution of the branch actions begins only after all branching activities are complete (all rows in *source-object* have been evaluated). The database server begins executing the branch actions according to the order in which the WHEN clauses were specified.

Once a non-matching row from *source-object* or a pair of matching rows from *source-object* and *target-object* is placed in a branch, it is not evaluated against the succeeding branches. This makes the order in which you specify WHEN clauses significant.

A row in *source-object* that is considered a match or non-match, but does not belong to any branch (that is, it does not satisfy any WHEN clause) is ignored. This can occur when the WHEN clauses contain AND clauses, and the row does not satisfy any of the AND clause conditions. In this case, the row is ignored since no action is defined for it.

In the transaction log, actions that modify data are recorded as individual INSERT, UPDATE, and DELETE statements.

Triggers defined on the target table

Triggers fire normally as each INSERT, UPDATE, and DELETE statement is executed during the merge operation. For example, when processing a branch that has an UPDATE action defined for it, the database server:

1. fires all BEFORE UPDATE triggers
2. executes the UPDATE statement on the candidate set of rows while firing any row-level UPDATE triggers
3. fires the AFTER UPDATE triggers

Triggers on *target-table* can cause conflicts during a merge operation if it impacts rows that will be updated in another branch. For example, suppose an action is performed on row A, causing a trigger to fire that deletes row B. However, row B has an action defined for it that has not yet been performed. When an action cannot be performed on a row, the merge operation fails, all changes are rolled back, and an error is returned.

A trigger defined with more than one trigger action is treated as if it has been specified once for each of the trigger actions with the same body (that is, it is equivalent to defining separate triggers, each with a single trigger action).

Considerations for immediate materialized views

Database server performance might be affected if the MERGE statement updates a large number of rows. To update numerous rows, consider truncating data in dependent immediate materialized views before executing the MERGE statement on a table. After executing the MERGE statement, execute a REFRESH MATERIALIZED VIEW statement. See "REFRESH MATERIALIZED VIEW statement" [*SQL Anywhere Server - SQL Reference*] and "TRUNCATE statement" [*SQL Anywhere Server - SQL Reference*].

Considerations for text indexes

Database server performance might be affected if the MERGE statement updates a large number of rows. Consider dropping dependent text indexes before executing the MERGE statement on a table. After executing the MERGE statement, recreate the text index. See "DROP TEXT INDEX statement" [*SQL Anywhere Server - SQL Reference*], and "CREATE TEXT INDEX statement" [*SQL Anywhere Server - SQL Reference*].

Example 1

Suppose you own a small business selling jackets and sweaters. Prices on material for the jackets have gone up by 5% and you want to adjust your prices to match. Using the following CREATE TABLE statement, you create a small table called myProducts to hold current pricing information for the jackets and sweaters you sell. The subsequent INSERT statements populate myProducts with data.

```
CREATE TABLE myProducts (
   product_id    NUMERIC(10),
   product_name  CHAR(20),
```



```
   product_size  CHAR(20),
   product_price NUMERIC(14,2));
INSERT INTO myProducts VALUES (1, 'Jacket', 'Small', 29.99);
INSERT INTO myProducts VALUES (2, 'Jacket', 'Medium', 29.99);
INSERT INTO myProducts VALUES (3, 'Jacket', 'Large', 39.99);
INSERT INTO myProducts VALUES (4, 'Sweater', 'Small', 18.99);
INSERT INTO myProducts VALUES (5, 'Sweater', 'Medium', 18.99);
INSERT INTO myProducts VALUES (6, 'Sweater', 'Large', 19.99);
SELECT * FROM myProducts;
```

product_id	product_name	product_size	product_price
1	Jacket	Small	29.99
2	Jacket	Medium	29.99
3	Jacket	Large	39.99
4	Sweater	Small	18.99
5	Sweater	Medium	18.99
6	Sweater	Large	19.99

Now, use the following statement to create another table called myPrices to hold information about the price changes for jackets. A SELECT statement is added at the end so that you can see the contents of the myPrices table before the merge operation is performed.

```
CREATE TABLE myPrices (
   product_id    NUMERIC(10),
   product_name  CHAR(20),
   product_size  CHAR(20),
   product_price NUMERIC(14,2),
   new_price     NUMERIC(14,2));
INSERT INTO myPrices (product_id) VALUES (1);
INSERT INTO myPrices (product_id) VALUES (2);
INSERT INTO myPrices (product_id) VALUES (3);
INSERT INTO myPrices (product_id) VALUES (4);
INSERT INTO myPrices (product_id) VALUES (5);
INSERT INTO myPrices (product_id) VALUES (6);
COMMIT;
SELECT * FROM myPrices;
```

product_id	product_name	product_size	product_price	new_price
1	(NULL)	(NULL)	(NULL)	(NULL)
2	(NULL)	(NULL)	(NULL)	(NULL)
3	(NULL)	(NULL)	(NULL)	(NULL)
4	(NULL)	(NULL)	(NULL)	(NULL)
5	(NULL)	(NULL)	(NULL)	(NULL)

product_id	product_name	product_size	product_price	new_price
6	(NULL)	(NULL)	(NULL)	(NULL)

Use the following MERGE statement to merge data from the myProducts table into the myPrices table. Notice that the *source-object* is a derived table that has been filtered to contain only those rows where product_name is Jacket. Notice also that the ON clause specifies that rows in the *target-object* and *source-object* match if the values in their product_id columns match.

```
MERGE INTO myPrices p
USING ( SELECT
     product_id,
     product_name,
     product_size,
     product_price
  FROM myProducts
  WHERE product_name='Jacket') pp
ON (p.product_id = pp.product_id)
WHEN MATCHED THEN
  UPDATE SET
    p.product_id=pp.product_id,
    p.product_name=pp.product_name,
    p.product_size=pp.product_size,
    p.product_price=pp.product_price,
    p.new_price=pp.product_price * 1.05;
SELECT * FROM myPrices;
```

product_id	product_name	product_size	product_price	new_price
1	Jacket	Small	29.99	31.49
2	Jacket	Medium	29.99	31.49
3	Jacket	Large	39.99	41.99
4	(NULL)	(NULL)	(NULL)	(NULL)
5	(NULL)	(NULL)	(NULL)	(NULL)
6	(NULL)	(NULL)	(NULL)	(NULL)

The column values for product_id 4, 5, and 6 remain NULL because those products did not match any of the rows in the myProducts table whose products were (`product_name='Jacket'`).

Example 2

The following example merges rows from the mySourceTable and myTargetTable tables, using the primary key values of myTargetTable to match rows. The row is considered a match if a row in mySourceTable has the same value as the primary key column of myTargetTable.

```
MERGE INTO myTargetTable
   USING mySourceTable ON PRIMARY KEY
   WHEN NOT MATCHED THEN INSERT
   WHEN MATCHED THEN UPDATE;
```


The WHEN NOT MATCHED THEN INSERT clause specifies that rows found in mySourceTable that are not found in myTargetTable must be added to myTargetTable. The WHEN MATCHED THEN UPDATE clause specifies that the matching rows of myTargetTable are updated to the values in mySourceTable.

The following syntax is equivalent to the syntax above. It assumes that myTargetTable has the columns (I1, I2, .. I*n*) and that the primary key is defined on columns (I1, I2). The mySourceTable has the columns (U1, U2, .. U*n*).

```
MERGE INTO myTargetTable ( I1, I2, .. ., In )
   USING mySourceTable ON myTargetTable.I1 = mySourceTable.U1
      AND myTargetTable.I2 = mySourceTable.U2
   WHEN NOT MATCHED
      THEN INSERT ( I1, I2, .. In )
         VALUES ( mySourceTable.U1, mySourceTable.U2, ..., mySourceTable.Un )
   WHEN MATCHED
      THEN UPDATE SET
      myTargetTable.I1 = mySourceTable.U1,
      myTargetTable.I2 = mySourceTable.U2,
      ...
      myTargetTable.In = mySourceTable.Un;
```

Using the RAISERROR action

One of the actions you can specify for a match or non-match action is RAISERROR. RAISERROR allows you to fail the merge operation if the condition of a WHEN clause is met.

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. See "MERGE statement" [*SQL Anywhere Server - SQL Reference*].

Specifying a custom SQLCODE can be beneficial when, later, you are trying to determine the specific circumstances that caused the error to be raised.

The custom SQLCODE must be a positive integer greater than 17000, and can be specified either as a number or a variable.

The following statements provide a simple demonstration of how customizing a custom SQLCODE affects what is returned:

Create the table targetTable as follows:

```
CREATE TABLE targetTable( c1 int );
INSERT INTO targetTable VALUES( 1 );
COMMIT;
```

The following statement returns an error with SQLSTATE = '23510' and SQLCODE = -1254:

```
MERGE INTO targetTable
   USING (SELECT 1 c1 ) AS sourceData
   ON targetTable.c1 = sourceData.c1
   WHEN MATCHED THEN RAISERROR;
SELECT sqlstate, sqlcode;
```

The following statement returns an error with SQLSTATE = '23510' and SQLCODE = -17001:

```
MERGE INTO targetTable
   USING (SELECT 1 c1 ) AS sourceData
   ON targetTable.c1 = sourceData.c1
   WHEN MATCHED THEN RAISERROR 17001
   WHEN NOT MATCHED THEN RAISERROR 17002;
SELECT sqlstate, sqlcode;
```

The following statement returns an error with SQLSTATE = '23510' and SQLCODE = -17002:

```
MERGE INTO targetTable
   USING (SELECT 2 c1 ) AS sourceData
   ON targetTable.c1 = sourceData.c1
   WHEN MATCHED THEN RAISERROR 17001
   WHEN NOT MATCHED THEN RAISERROR 17002;
SELECT sqlstate, sqlcode;
```

Import data with proxy tables

A proxy table is a local table containing metadata used to access a table on a remote database server as if it were a local table. These let you import data directly.

Use proxy tables to import data when you:

- have access to remote data
- want to import data directly from another database

Impact on the database

Changes are recorded in the transaction log when you import using proxy tables. This means that if there is a media failure involving the database file, you can recover information about the changes you made from the transaction log.

How to use proxy tables

Create a proxy table, and then use an INSERT statement with a SELECT clause to insert data from the remote database into a permanent table in your database.

For more information about remote data access, see “Accessing remote data” on page 745.

For more information about INSERT statements, see “INSERT statement” [*SQL Anywhere Server - SQL Reference*].

Handling conversion errors during import

When you load data from external sources, there may be errors in the data. For example, there may be invalid dates and numbers. Use the conversion_error database option to ignore conversion errors and convert invalid values to NULL values.

For more information about setting database options, see "SET OPTION statement" [*SQL Anywhere Server - SQL Reference*], and "conversion_error option" [*SQL Anywhere Server - Database Administration*].

Import tables

To import a table

1. Ensure that the table you want to place the data in exists.
2. In Interactive SQL, from the **Data** menu choose **Import**.
3. Select **In A Text File** and click **Next**.
4. In the **File Name** field, click **Browse** to add the file.
5. Select **In An Existing Table**.
6. Click **Next**.
7. For ASCII files, specify the way the ASCII file is read and click **Next**.
8. Click **Import**.
9. Click **Close**.

To import a table (Interactive SQL SQL Statements pane)

1. Use the CREATE TABLE statement to create the destination table. For example:

```
CREATE TABLE GROUPO.Departments (
DepartmentID          integer NOT NULL,
DepartmentName        char(40) NOT NULL,
DepartmentHeadID      integer NULL,
CONSTRAINT DepartmentsKey PRIMARY KEY (DepartmentID) );
```

2. Execute a LOAD TABLE statement. For example,

```
LOAD TABLE Departments
FROM 'departments.csv';
```

3. To keep trailing blanks in your values, use the STRIP OFF clause in your LOAD TABLE statement. The default setting (STRIP ON) strips trailing blanks from values before inserting them.

   The LOAD TABLE statement adds the contents of the file to the existing rows of the table; it does not replace the existing rows in the table. You can use the TRUNCATE TABLE statement to remove all the rows from a table.

   Neither the TRUNCATE TABLE statement nor the LOAD TABLE statement fires triggers or perform referential integrity actions, such as cascaded deletes.

See also

- "CREATE TABLE statement" [*SQL Anywhere Server - SQL Reference*]
- "LOAD TABLE statement" [*SQL Anywhere Server - SQL Reference*]

Table structures for import

The structure of the source data does not need to match the structure of the destination table itself. For example, the column data types may be different or in a different order, or there may be extra values in the import data that do not match columns in the destination table.

Rearranging the table or data

If you know that the structure of the data you want to import does not match the structure of the destination table, you can:

- provide a list of column names to be loaded in the LOAD TABLE statement
- rearrange the import data to fit the table with a variation of the INSERT statement and a global temporary table
- use the INPUT statement to specify a specific set or order of columns

Allowing columns to contain NULL values

If the file you are importing contains data for a subset of the columns in a table, or if the columns are in a different order, you can also use the LOAD TABLE statement DEFAULTS option to fill in the blanks and merge non-matching table structures.

- If DEFAULTS is OFF, any column not present in the column list is assigned NULL. 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.
- If DEFAULTS is ON and the column has a default value, that value is used.

For example, you can define a default value for the City column in the Customers table and then load new rows into the Customers table from a fictitious file called new_customers.txt using a LOAD TABLE statement like this:

```
ALTER TABLE Customers
ALTER City DEFAULT 'Waterloo';
LOAD TABLE Customers ( Surname, GivenName, Street, State, Phone )
FROM 'new_customers.txt'
DEFAULTS ON;
```

Since a value is not provided for the City column, the default value is supplied. If `DEFAULTS OFF` had been specified, the City column would have been assigned the empty string.

Merge different table structures

Use a variation of the INSERT statement and a global temporary table to rearrange the import data to fit the table.

To load data with a different structure using a global temporary table

1. In the **SQL Statements** pane, create a global temporary table with a structure matching that of the input file.

   You can use the CREATE TABLE statement to create the global temporary table.

2. Use the LOAD TABLE statement to load your data into the global temporary table.

   When you close the database connection, the data in the global temporary table disappears. However, the table definition remains. You can use it the next time you connect to the database.

3. Use the INSERT statement with a SELECT clause to extract and summarize data from the temporary table and copy the data into one or more permanent database tables.

See also

- "CREATE TABLE statement" [*SQL Anywhere Server - SQL Reference*]
- "LOAD TABLE statement" [*SQL Anywhere Server - SQL Reference*]

Importing binary files

You can import binary files, such as JPEG, bitmap, or Microsoft Word files, into your database using the xp_read_file system procedure. See "Inserting documents and images" on page 519.

Exporting data

Exporting data is an administrative task that involves writing data out of your database. Exporting data is a useful if you need to share large portions of your database, or extract portions of your database according to particular criteria. Use SQL Anywhere to:

- export individual tables, query results, or table schema
- create scripts that automate exporting so that you can export several tables consecutively
- export to many different file formats
- export data to a file on a client computer
- export files between SQL Anywhere and Adaptive Server Enterprise using the BCP FORMAT clause

Before exporting data, determine what resources you have and the type of information you want to export from your database.

For performance reasons, if you want to export an entire database, unload the database instead of exporting the data. See "Rebuilding databases" on page 731.

Export limitations

When exporting data from a SQL Anywhere database to an Excel database with the Microsoft Excel ODBC driver, the following data type changes can occur:

- When you export data that is stored as CHAR, LONG VARCHAR, NCHAR, NVARCHAR or LONG NVARCHAR data type, the data is stored as VARCHAR (the closest type supported by the Excel driver).

  Note that the Microsoft Excel ODBC driver supports text column widths up to 255 characters.

- Data stored as MONEY and SMALLMONEY data types is exported to the CURRENCY data type. Otherwise numerical data is exported as numbers.

See also

- "UNLOAD statement" [*SQL Anywhere Server - SQL Reference*]
- "OUTPUT statement [Interactive SQL]" [*SQL Anywhere Server - SQL Reference*]
- "Performance aspects of bulk operations" on page 699
- "Use the OUTPUT statement to output NULLs" on page 726
- "Accessing data on client computers" on page 728

Export data with the Export Wizard

Use the **Export Wizard** to export query results in a specific format to a file or database.

To export result sets data using Interactive SQL

1. Execute a query.
2. In Interactive SQL, choose **Data** » **Export**.
3. Follow the instructions in the **Export Wizard** wizard.

To export result sets data to an UltraLite database using Interactive SQL

1. Execute the following query while connected to the SQL Anywhere sample database.

   ```
   SELECT * FROM Employees
   WHERE State = 'GA';
   ```

   The result set includes a list of all employees who live in Georgia.

2. Choose **Data** » **Export**.
3. Click **In A Database**.
4. In the **Database Type** list, select **UltraLite**.

5. In the **User Id** field, type **dba**.
6. In the **Password** field, type **sql**.
7. In the **Database File** field, type *C:\Documents and Settings\All Users\Documents\SQL Anywhere 12\Samples\UltraLite\CustDB\custdb.udb*.
8. Click **Next**.
9. Click **In A New Table**.
10. In the **Owner** field, type **dba**.
11. In the **Table Name** field, type **NewTable**.
12. Click **Export**.
13. Click **Close**.
14. Choose **SQL** » **Previous SQL**.

    The OUTPUT USING statement created and used by the **Export Wizard** appears in the **SQL Statements** pane:

```
--  Generated by the Export Wizard
OUTPUT USING 'driver=UltraLite 12;UID=dba;PWD=sql;
DBF=C:\Documents and Settings\All Users\Documents\SQL Anywhere 12\Samples
\UltraLite\CustDB\custdb.udb'
INTO "dba"."NewTable"
CREATE TABLE ON
```

Export data with the OUTPUT statement

Use the OUTPUT statement to export query results, tables, or views from your database.

The OUTPUT statement is useful when compatibility is an issue because it can write out the result set of a SELECT statement in several different file formats. You can use the default output format, or you can specify the file format on each OUTPUT statement. Interactive SQL can execute a command file containing multiple OUTPUT statements.

The default Interactive SQL output format is specified on the **Import/Export** tab of the **Interactive SQL Options** window (accessed by choosing **Tools** » **Options** in Interactive SQL).

Use the Interactive SQL OUTPUT statement when you want to:

- export all or part of a table or view in a format other than text
- automate the export process using a command file

Impact on the database

If you have a choice between using the OUTPUT statement, UNLOAD statement, or UNLOAD TABLE statement, choose the UNLOAD TABLE statement for performance reasons.

There are performance impacts associated with exporting large amounts of data with the OUTPUT statement. Use the OUTPUT statement on the same computer as the server if possible to avoid sending large amounts of data across the network.

For more information, see "OUTPUT statement [Interactive SQL]" [*SQL Anywhere Server - SQL Reference*].

Example

The following example exports the data from the Employees table in the SQL Anywhere sample database to a *.txt* file named *Employees.txt*.

```
SELECT * FROM Employees;
OUTPUT TO Employees.txt
    FORMAT TEXT;
```

The following example exports data from the Employees table in the SQL Anywhere sample database to a new table in a SQL Anywhere database named *mydatabase.db*

```
SELECT * FROM Employees;
OUTPUT USING 'driver=SQL Anywhere 12;UID=dba;PWD=sql;DBF=C:\Tobedeleted
\mydatabase.db;CON='''''
    INTO "dba"."newcustomers"
    CREATE TABLE ON;
```

To export data to an Excel file using the OUTPUT statement (Interactive SQL)

1. In Interactive SQL, connect to a SQL Anywhere database.
2. Execute an OUTPUT statement using the READONLY clause. For example:

   ```
   SELECT * FROM SalesOrders;
   OUTPUT USING 'Driver=Microsoft Excel Driver (*.xls);
   DBQ=c:\\test\\sales.xls;
   READONLY=0' INTO "newSalesData";
   ```

   A new Excel file, named *sales.xls*, is created. It will contain a worksheet called newSalesData.

   Note that the Microsoft Excel driver is a 32-bit driver so the 32-bit version of Interactive SQL is required for this example.

To export data to a CSV file

1. In Interactive SQL, connect to the SQL Anywhere database.
2. Execute an OUTPUT statement with the clauses FORMAT TEXT, QUOTE "", and WITH COLUMN NAMES to create a comma-delimited format with the column names in the first line of the file. String values will be enclosed with quotation marks.

   ```
   SELECT * FROM SalesOrders;
   OUTPUT TO 'c:\\test\\sales.csv'
   ```



```
FORMAT TEXT
QUOTE '"'
WITH COLUMN NAMES;
```

Export data with the UNLOAD TABLE statement

The UNLOAD TABLE statement lets you export data efficiently in text formats only. The UNLOAD TABLE statement exports one row per line, with values separated by a comma delimiter. To make reloading faster, the data is exported in order by primary key values.

Use the UNLOAD TABLE statement when you:

- want to export entire tables in text format
- are concerned about database performance
- export data to a file on a client computer

Impact on the database

The UNLOAD TABLE statement places an exclusive lock on the whole table while you are unloading it.

If you have a choice between using the OUTPUT statement, UNLOAD statement, or UNLOAD TABLE statement, choose the UNLOAD TABLE statement for performance reasons.

Example

Using the SQL Anywhere sample database, you can unload the Employees table to a text file named *employee_data.csv* by executing the following command:

```
UNLOAD TABLE Employees TO 'employee_data.csv';
```

Because it is the database server that unloads the table, *employee_data.csv* specifies a file on the database server computer.

See also

- “Accessing data on client computers” on page 728
- “UNLOAD statement” [*SQL Anywhere Server - SQL Reference*]
- “OUTPUT statement [Interactive SQL]” [*SQL Anywhere Server - SQL Reference*]

Export data with the UNLOAD statement

The UNLOAD statement is similar to the OUTPUT statement in that they both export query results to a file. However, the UNLOAD statement exports data more efficiently in a text format. The UNLOAD statement exports with one row per line, with values separated by a comma delimiter.

Use the UNLOAD statement to unload data when you want to:

- export query results if performance is an issue
- store output in text format
- embed an export command in an application
- export data to a file on a client computer

Impact on the database

If you have a choice between using the OUTPUT statement, UNLOAD statement, or UNLOAD TABLE statement, choose the UNLOAD TABLE statement for performance reasons.

To use the UNLOAD statement, the user must have the permissions required to execute the SELECT that is specified within the UNLOAD statement.

For more information about controlling who can use the UNLOAD statement, see "-gl dbeng12/dbsrv12 server option" [*SQL Anywhere Server - Database Administration*].

The UNLOAD statement is executed at the current isolation level.

Example

Using the SQL Anywhere sample database, you can unload a subset of the Employees table to a text file named *employee_data.csv* by executing the following command:

```
UNLOAD
SELECT * FROM Employees
WHERE State = 'GA'
TO 'employee_data.csv'
QUOTE '"';
```

Because it is the database server that unloads the result set, *employee_data.csv* specifies a file on the database server computer.

See also

- "Accessing data on client computers" on page 728
- "UNLOAD statement" [*SQL Anywhere Server - SQL Reference*]
- "OUTPUT statement [Interactive SQL]" [*SQL Anywhere Server - SQL Reference*]

Export data with the dbunload utility

Use the dbunload utility to export one, many, or all the database tables. You can export table data, and table schemas. To rearrange your database tables, you can also use the dbunload utility to create the necessary command files and modify them as needed. You can unload tables with structure only, data only, or with both structure and data.

You can also extract one or many tables with or without command files. These files can be used to create identical tables in different databases.

Note
The dbunload utility is functionally equivalent to the Sybase Central **Unload Database Wizard**. You can use either one interchangeably to produce the same results.

Use the dbunload utility when you:

- need to rebuild or extract your database
- want to export data in text format
- need to process large amounts of data quickly
- have flexible file format requirements

For more information about the dbunload utility, see "Unload utility (dbunload)" [*SQL Anywhere Server - Database Administration*].

Export data with the Unload Database Wizard

Use the **Unload Database Wizard** to unload an existing database into a new database.

When using the **Unload Database Wizard** to unload your database, you can choose to unload all the objects in a database, or a subset of tables from the database. Only tables for users selected in the **Configure Owner Filter** window appear in the **Unload Database Wizard**. If you want to view tables belonging to a particular database user, right-click the database you are unloading, choose **Configure Owner Filter**, and then select the user in the resulting window.

Note
When you unload only tables, the user IDs that own the tables are not unloaded. You must create the user IDs that own the tables in the new database before reloading the tables.

You can also use the **Unload Database Wizard** to unload an entire database in text comma-delimited format and to create the necessary Interactive SQL command files to completely recreate your database. This is useful for creating SQL Remote extractions or building new copies of your database with the same or a slightly modified structure. The **Unload Database Wizard** is useful for exporting SQL Anywhere files intended for reuse within SQL Anywhere.

The **Unload Database Wizard** also gives you the option to reload into an existing database or a new database, rather than into a reload file.

The dbunload utility is functionally equivalent to the **Unload Database Wizard**. You can use either one interchangeably to produce the same results.

Note
If the database you want to unload is already running, and you start the **Unload Database Wizard**, the SQL Anywhere 12 plug-in automatically stops the database before you can unload it.

For information about special considerations when unloading a database, see "Unload utility (dbunload)" [*SQL Anywhere Server - Database Administration*].

To unload a database file or a running database (Sybase Central)

1. Choose **Tools** » **SQL Anywhere 12** » **Unload Database**.
2. Follow the instructions in the **Unload Database Wizard**.

Export data with the Unload Data window

You can use the **Unload Data** window in Sybase Central to unload one or more tables in a database. This functionality is also available with either the **Unload Database Wizard** or the Unload utility (dbunload), but this window allows you to unload tables in one step, instead of completing the entire **Unload Database Wizard**.

To unload tables using the Unload Data window

1. Use the SQL Anywhere 12 plug-in to connect to the database as a user with DBA authority.
2. Double-click **Tables**.
3. Right-click the table you want to export data from, and choose **Unload Data**.
4. Complete the **Unload Data** window. Click **OK**.

Export query results

Use the **Data** menu, the OUTPUT statement, or the UNLOAD statement to export queries (including queries on views) to a file.

Use the BCP FORMAT clause to import and export files between SQL Anywhere and Adaptive Server Enterprise. For more information, see "Adaptive Server Enterprise compatibility" on page 745.

To export query results (Interactive SQL Data menu)

1. Enter your query in the **SQL Statements** pane of Interactive SQL.
2. Choose **SQL** » **Execute**.
3. Choose **Data** » **Export**.
4. Specify a location for the results and click **Next**.
5. For text, HTML, and XML files, type a file name in the **File Name** field and click **Export**.

   For an ODBC database:

a. Select a database and click **Next**.

b. Select a location to save the data and click **Export**.

6. Click **Close**.

To export query results (Interactive SQL OUTPUT statement)

1. Enter your query in the **SQL Statements** pane of Interactive SQL.

2. At the end of the query, type **OUTPUT TO** '*filename*'. For example, to export the entire Employees table to the file *employees.txt*, enter the following query:

```
SELECT *
FROM Employees;
OUTPUT TO 'employees.txt';
```

3. To export query results and append the results to another file, use the APPEND clause.

```
SELECT * FROM Employees;
OUTPUT TO 'employees.txt'
APPEND;
```

To export query results and include messages, use the VERBOSE clause.

```
SELECT * FROM Employees;
OUTPUT TO 'employees.txt'
VERBOSE;
```

4. Choose **SQL** » **Execute**.

If the export is successful, the **Messages** tab displays the amount of time it to took to export the query result set, the file name and path of the exported data, and the number of rows written. If the export is unsuccessful, a message appears indicating that the export was unsuccessful.

For more information about exporting query results using the OUTPUT statement, see “OUTPUT statement [Interactive SQL]” [*SQL Anywhere Server - SQL Reference*].

Tips

You can combine the APPEND and VERBOSE clauses to append both results and messages to an existing file.

For example, type **OUTPUT TO** '*filename*' **APPEND VERBOSE**.

The OUTPUT statement with its clauses APPEND and VERBOSE are equivalent to the >#, >>#, >&, and >>& operators of earlier versions of Interactive SQL. You can still use these operators to redirect data, but the new Interactive SQL statements allow for more precise output and easier to read code.

For more information about APPEND and VERBOSE, see “OUTPUT statement [Interactive SQL]” [*SQL Anywhere Server - SQL Reference*].

To export query results (UNLOAD statement)

1. In the **SQL Statements** pane, enter the UNLOAD statement. For example,

```
UNLOAD
SELECT *
FROM Employees
TO 'employee_data.csv';
```

2. Choose **SQL** » **Execute**.

   If the export is successful, the **Messages** tab displays the amount of time it to took to export the query result set, the file name and path of the exported data, and the number of rows written. If the export is unsuccessful, a message appears indicating that the export was unsuccessful.

Use the OUTPUT statement to output NULLs

You may want to extract data for use in other software products. Because the other software products may not understand NULL values, there are two ways to specify how NULL values appear when using the OUTPUT statement from Interactive SQL:

- the output_nulls option lets you specify the output value used by the OUTPUT statement
- the IFNULL function lets you apply the output value to a particular instance or query

Both options allow you to output a specific value in place of a NULL value. By specifying how NULL values are output, you have greater compatibility with other software products.

To specify a NULL value output (Interactive SQL)

- Execute a SET OPTION statement that changes the value of the output_nulls option. The following example changes the value that appears for NULL values to (unknown):

```
SET OPTION output_nulls = '(unknown)';
```

For more information about setting Interactive SQL options, see "SET OPTION statement" [*SQL Anywhere Server - SQL Reference*].

To change the value that appears in place of a NULL value on the Results pane (Interactive SQL)

1. Choose **Tools** » **Options**.
2. Click **SQL Anywhere**.
3. Click the **Results** tab.
4. In the **Display Null Values As** field, type **Value**.

5. Click **OK**.

Export databases

> **Note**
> If the database you want to unload is already running, and you start the **Unload Database Wizard**, the SQL Anywhere 12 plug-in automatically stops the database before you can unload it.

To unload all or part of a database (Sybase Central)

1. Choose **Tools** » **SQL Anywhere 12** » **Unload Database**.
2. Follow the instructions in the **Unload Database Wizard**.

To unload all or part of a database (command line)

- Run the dbunload utility, and use the -c option to specify the connection parameters.

  To unload the entire database to the directory *c:\DataFiles* on the server computer:

  ```
  dbunload -c "DBN=demo;UID=DBA;PWD=sql" c:\DataFiles
  ```

  The statements required to recreate the schema and reload the tables are written to *reload.sql* in the local current directory.

  To export data only, use -d. For example:

  ```
  dbunload -c "DBN=demo;UID=DBA;PWD=sql" -d c:\DataFiles
  ```

  The statements required to reload the tables are written to *reload.sql* in the local current directory.

  To export schema only, use -n. For example:

  ```
  dbunload -c "DBN=demo;UID=DBA;PWD=sql" -n
  ```

  The statements required to recreate the schema are written to *reload.sql* in the local current directory.

For more information, see "Unload utility (dbunload)" [*SQL Anywhere Server - Database Administration*].

Export tables

You can also export a table by selecting all the data in a table and exporting the query results. See "Export query results" on page 724.

Use the same procedures to export views.

To export a table (command line)

- Run the following command:

```
dbunload -c "DBN=demo;UID=DBA;PWD=sql"
-t Employees c:\DataFiles
```

  In this command, -c specifies the database connection parameters and -t specifies the name of the table or tables you want to export. This dbunload command unloads the data from the SQL Anywhere sample database (assumed to be running on the default database server with the default database name) into a set of files in the *c:\DataFiles* directory on the server computer. A command file to rebuild the tables from the data files is created with the default name *reload.sql* in the local current directory.

  You can unload more than one table by separating the table names with a comma (,) delimiter.

To export a table (SQL)

- Execute an UNLOAD TABLE statement. For example,

```
UNLOAD TABLE Departments
TO 'departments.csv';
```

  This statement unloads the Departments table from the SQL Anywhere sample database into the file *departments.csv* in the database server's current working directory. If you are running against a network database server, the command unloads the data into a file on the server computer, not the client computer. Also, the file name passes to the server as a string. Using escape backslash characters in the file name prevents misinterpretation if a directory or file name begins with an n (\n is a newline character) or any other special characters.

  Each row of the table is output on a single line of the output file, and no column names are exported. The columns are delimited by a comma. The delimiter character can be changed using the DELIMITED BY clause. The fields are not fixed-width fields. Only the characters in each entry are exported, not the full width of the column.

See also

- "Export query results" on page 724
- "Unload utility (dbunload)" [*SQL Anywhere Server - Database Administration*]
- "UNLOAD statement" [*SQL Anywhere Server - SQL Reference*]

Accessing data on client computers

SQL Anywhere allows you to load data from, and unload data to, a file on a client computer using SQL statements and functions, without requiring copying files to the database server computer. To do this, the database server initiates the transfer using a Command Sequence communication protocol (CmdSeq) file handler. The CmdSeq file handler is invoked after the database server receives a request from the client application requiring a transfer of data to or from the client computer, and before sending the response. The file handler supports simultaneous and interleaved transfer of multiple files from the client at any

given time. For example, the database server can initiate the transfer of multiple files simultaneously if the statement executed by the client application requires it.

Using a CmdSeq file handler to achieve transfer of client data means that applications do not require any new specialized code and can start benefitting immediately from the feature using the SQL components listed below:

- **READ_CLIENT_FILE function** The READ_CLIENT_FILE function reads data from the specified file on the client computer, and returns a LONG BINARY value representing the contents of the file. This function can be used anywhere in SQL code that a BLOB can be used. The data returned by the READ_CLIENT_FILE function is not materialized in memory when possible, unless the statement explicitly causes materialization to take place. For example, the LOAD TABLE statement streams the data from the client file without materializing it. Assigning the value returned by the READ_CLIENT_FILE function to a connection variable causes the database server to retrieve and materialize the client file contents. See "READ_CLIENT_FILE function [String]" [*SQL Anywhere Server - SQL Reference*].

- **WRITE_CLIENT_FILE function** The WRITE_CLIENT_FILE function writes data to the specified file on the client computer. See "WRITE_CLIENT_FILE function [String]" [*SQL Anywhere Server - SQL Reference*].

- **READCLIENTFILE authority** READCLIENTFILE authority allows you to read from a file on a client computer. See "READCLIENTFILE authority" [*SQL Anywhere Server - Database Administration*].

- **WRITECLIENTFILE authority** WRITECLIENTFILE authority allows you to write to a file on a client computer. See "WRITECLIENTFILE authority" [*SQL Anywhere Server - Database Administration*].

- **LOAD TABLE ... USING CLIENT FILE clause** The USING CLIENT FILE clause allows you to load a table using data in a file located on the client computer. For example, `LOAD TABLE ... USING CLIENT FILE 'my-file.txt';` loads a file called *my-file.txt* from the client computer. See "LOAD TABLE statement" [*SQL Anywhere Server - SQL Reference*].

- **LOAD TABLE ... USING VALUE clause** The USING VALUE clause allows you to specify a BLOB expression as a value. The BLOB expression can make use of the READ_CLIENT_FILE function to load a BLOB from a file on a client computer. For example, `LOAD TABLE ... USING VALUE READ_CLIENT_FILE( 'my-file' )`, where *my-file* is a file on the client computer. See "LOAD TABLE statement" [*SQL Anywhere Server - SQL Reference*].

- **UNLOAD TABLE ... INTO CLIENT FILE clause** The INTO CLIENT FILE clause allows you to specify a file on the client computer to unload data into. See "UNLOAD statement" [*SQL Anywhere Server - SQL Reference*].

- **UNLOAD TABLE ... INTO VARIABLE clause** The INTO VARIABLE clause allows you to specify a variable to unload data into. See "UNLOAD statement" [*SQL Anywhere Server - SQL Reference*].

- **read_client_file and write_client_file secure features** The read_client_file and write_client_file secure features control the use of statements that can cause a client file to be read

from, or written to. See "Specifying secured features" [*SQL Anywhere Server - Database Administration*], and "-sf dbeng12/dbsrv12 server option" [*SQL Anywhere Server - Database Administration*].

Note that in order to allow reading from or writing to a client file from a procedure, function or other indirect statements, a callback function must be registered. The callback function is called to confirm that the application allows the client transfer that it did not directly request.

See also

- "SQLSetConnectAttr extended connection attributes" [*SQL Anywhere Server - Programming*] (ODBC)
- "db_register_a_callback function" [*SQL Anywhere Server - Programming*] (ESQL)
- "Using JDBC callbacks" [*SQL Anywhere Server - Programming*] (JDBC)

Client-side data security

SQL Anywhere provides means to ensure that the transfer of client files does not permit the unauthorized transfer of data residing on the client computer, which is often in a different location than the database server computer.

To do this, the database server tracks the origin of each executed statement, and determines if the statement was received directly from the client application. When initiating the transfer of a new file from the client, the database server includes information about the origin of the statement. The CmdSeq file handler then allows the transfer of files for statements sent directly by the client application. If the statement was not sent directly by the client application, the application must register a verification callback. If no callback is registered, the transfer is denied and the statement fails with an error.

Also, the transfer of client data is not allowed until after the connection has been successfully established. This restriction prevents unauthorized access using connection strings or login procedures.

To protect against attempts to gain access to a system by users posing as an authorized user, consider encrypting the data that is being transferred.

SQL Anywhere also provides the following security mechanisms to control access at various levels:

- **Server level security** The read_client_file and write_client_file secured features allow you to disable all client-side transfers on a server-wide basis. See "-sf dbeng12/dbsrv12 server option" [*SQL Anywhere Server - Database Administration*].

- **Application and DBA level security** The allow_read_client_file and allow_write_client_file database options provide access control at the database, user, or connection level. For example, an application could set this database option to OFF after connecting to prevent itself from being used for any client-side transfers. See "allow_read_client_file option" [*SQL Anywhere Server - Database Administration*].

- **User level security** READCLIENTFILE and WRITECLIENTFILE authority provides user level access control for reading data from, and writing data to, a client computer, respectively. See "READCLIENTFILE authority" [*SQL Anywhere Server - Database Administration*], and "WRITECLIENTFILE authority" [*SQL Anywhere Server - Database Administration*].

Planning for recovery when loading client-side data

If you need to recover a LOAD TABLE statement from your transaction log, files on the client computer that you used to load data are likely no longer available to SQL Anywhere, or have changed, so the original data is no longer available. To prevent this situation from occurring, make sure that logging is not turned off. Then, specify either the WITH ROW LOGGING or WITH CONTENT LOGGING clauses when loading the data. These clauses cause the data you are loading to be recorded in the transaction log, so that they can be replayed later in the event of a recovery.

The WITH ROW LOGGING causes each inserted row to be recorded as an INSERT statement in the transaction log. The WITH CONTENT LOGGING causes the inserted data to be recorded in the transaction log in chunks for the database server to process during recovery. Both methods are suitable for ensuring that the client-side data is available for loading during recovery. However, you cannot use WITH CONTENT LOGGING when loading data into a database that is involved in synchronization.

When you specify any of the following LOAD TABLE statements, but do not specify a logging level, WITH CONTENT LOGGING is the default behavior:

- LOAD TABLE ... USING CLIENT FILE *client-filename-expression*
- LOAD TABLE ... USING VALUE *value-expression*
- LOAD TABLE ... USING COLUMN *column-expression*

For more information about how to record loaded data in the transaction log during a load operation, see "LOAD TABLE statement" [*SQL Anywhere Server - SQL Reference*].

Rebuilding databases

Rebuilding a database is a specific type of import and export involving unloading and reloading your entire database. The rebuild (unload/load) and extract procedures are used to rebuild databases, to create new databases from part of an existing one, and to eliminate unused free pages.

If you are rebuilding your database to upgrade it to a newer version of SQL Anywhere, see "Upgrading SQL Anywhere" [*SQL Anywhere 12 - Changes and Upgrading*].

You can rebuild your database from Sybase Central or by using the dbunload utility.

Note
It is good practice to make backups of your database before rebuilding, especially if you choose to replace the original database with the rebuilt database.

For more information, see "Backup and data recovery" [*SQL Anywhere Server - Database Administration*].

With importing and exporting, the destination of the data is either into your database or out of your database. Importing reads data into your database. Exporting writes data out of your database. Often the information is either coming from or going to another non-SQL Anywhere database.

If you specify the encryption options -ek, -ep, or -et, the LOAD TABLE statements in the *reload.sql* file must include the encryption key. Hard-coding the key compromises security, so a parameter in the *reload.sql* file specifies the encryption key. When you execute the *reload.sql* file with Interactive SQL, you must specify the encryption key as a parameter. If you do not specify the key in the READ statement, Interactive SQL prompts for the key. See "Interactive SQL utility (dbisql)" [*SQL Anywhere Server - Database Administration*].

Loading and unloading takes data and schema out of a SQL Anywhere database and then places the data and schema back into a SQL Anywhere database. The unloading procedure produces data files and a *reload.sql* file which contains table definitions required to recreate the table exactly. Running the *reload.sql* script recreates the tables and loads the data back into them.

Rebuilding a database can be a time-consuming operation, and can require a large amount of disk space. As well, the database is unavailable for use while being unloaded and reloaded. For these reasons, rebuilding a database is not advised in a production environment unless you have a definite goal in mind.

From one SQL Anywhere database to another

Rebuilding generally copies data out of a SQL Anywhere database and then reloads that data back into a SQL Anywhere database. Unloading and reloading are related since you usually perform both tasks, rather than just one or the other.

Rebuilding versus exporting

Rebuilding is different from exporting in that rebuilding exports and imports table definitions and schema in addition to the data. The unload portion of the rebuild process produces text format data files and a *reload.sql* file that contains table and other definitions. You can run the *reload.sql* script to recreate the tables and load the data into them.

For more information, see "Internal versus external unloads and reloads" [*SQL Anywhere Server - Database Administration*].

Consider extracting a database (creating a new database from an old database) if you are using SQL Remote or MobiLink. See "Extracting databases" on page 739.

Rebuilding replicating databases

The procedure for rebuilding a database depends on whether the database is involved in replication or not. If the database is involved in replication, you must preserve the transaction log offsets across the operation, as the Message Agent requires this information. If the database is not involved in replication, the process is simpler.

See also

- "Minimize downtime when rebuilding a database" on page 738
- "Rebuild databases involved in synchronization or replication" on page 734
- "Rebuild databases not involved in synchronization or replication" on page 734
- "Change a database from one collation to another" [*SQL Anywhere Server - Database Administration*]
- "Refresh manual views" on page 45

Reasons to rebuild databases

There are several reasons to consider rebuilding your database. You might rebuild your database if you want to do any of the following:

- **Upgrade your database file format** Some new features are made available by applying the Upgrade utility, but others require a database file format upgrade, which is performed by unloading and reloading the database. To determine if an unload and reload is required to obtain a new feature, see "Upgrading to SQL Anywhere 12" [*SQL Anywhere 12 - Changes and Upgrading*].

  New versions of the SQL Anywhere database server can be used without upgrading your database. If you want to use features of the new version that require access to new system tables or database options, you must use the Upgrade utility to upgrade your database. The Upgrade utility does not unload or reload any data.

  If you want to use the new version of SQL Anywhere that relies on changes in the database file format, you must unload and reload your database. You should back up your database before rebuilding the database.

  Note
  If you are upgrading from version 9 or earlier, you must rebuild the database file. If you are upgrading from version 10.0.0 or later, you can use the Upgrade utility or rebuild your database.

  For more information about upgrading your database, see "Upgrading SQL Anywhere" [*SQL Anywhere 12 - Changes and Upgrading*].

  For information about upgrading SQL Anywhere or rebuilding a database involved in a database mirroring system, see "Upgrading SQL Anywhere software and databases in a database mirroring system" [*SQL Anywhere 12 - Changes and Upgrading*].

- **Reclaim disk space** Databases do not shrink if you delete data. Instead, any empty pages are simply marked as free so they can be used again. They are not removed from the database unless you rebuild it. Rebuilding a database can reclaim disk space if you have deleted a large amount of data from your database and do not anticipate adding more.

- **Improve database performance** Rebuilding databases can improve performance. Since the database can be unloaded and reloaded in order by primary keys, access to related information can be faster as related rows may appear on the same or adjacent pages.

Note
If you detect that performance is poor because a table is highly fragmented, you can reorganize the table. See "REORGANIZE TABLE statement" [*SQL Anywhere Server - SQL Reference*].

See also

- "Upgrade utility (dbupgrad)" [*SQL Anywhere Server - Database Administration*]
- "Unload utility (dbunload)" [*SQL Anywhere Server - Database Administration*]

Rebuild databases not involved in synchronization or replication

The following procedures should be used only if your database is not involved in synchronization or replication.

To rebuild a database not involved in synchronization or replication (command line)

1. Run the dbunload utility, specifying one of the following options:

To do this...	Use this option...	Example
Rebuild to a new database	-an	`dbunload -c "DBF=demo.db;UID=DBA;PWD=sql" -an DemoBackup.db`
Reload to an existing database	-ac	`dbunload -c "DBF=demo.db;UID=DBA;PWD=sql" -ac "UID=DBA;PWD=sql;DBF=mynewdemo.db"`
Replace an existing database	-ar	`dbunload -c "DBF=demo.db;UID=DBA;PWD=sql" -ar`

   If you use one of these options, no interim copy of the data is created on disk, so you do not need to specify an unload directory on the command line. This provides greater security for your data. The -ar and -an options should also execute more quickly than the **Unload Database Wizard** in Sybase Central, but -ac is slower than the **Unload Database Wizard**.

2. Shut down the database and archive the transaction log before using the reloaded database.

Notes

The -an and -ar options only apply to connections to a personal server, or connections to a network server over shared memory.

There are additional options available for the dbunload utility that allow you to tune the unload and connection parameter options that allow you to specify a running or non-running database and database parameters.

Rebuild databases involved in synchronization or replication

This section applies to SQL Anywhere MobiLink clients (clients using dbmlsync) and SQL Remote.

If a database is participating in synchronization or replication, particular care needs to be taken if you want to rebuild the database. Synchronization and replication are based on the offsets in the transaction log. When you rebuild a database, the offsets in the old transaction log are different than the offsets in the new log, making the old log unavailable. For this reason, good backup practices are especially important when participating in synchronization or replication.

There are two ways of rebuilding a database involved in synchronization or replication. The first method uses the dbunload utility -ar option to make the unload and reload occur in a way that does not interfere with synchronization or replication. The second method is a manual method of doing the same task.

All subscriptions must be synchronized before rebuilding a database participating in MobiLink synchronization.

To rebuild a database involved in synchronization or replication (dbunload utility)

1. Shut down the database.
2. Perform a full off-line backup by copying the database and transaction log files to a secure location.
3. Run the following dbunload command to rebuild the database:

   ```
   dbunload -c connection-string -ar directory
   ```

   The *connection-string* is a connection with DBA authority, and *directory* is the directory used in your replication environment for old transaction logs. There can be no other connections to the database.

   The -ar option only applies to connections to a personal server, or connections to a network server over shared memory.

   For more information, see "Unload utility (dbunload)" [*SQL Anywhere Server - Database Administration*].
4. Shut down the new database and then perform the validity checks that you would usually perform after restoring a database.

   For more information about validity checking, see "Validate a database" [*SQL Anywhere Server - Database Administration*].
5. Start the database using any production options you need. You can now allow user access to the reloaded database.

Note
There are additional options available for the dbunload utility that allow you to tune the unload and connection parameter options that allow you to specify a running or non-running database and database parameters. See "Unload utility (dbunload)" [*SQL Anywhere Server - Database Administration*].

To rebuild a database involved in synchronization or replication, with manual intervention

1. Shut down the database.
2. Perform a full off-line backup by copying the database and transaction log files to a secure location.
3. Run the dbtran utility to display the starting offset and ending offset of the database's current transaction log file.

   Note the ending offset for use in Step 8.

4. Rename the current transaction log file so that it is not modified during the unload process, and place this file in the dbremote off-line logs directory.

5. Rebuild the database.

   For information about this step, see "Rebuilding databases" on page 731.

6. Shut down the new database.

7. Erase the current transaction log file for the new database.

8. Use dblog on the new database with the ending offset noted in Step 3 as the -z parameter, and also set the relative offset to zero.

   ```
   dblog -x 0 -z 0000698242 -ir -is database-name.db
   ```

9. When you run the Message Agent, provide it with the location of the original off-line directory on its command line.

10. Start the database. You can now allow user access to the reloaded database.

Using the dbunload utility to rebuild databases

The dbunload and dbisql utilities let you unload an entire database in text comma-delimited format and create the necessary Interactive SQL command files to completely recreate your database. This may be useful for creating SQL Remote extractions or building new copies of your database with the same or a slightly modified structure. This utility is useful for exporting SQL Anywhere files intended for reuse within SQL Anywhere.

Note
The dbunload utility and the **Unload Database Wizard** are functionally equivalent. You can use them interchangeably to produce the same results.

Use the dbunload utility when you:

- want to rebuild your database or extract data from your database
- want to export in text format
- need to process large amounts of data quickly
- have flexible file format requirements

For more information, see:

- "Rebuild databases not involved in synchronization or replication" on page 734
- "Rebuild databases involved in synchronization or replication" on page 734

Using the UNLOAD TABLE statement to rebuild databases

The UNLOAD TABLE statement lets you export data efficiently in a specific character encoding. Consider using the UNLOAD TABLE statement to rebuild databases when you want to export data in text format.

Impact on the database

The UNLOAD TABLE statement places an exclusive lock on the entire table.

For more information, see "UNLOAD statement" [*SQL Anywhere Server - SQL Reference*].

Export table data or table schema

The Unload utility has options that allow you to unload only table data or the table schema.

The dbunload commands in these examples unload the data or schema from the SQL Anywhere sample database table (assumed to be running on the default database server with the default database name) into a file in the *c:\DataFiles* directory on the server computer. The statements required to recreate the schema and reload the specified tables are written to *reload.sql* in the local current directory.

To export table data (command line)

- Run the dbunload command, specifying: connection parameters using the -c option, table(s) you want to export data for using the -t option, and whether you want to unload only data by specifying the -d option.

  For example, to export the data from the Employees table, run the following command:

  ```
  dbunload -c "DBN=demo;UID=DBA;PWD=sql" -d -t Employees c:\DataFiles
  ```

  You can unload more than one table by separating the table names with a comma delimiter.

To export table schema (command line)

- Run the dbunload command, specifying: connection parameters using the -c option, the table(s) you want to export data for using the -t option, and whether you want to unload only the schema by specifying the -n option.

  For example, to export the schema for the Employees table, execute the following command:

  ```
  dbunload -c "DBN=demo;UID=DBA;PWD=sql" -n -t Employees
  ```

  You can unload more than one table by separating the table names with a comma delimiter.

Reload a database

Reloading involves creating an empty database file and using the *reload.sql* file to create the schema and insert all the data unloaded from another SQL Anywhere database into the newly created tables. You reload databases from the command line.

To reload a database (command line)

1. Run the dbinit utility to create a new empty database file.

   For example, the following command creates a file named *mynewdemo.db*.

   ```
   dbinit mynewdemo.db
   ```

2. Execute the *reload.sql* script.

   For example, the following command loads and runs the *reload.sql* script in the current directory.

   ```
   dbisql -c "DBF=mynewdemo.db;UID=DBA;PWD=sql" reload.sql
   ```

Minimize downtime when rebuilding a database

It is recommended that you make backup copies of your database files before rebuilding a database. Also, verify that no other scheduled backups can rename the production database log. If the log is renamed, the transactions from the renamed logs must be applied to the rebuilt database in the correct order.

To minimize the downtime during a rebuild

1. Using dbbackup -r, create a backup of the database and log, and rename the log.

   For more information, see "Backup utility (dbbackup)" [*SQL Anywhere Server - Database Administration*].

2. Rebuild the backed up database on another computer.

3. Perform another dbbackup -r on the production server to rename the transaction log.

4. Run dbtran on the transaction log and apply the transactions to the rebuilt server.

   For more information, see "Log Translation utility (dbtran)" [*SQL Anywhere Server - Database Administration*].

5. Shut down the production server and copy the database and log.

6. Copy the rebuilt database onto the production server.

7. Run dbtran on the log from Step 5.

8. Start the server on the rebuilt database, but do not allow users to connect.

9. Apply the transactions from Step 8.

10. Allow users to connect.

Extracting databases

Database extraction is used by SQL Remote. Extracting creates a remote SQL Anywhere database from a consolidated SQL Anywhere database.

You can use the Sybase Central **Extract Database Wizard** or the Extraction utility to extract databases. The Extraction utility (dbxtract) is the recommended way of creating remote databases from a consolidated database for use in SQL Remote replication.

For more information about how to perform database extractions, see:

- "Extraction utility (dbxtract)" [*SQL Remote*]
- "Extracting remote databases" [*SQL Remote*]
- "Deploying remote databases" [*MobiLink - Client Administration*]

Migrating databases to SQL Anywhere

Use the sa_migrate system procedures or the **Migrate Database Wizard**, to import tables from the following sources:

- SQL Anywhere
- UltraLite
- Sybase Adaptive Server Enterprise
- IBM DB2
- Microsoft SQL Server
- Microsoft Access
- Oracle
- MySQL
- Advantage Database Server
- generic ODBC driver that connects to a remote server

Before you can migrate data using the **Migrate Database Wizard**, or the sa_migrate set of system procedures, you must first create a **target database**. The target database is the database into which data is migrated.

For information about creating a database, see "Creating a SQL Anywhere database" [*SQL Anywhere Server - Database Administration*].

Use the Migrate Database Wizard

You can create a remote server to connect to the remote database, and an external login (if required) to connect the current user to the remote database using the **Migrate Database Wizard**.

To import remote tables (Sybase Central)

1. Use the SQL Anywhere 12 plug-in to connect to the database as a user with DBA authority.
2. Choose **Tools** » **SQL Anywhere 12** » **Migrate Database**.
3. Click **Next**.
4. Select the target database, and click **Next**.
5. Select the remote server you want to use to connect to the remote database, and then click **Next**.

   If you have not created a remote server, click **Create Remote Server Now** and follow the instructions in the **Create Remote Server Wizard**. For more information about creating a remote server, see "Create remote servers using the CREATE SERVER statement" on page 748.

   You can also create an external login for the remote server. By default, SQL Anywhere uses the user ID and password of the current user when it connects to a remote server on behalf of that user. However, if the remote server does not have a user defined with the same user ID and password as the current user, you must create an external login. The external login assigns an alternate login name and password for the current user so that user can connect to the remote server.
6. Select the tables that you want to migrate, and then click **Next**.

   You cannot migrate system tables, so no system tables appear in this list.
7. Select the user that will own the tables on the target database, and then click **Next**.

   If you have not created a user, click **Create User Now** and follow the instructions in the **Create User Wizard**. For more information, see "Creating new users" [*SQL Anywhere Server - Database Administration*].
8. Select whether you want to migrate the data and/or the foreign keys from the remote tables and whether you want to keep the proxy tables that are created for the migration process, and then click **Next**.
9. Click **Finish**.

Use the sa_migrate system procedures

Use the sa_migrate system procedures to migrate remote data. Use the extended method if you want to remove tables or foreign key mappings.

Migrating all tables using the sa_migrate system procedures

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. For these reasons, you should migrate tables associated with one owner at a time.

Migrating all tables for a remote user

1. Create a target database. See "Creating a SQL Anywhere database" [*SQL Anywhere Server - Database Administration*].

2. From Interactive SQL, connect to the target database.

3. Create a remote server to connect to the remote database. See "Create remote servers using the CREATE SERVER statement" on page 748.

4. Create an external login to connect to the remote database. This is only required when the user has different passwords on the target and remote databases, or when you want to login using a different user ID on the remote database than the one you are using on the target database. See "Create external logins" on page 757.

5. Create a local user who will own the migrated tables in the target database. See "Creating new users" [*SQL Anywhere Server - Database Administration*].

6. In the **SQL Statements** pane, run the sa_migrate system procedure. For example,

   ```
   CALL sa_migrate( 'local_user1', 'rmt_server1', NULL, 'remote_user1', NULL,
   1, 1, 1 );
   ```

   This procedure calls several procedures in turn and migrates all the remote tables belonging to the user remote_user1 using the specified criteria.

If you do not want all the migrated tables to be owned by the same user on the target database, you must run the sa_migrate procedure for each owner on the target database, specifying the *local-table-owner* and *owner-name* arguments.

For more information, see "sa_migrate system procedure" [*SQL Anywhere Server - SQL Reference*].

Migrating individual tables using the sa_migrate system procedures

Do not supply NULL for both the *table-name* and *owner-name* parameters. Doing so 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.

To import remote tables (with modifications)

1. Create a target database. See "Creating a SQL Anywhere database" [*SQL Anywhere Server - Database Administration*].

2. From Interactive SQL, connect to the target database.

3. Create a remote server to connect to the remote database. See "Create remote servers using the CREATE SERVER statement" on page 748.

4. Create an external login to connect to the remote database. This is only required when the user has different passwords on the target and remote databases, or when you want to login using a different

user ID on the remote database than the one you are using on the target database. See "Create external logins" on page 757.

5. Create a local user who will own the migrated tables in the target database. See "Creating new users" [*SQL Anywhere Server - Database Administration*].

6. Run the sa_migrate_create_remote_table_list system procedure. For example,

   ```
   CALL sa_migrate_create_remote_table_list( 'rmt_server1',
        NULL, 'remote_user1', 'mydb' );
   ```

   You must specify a database name for Adaptive Server Enterprise and Microsoft SQL Server databases.

   This populates the dbo.migrate_remote_table_list table with a list of remote tables to migrate. You can delete rows from this table for remote tables that you do not want to migrate.

   For more information, see "sa_migrate_create_remote_table_list system procedure" [*SQL Anywhere Server - SQL Reference*].

7. Run the sa_migrate_create_tables system procedure. For example:

   ```
   CALL sa_migrate_create_tables( 'local_user1' );
   ```

   This procedure takes the list of remote tables from dbo.migrate_remote_table_list and creates a proxy table and a base table for each remote table listed. This procedure also creates all primary key indexes for the migrated tables.

   For more information, see "sa_migrate_create_tables system procedure" [*SQL Anywhere Server - SQL Reference*].

8. If you want to migrate the data from the remote tables into the base tables on the target database, run the sa_migrate_data system procedure. For example,

   Execute the following statement:

   ```
   CALL sa_migrate_data( 'local_user1' );
   ```

   This procedure migrates the data from each remote table into the base table created by the sa_migrate_create_tables procedure.

   For more information, see "sa_migrate_data system procedure" [*SQL Anywhere Server - SQL Reference*].

   If you do not want to migrate the foreign keys from the remote database, you can skip to step 10.

9. Run the sa_migrate_create_remote_fks_list system procedure. For example,

   ```
   CALL sa_migrate_create_remote_fks_list( 'rmt_server1' );
   ```

   This procedure populates the table dbo.migrate_remote_fks_list with the list of foreign keys associated with each of the remote tables listed in dbo.migrate_remote_table_list.

   You can remove any foreign key mappings you do not want to recreate on the local base tables.

For more information, see "sa_migrate_create_remote_fks_list system procedure" [*SQL Anywhere Server - SQL Reference*].

10. Run the sa_migrate_create_fks system procedure. For example,

    ```
    CALL sa_migrate_create_fks( 'local_user1' );
    ```

    This procedure creates the foreign key mappings defined in dbo.migrate_remote_fks_list on the base tables.

    For more information, see "sa_migrate_create_fks system procedure" [*SQL Anywhere Server - SQL Reference*].

11. If you want to drop the proxy tables that were created for migration purposes, run the sa_migrate_drop_proxy_tables system procedure. For example,

    ```
    CALL sa_migrate_drop_proxy_tables( 'local_user1' );
    ```

    This procedure drops all proxy tables created for migration purposes and completes the migration process.

    For more information, see "sa_migrate_drop_proxy_tables system procedure" [*SQL Anywhere Server - SQL Reference*].

Using SQL command files

This section describes how to process files consisting of a set of commands. **Command files** are text files that contain SQL statements, and are useful if you want to run the same SQL statements repeatedly. Command files are also commonly called **scripts**. 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 database.

Creating command files

You can use any text editor that you like to create command files. You can include comment lines along with the SQL statements to be executed. See "Comments" [*SQL Anywhere Server - SQL Reference*].

Run SQL command files in Interactive SQL

You can execute command files in any of the following ways:

- You can run a command file without loading it into the **SQL Statements** pane.

  #### To run a command file immediately

  1. In Interactive SQL, choose **File** » **Run Script**.
  2. Locate the file, and click **Open**.

The contents of the specified file are run immediately. A **Status** window appears to show the execution progress.

The **Run Script** menu item is the equivalent of a READ statement. See below for an example of the READ statement.

- You can also run a command file without loading it into the **SQL Statements** pane with the Interactive SQL READ statement.

  To run a command file using the Interactive SQL READ statement

  - In the **SQL Statements** pane, type the following command:

    ```
    READ 'c:\\filename.sql';
    ```

    In this statement, *c:\filename.sql* is the path, name, and extension of the file. Single quotation marks (as shown) are required only if the path contains spaces.

    For more information, see "READ statement [Interactive SQL]" [*SQL Anywhere Server - SQL Reference*].

- You can supply a command file as a command line argument for Interactive SQL.

  To run a command file in batch mode (command line)

  - Run the dbisql utility and supply a command file as a command line argument.

    For example, the following command runs the command file *myscript.sql* against the SQL Anywhere sample database.

    ```
    dbisql -c "DSN=SQL Anywhere 12 Demo" myscript.sql
    ```

- You can load a command file into the **SQL Statements** pane and execute it directly from there.

  To load commands from a file into the SQL Statements pane

  1. Choose **File** » **Open**.
  2. Locate the file, and click **Open**.

     The commands are displayed in the **SQL Statements** pane where you read, edit, or execute them.

  On Windows platforms you can make Interactive SQL the default editor for *.sql* command files. This lets you double-click the file so that its contents appears in the **SQL Statements** pane of Interactive SQL. See "Setting Interactive SQL as the default editor for .sql files" [*SQL Anywhere Server - Database Administration*].

- You can also a load command file into the **SQL Statements** pane from your favorites.

  See "Using favorites" [*SQL Anywhere Server - Database Administration*].

Writing database output to a file

In Interactive SQL, the result set data for each command remains on the **Results** tab in the **Results** pane only until the next command is executed. To keep a record of your data, you can save the output of each statement to a separate file. If *statement1* and *statement2* are two SELECT statements, then you can output them to *file1* and *file2*, respectively, as follows:

statement1; OUTPUT TO *file1*;
statement2; OUTPUT TO *file2*;

For example, the following command saves the result of a query to a file named *Employees.txt*:

```
SELECT * FROM Employees;
OUTPUT TO 'C:\\My Documents\\Employees.txt';
```

See also

- "OUTPUT statement [Interactive SQL]" [*SQL Anywhere Server - SQL Reference*]
- "Export data with the UNLOAD statement" on page 721

Adaptive Server Enterprise compatibility

You can import and export files between SQL Anywhere and Adaptive Server Enterprise using the BCP FORMAT clause. If you are exporting BLOB data from SQL Anywhere for use in Adaptive Server Enterprise, use the BCP format clause with the UNLOAD TABLE statement.

When using the BCP out command to export files from Adaptive Server Enterprise so that you can import the data into SQL Anywhere, the data must be in text/ASCII format, and it must be comma delimited. You can use the -c option for the BCP out command to export the data in text/ASCII format. The -t option lets you change the delimiter, which is a tab by default. If you do not change the delimiter, then you must specify **DELIMITED BY '\x09'** in the LOAD TABLE statement when you import the data into your SQL Anywhere database.

For more information about BCP and the FORMAT clause, see "LOAD TABLE statement" [*SQL Anywhere Server - SQL Reference*], or "UNLOAD statement" [*SQL Anywhere Server - SQL Reference*].

Accessing remote data

SQL Anywhere remote data access gives you access to data in other data sources. You can use this feature to migrate data into a SQL Anywhere database. You can also use the feature to query data across databases.

With remote data access you can:

- Use SQL Anywhere to move data from one location to another using insert-select.
- Access data in relational databases such as Sybase, Oracle, and IBM DB2.

- Access desktop data such as Excel spreadsheets, Microsoft Access databases, FoxPro, and text files.
- Access any other data source that supports an ODBC interface.
- Perform joins between local and remote data, although performance is much slower than if all the data is in a single SQL Anywhere database.
- Perform joins between tables in separate SQL Anywhere databases. Performance limitations here are the same as with other remote data sources.
- Use SQL Anywhere features on data sources that would normally not have that ability. For instance, you could use a Java function against data stored in Oracle, or perform a subquery on spreadsheets. SQL Anywhere compensates for features not supported by a remote data source by operating on the data after it is retrieved.
- Access remote servers directly using passthrough mode.
- Execute remote procedure calls to other servers.

SQL Anywhere allows access to the following external data sources:

- SQL Anywhere
- Adaptive Server Enterprise
- Advantage Database Server
- IBM DB2
- Microsoft Access
- Microsoft SQL Server
- MySQL
- Oracle
- Sybase IQ
- UltraLite
- Other ODBC data sources

Note
You cannot create a remove server for an UltraLite database running on Mac OS X.

For platform availability, see http://www.sybase.com/detail?id=1002288.

Remote table mappings

SQL Anywhere presents tables to a client application as if all the data in the tables were stored in the database to which the application is connected. Internally, when a query involving remote tables is executed, the storage location is determined, and the remote location is accessed so that data can be retrieved.

To have remote tables appear as local tables to the client, you create local proxy tables that map to the remote data.

To create a proxy table

1. Define the server where the remote data is located. This specifies the type of server and location of the remote server. See "Working with remote servers" on page 748.

2. Map the local user login information to the remote server user login information if the logins on the two servers are different. See "Working with external logins" on page 757.

3. Create the proxy table definition. This specifies the mapping of a local proxy table to the remote table. This includes the server where the remote table is located, the database name, owner name, table name, and column names of the remote table.

For more information, see "Working with proxy tables" on page 758.

Administering remote table mappings

To manage remote table mappings and remote server definitions, you can use Sybase Central or you can use a tool such as Interactive SQL to execute the SQL statements.

Caution
Some remote servers, such as Microsoft Access, Microsoft SQL Server, and Sybase Adaptive Server Enterprise do not preserve cursors across COMMITs and ROLLBACKS. With these remote servers, you cannot use the Data tab in the SQL Anywhere 12 plug-in to view or modify the contents of a proxy table. However, you can still use Interactive SQL to view and edit the data in these proxy tables as long as autocommit is turned off (this is the default behavior in Interactive SQL). Other RDBMSs, including Oracle, IBM DB2, and SQL Anywhere do not have this limitation.

Server classes

A **server class** specifies the access method used to interact with the server. A server class is assigned to each remote server. Different types of remote servers require different access methods. The server class provides SQL Anywhere detailed server capability information. SQL Anywhere adjusts its interaction with the remote server based on those capabilities.

The ODBC-based server classes are:

- **saodbc** for SQL Anywhere.
- **ulodbc** for UltraLite.

  Note
  You cannot create a remove server for an UltraLite database running on Mac OS X.

- **adsodbc** for Advantage Database Server.
- **aseodbc** for Sybase SQL Server and Adaptive Server Enterprise (version 10 and later).

- **db2odbc** for IBM DB2.
- **iqodbc** for Sybase IQ.
- **msaccessodbc** for Microsoft Access.
- **mssodbc** for Microsoft SQL Server.
- **mysqlodbc** for MySQL.
- **odbc** for all other ODBC data sources.
- **oraodbc** for Oracle servers (version 8.0 and later).

Note
When using remote data access, if you use an ODBC driver that does not support Unicode, then character set conversion is not performed on data coming from that ODBC driver.

For a full description of remote server classes, see "Server classes for remote data access" on page 773.

Accessing remote data from PowerBuilder DataWindows

Set the DBParm Block parameter to 1 on connect to access remote data from a PowerBuilder DataWindow.

- In the design environment, you can set the Block parameter by accessing the **Transaction** tab in the **Database Profile Setup** window and setting the Retrieve Blocking Factor to 1.
- In a connection string, use the following parameter:

```
DBParm="Block=1"
```

Working with remote servers

Before you can map remote objects to a local proxy table, you must define the remote server where the remote object is located. When you define a remote server, an entry is added to the ISYSSERVER system table for the remote server.

Create remote servers using the CREATE SERVER statement

Use the CREATE SERVER statement to set up remote server definitions. To use Sybase Central to create remote server definitions, see "Create remote servers" on page 750.

For ODBC connections, each remote server corresponds to an ODBC data source. For some systems, including SQL Anywhere, each data source describes a database, so a separate remote server definition is needed for each database.

You must have RESOURCE authority to create a remote server.

On Unix platforms, you need to reference the ODBC driver manager as well.

For a full description of the CREATE SERVER statement, see "CREATE SERVER statement" [*SQL Anywhere Server - SQL Reference*].

Example 1

The following statement creates an entry in the ISYSSERVER system table for the Adaptive Server Enterprise server called RemoteASE:

```
CREATE SERVER RemoteASE
CLASS 'ASEJDBC'
USING 'rimu:6666';
```

- **RemoteASE** is the name of the remote server.
- **ASEJDBC** is a keyword indicating that the remote server is Adaptive Server Enterprise and the connection to it is JDBC-based.
- **rimu:6666** is the computer name and the TCP/IP port number where the remote server is located.

Example 2

The following statement creates an entry in the ISYSSERVER system table for the ODBC-based SQL Anywhere server named RemoteSA:

```
CREATE SERVER RemoteSA
CLASS 'SAODBC'
USING 'test4';
```

- **RemoteSA** is the name by which the remote server is known within this database.
- **SAODBC** is a keyword indicating that the server is SQL Anywhere and the connection to it uses ODBC.
- **test4** is the ODBC Data Source Name (DSN).

Example 3

On Unix platforms, the following statement creates an entry in the ISYSSERVER system table for the ODBC-based SQL Anywhere server named RemoteSA:

```
CREATE SERVER RemoteSA
CLASS 'SAODBC'
USING 'driver=SQL Anywhere 12;dsn=my_sa_dsn';
```

- **RemoteSA** is the name by which the remote server is known within this database.
- **SAODBC** is a keyword indicating that the server is SQL Anywhere and the connection to it uses ODBC.
- **USING** is the reference to the ODBC driver manager.

Example 4

On Unix platforms the following statement creates an entry in the ISYSSERVER system table for the ODBC-based Adaptive Server Enterprise server named RemoteASE:

```
CREATE SERVER RemoteASE
CLASS 'ASEODBC'
USING '/opt/sybase/ase_odbc_1500/DataAccess/ODBC/lib/
libsybdrvodb.so;dsn=my_ase_dsn';
```

- **RemoteASE** is the name by which the remote server is known within this database.
- **ASEODBC** is a keyword indicating that the server is Adaptive Server Enterprise and the connection to it uses ODBC.
- **USING** is the reference to the ODBC driver manager.

Create remote servers

To create a remote server (Sybase Central)

1. Use the SQL Anywhere 12 plug-in to connect to the host database as a user with DBA authority.
2. In the left pane, double-click **Remote Servers**.
3. Click **File** » **New** » **Remote Server**.
4. In the **What Do You Want To Name The New Remote Server** field, type a name for the remote server, and then click **Next**.
5. Select a remote server type, and then click **Next**.
6. Select a connection type, and in the **What Is The Connection Information** field type the connection information:
   - For ODBC, supply a data source name or specify the ODBC Driver= parameter.
   - For JDBC, supply a URL in the form *computer-name*:*port-number*.

   The data access method (JDBC or ODBC) is the method used by SQL Anywhere to access the remote database. This is not related to the method used by Sybase Central to connect to your database.
7. Click **Next**.
8. Specify whether you want the remote server to be read-only and then click **Next**.
9. Click **Create An External Login For The Current User** and complete the required fields.

   By default, SQL Anywhere uses the user ID and password of the current user when it connects to a remote server on behalf of that user. However, if the remote server does not have a user defined with the same user ID and password as the current user, you must create an external login. The external

login assigns an alternate login name and password for the current user so that user can connect to the remote server. See "CREATE EXTERNLOGIN statement" [*SQL Anywhere Server - SQL Reference*].

10. Click **Test Connection** to test the remote server connection.

11. Click **Finish**.

Delete remote servers

You can use Sybase Central or a DROP SERVER statement to delete a remote server from the ISYSSERVER system table. All remote tables defined on that server must already be dropped for this action to succeed.

To delete a remote server (Sybase Central)

1. Use the SQL Anywhere 12 plug-in to connect to the host database as a user with DBA authority.
2. In the left pane, double-click **Remote Servers**.
3. Select the remote server, and then click **File** » **Delete**.

To delete a remote server (SQL)

1. Connect to the host database as a user with DBA authority.
2. Execute a DROP SERVER statement.

For more information, see "DROP SERVER statement" [*SQL Anywhere Server - SQL Reference*].

Example

The following statement drops the server named RemoteSA:

```
DROP SERVER RemoteSA;
```

Alter remote servers

Changes to the remote server do not take effect until the next connection to the remote server.

To alter the properties of a remote server (Sybase Central)

1. Use the SQL Anywhere 12 plug-in to connect to the host database as a user with RESOURCE authority.
2. In the left pane, double-click **Remote Servers**.
3. Select the remote server, and then choose **File** » **Properties**.
4. Alter the remote server settings, and then click **OK**.

To alter the properties of a remote server (SQL)

1. Connect to the host database as a user with DBA authority.
2. Execute an ALTER SERVER statement.

Example

The following statement changes the server class of the server named RemoteASE to aseodbc. In this example, the Data Source Name for the server is RemoteASE.

```
ALTER SERVER RemoteASE
CLASS 'aseodbc';
```

The ALTER SERVER statement can also be used to enable or disable a server's known capabilities. See "ALTER SERVER statement" [*SQL Anywhere Server - SQL Reference*].

List the remote tables on a server

When configuring SQL Anywhere to get a list of the remote tables available on a particular server, it may be helpful to use the sp_remote_tables system procedure. The sp_remote_tables procedure returns a list of the tables on a remote server.

```
sp_remote_tables(
@server-name
[, @table-name
[, @table-owner
[, @table-qualifier
[, @with-table-type ] ] ] ]
)
```

If you specify *table-name* or *table-owner*, the list of tables is limited to only those that match.

For example, to get a list of all the Microsoft Excel worksheets available from a remote server named excel:

```
CALL sp_remote_tables  excel;
```

Or to get a list of all the tables in the production database in an Adaptive Server Enterprise server named asetest, owned by fred:

```
CALL sp_remote_tables  asetest, null, fred, production;
```

For more information, see "sp_remote_tables system procedure" [*SQL Anywhere Server - SQL Reference*].

List remote server capabilities

The sp_servercaps system 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 passed to a remote server.

You can also view capability information for remote servers by querying the SYSCAPABILITY and SYSCAPABILITYNAME system views. These system views are empty until after SQL Anywhere first connects to a remote server.

When using the sp_servercaps system procedure, the *server-name* specified must be the same *server-name* used in the CREATE SERVER statement.

Execute the stored procedure sp_servercaps as follows:

```
CALL sp_servercaps  server-name;
```

See also

- "sp_servercaps system procedure" [*SQL Anywhere Server - SQL Reference*]
- "SYSCAPABILITY system view" [*SQL Anywhere Server - SQL Reference*]
- "SYSCAPABILITYNAME system view" [*SQL Anywhere Server - SQL Reference*]
- "CREATE SERVER statement" [*SQL Anywhere Server - SQL Reference*]

Using directory access servers

A **directory access server** is a remote server that gives you access to the local file structure of the computer running the database server. Once you are connected to the directory access server, you use proxy tables to access any subdirectories on the computer. Database users must have an external login to use the directory access server.

You cannot alter a directory access server after it is created. If you need to change a directory access server, you must drop it and recreate it with different settings. You must first drop any proxy tables that reference the directory access server and then recreate them after recreating the directory access server.

The following describes the format of the proxy table.

- **permissions VARCHAR(10)** A Posix-style permission string such as "drwxrwxrwx".
- **size BIGINT** The size of the file in bytes.
- **access_date_time TIMESTAMP** The date and time the file was last accessed (for example, 2010-02-08 11:00:24.000).
- **modified_date_time TIMESTAMP** The date and time the file was last modified (for example, 2009-07-28 10:50:11.000).
- **create_date_time TIMESTAMP** The date and time the file was created (for example, 2008-12-18 10:32:26.000).
- **owner VARCHAR(20)** The user ID of the file's creator (for example, "root" on Linux). For Windows, this value is always "0".
- **file_name VARCHAR(260)** The name of the file, including a relative path (for example, bin\perl.exe).

- **contents LONG VARBINARY** The contents of the file when this column is explicitly referenced in the result set.

Note
Executing a `SELECT *` statement from a proxy table that is mapped to a directory access server returns an empty string in the contents column. The database server behaves this way because in most cases the intention is not to retrieve the contents of each file in the directory structure and the files could potentially be large.

However, if the contents column is explicitly specified either in the select list, the WHERE clause, or any other portion of the query, then the file contents are fetched.

It is recommended that an application performs a `SELECT *` statement to retrieve the list of files and information about each file (such as the size or modified time). The application can then execute a `SELECT contents WHERE filename=...` to fetch the contents of a particular file.

Create directory access servers

You use the CREATE SERVER statement or **Create Directory Access Server Wizard** in Sybase Central to create a directory access server.

When you create a directory access server, you can control the number of subdirectories that can be accessed and whether the directory access server can be used to modify existing files.

The following steps are required to set up a directory access server:

1. Create a remote server for the directory (requires DBA authority).
2. Create external logins for the database users who can use the directory access server (requires DBA authority).
3. Create proxy tables to access the directories on the computer (requires RESOURCE authority).

To create and configure a directory access server (Sybase Central)

1. Use the SQL Anywhere 12 plug-in to connect to the host database as a user with DBA authority.
2. In the left pane, double-click **Directory Access Servers**.
3. Click **File** » **New** » **Directory Access Server**.
4. Follow the instructions in the **Create Directory Access Server Wizard**.

To create and configure a directory access server (SQL)

1. Connect to the host database as a user with DBA authority.
2. Create a remote server using the CREATE SERVER statement.

For example:

```
CREATE SERVER my_dir_tree
CLASS 'directory'
USING 'root=c:\Program Files';
```

3. Create an external login using the CREATE EXTERNLOGIN statement.

   For example:

   ```
   CREATE EXTERNLOGIN DBA TO my_dir_tree;
   ```

4. Create a proxy table for the directory using the CREATE EXISTING TABLE statement.

   For example:

   ```
   CREATE EXISTING TABLE my_program_files AT 'my_dir_tree;;;.';
   ```

   In this example, my_program_files is the name of the directory, and my_dir_tree is the name of the directory access server.

Example

The following statements create a new directory access server named directoryserver3 that can be used to access up to three levels of subdirectories, create an external login to the directory access server for the DBA user, and create a proxy table named diskdir3.

```
CREATE SERVER directoryserver3
CLASS 'DIRECTORY'
USING 'ROOT=c:\mydir;SUBDIRS=3';
CREATE EXTERNLOGIN DBA TO directoryserver3;
CREATE EXISTING TABLE diskdir3 AT 'directoryserver3;;;.';
```

Using the sp_remote_tables system procedure, you can see all the subdirectories located in *c:\mydir* on the computer running the database server:

```
CALL sp_remote_tables( 'directoryserver3' );
```

Using the following SELECT statement, you can view the contents of the file *c:\mydir\myfile.txt*:

```
SELECT contents
FROM diskdir3
WHERE file_name = 'myfile.txt';
```

Alternatively, you can select data from the directories:

```
-- Get the list of directories in this disk directory tree.
SELECT permissions, file_name, size
FROM diskdir3
WHERE PERMISSIONS LIKE 'd%';
-- Get the list of files.
SELECT permissions, file_name, size
FROM diskdir3
WHERE PERMISSIONS NOT LIKE 'd%';
```

See also

- "CREATE SERVER statement" [*SQL Anywhere Server - SQL Reference*]
- "CREATE EXTERNLOGIN statement" [*SQL Anywhere Server - SQL Reference*]
- "CREATE TABLE statement" [*SQL Anywhere Server - SQL Reference*]
- "CREATE EXISTING TABLE statement" [*SQL Anywhere Server - SQL Reference*]

Drop directory access servers

You cannot alter an existing directory access server: you must drop the existing directory access server using a DROP SERVER statement, and then create a new one.

Drop a directory access server (Sybase Central)

1. Use the SQL Anywhere 12 plug-in to connect to the host database as a user with DBA authority.
2. In the left pane, double-click **Directory Access Servers**.
3. Select the directory access server, and then choose **Edit** » **Delete**.

Drop a directory access server (SQL)

1. Connect to the host database as a user with DBA authority.
2. Execute a DROP SERVER statement.

   For example:

   ```
   DROP SERVER my_directory_server;
   ```

Use the DROP TABLE statement to drop a proxy table used by the directory access server.

Drop a proxy table (Sybase Central)

1. Use the SQL Anywhere 12 plug-in to connect to the host database as a user with DBA authority.
2. In the left pane, double-click **Directory Access Servers**.
3. In the right pane, click the **Proxy Tables** tab.
4. Select the proxy table, and then choose **Edit** » **Delete**.
5. Click **Yes**.

Drop a proxy table (SQL)

1. Connect to the host database as a user with DBA authority.
2. Execute a DROP TABLE statement.

   For example:


```
DROP TABLE my_files;
```

See also

- "DROP SERVER statement" [*SQL Anywhere Server - SQL Reference*]
- "DROP TABLE statement" [*SQL Anywhere Server - SQL Reference*]

Working with external logins

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 to be used when communicating with a remote server.

For more information, see "Using Windows integrated logins" [*SQL Anywhere Server - Database Administration*].

Create external logins

Use one of the following procedures to create an external login.

To create an external login (Sybase Central)

1. Use the SQL Anywhere 12 plug-in to connect to the host database as a user with DBA authority or as the owner of the external login.
2. In the left pane, double-click **Remote Servers**.
3. Select the remote server, and in the right pane click the **External Logins** tab.
4. From the **File** menu, choose **New** » **External Login**.
5. Follow the instructions in the **Create External Login Wizard**.

To create an external login (SQL)

1. Connect to the host database as a user with DBA authority.
2. Execute a CREATE EXTERNLOGIN statement.

Example

The following statement allows the local user fred to gain access to the server RemoteASE, using the remote login frederick with password banana.

```
CREATE EXTERNLOGIN fred
TO RemoteASE
REMOTE LOGIN frederick
IDENTIFIED BY banana;
```

For more information, see "CREATE EXTERNLOGIN statement" [*SQL Anywhere Server - SQL Reference*].

Drop external logins

Use one of the following procedures to delete an external login from the SQL Anywhere system tables.

To delete an external login (Sybase Central)

1. Use the SQL Anywhere 12 plug-in to connect to the host database as a user with DBA authority or as the owner of the external login.
2. In the left pane, double-click **Remote Servers**.
3. Select the remote server, and in the right pane click the **External Logins** tab.
4. Select the external login, and then choose **File** » **Delete**.
5. Click **Yes**.

To delete an external login (SQL)

1. Connect to the host database as a user with DBA authority.
2. Execute a DROP EXTERNLOGIN statement.

Example

The following statement drops the external login for the local user fred created in the example above:

```
DROP EXTERNLOGIN fred TO RemoteASE;
```

See also

- "DROP EXTERNLOGIN statement" [*SQL Anywhere Server - SQL Reference*]

Working with proxy tables

Location transparency of remote data is enabled by creating a local **proxy table** that maps to the remote object. You can use a proxy table to access any object (including tables, views, and materialized views) that the remote database exports as a candidate for a proxy table. Use one of the following statements to create a proxy table:

- If the table already exists at the remote storage location, use the CREATE EXISTING TABLE statement. This statement defines the proxy table for an existing table on the remote server.
- If the table does not exist at the remote storage location, use the CREATE TABLE statement. This statement creates a new table on the remote server, and also defines the proxy table for that table.

Note
You cannot modify data in a proxy table when you are within a savepoint. See "Savepoints within transactions" on page 92.

When a trigger is fired on a proxy table, the permissions used are those of the user who caused the trigger to fire, not those of the proxy table owner.

Specify proxy table locations

The AT keyword is used with both CREATE TABLE and CREATE EXISTING TABLE to define the location of an existing object. This location string has four components, each separated by either a period or a semicolon. The semicolon delimiter allows file names and extensions to be used in the database and owner fields.

The syntax of the AT clause is

```
... AT 'server.database.owner.table-name'
```

- **server** This is the name by which the server is known in the current database, as specified in the CREATE SERVER statement. This field is mandatory for all remote data sources.
- **database** The meaning of the database field depends on the data source. Sometimes this field does not apply and should be left empty. The delimiter is still required, however.

  If the data source is Adaptive Server Enterprise, *database* specifies the database where the table exists. For example master or pubs2.

  If the data source is SQL Anywhere, this field does not apply; leave it empty.

  If the data source is Excel, Lotus Notes, or Access, you must include the name of the file containing the table. If the file name includes a period, use the semicolon delimiter.
- **owner** If the database supports the concept of ownership, this field represents the owner name. This field is only required when several owners have tables with the same name.
- **table-name** This field specifies the name of the table. For an Excel spreadsheet, this is the name of the sheet in the workbook. If *table-name* is left empty, the remote table name is assumed to be the same as the local proxy table name.

Examples

The following examples illustrate the use of location strings:

- SQL Anywhere:

  ```
  'RemoteSA..GROUPO.Employees'
  ```
- Adaptive Server Enterprise:

  ```
  'RemoteASE.pubs2.dbo.publishers'
  ```

- Excel:

  ```
  'excel;d:\pcdb\quarter3.xls;;sheet1$'
  ```

- Access:

  ```
  'access;\\server1\production\inventory.mdb;;parts'
  ```

Create proxy tables (Sybase Central)

Use one of the following procedures to create a proxy table. You cannot create proxy tables for system tables.

The CREATE EXISTING TABLE statement creates a proxy table that maps to an existing table on the remote server. SQL Anywhere derives the column attributes and index information from the object at the remote location.

For information about the CREATE EXISTING TABLE statement, see "CREATE EXISTING TABLE statement" [*SQL Anywhere Server - SQL Reference*].

To create a proxy table (Sybase Central)

1. Use the SQL Anywhere 12 plug-in to connect to the host database as a user with DBA authority.
2. In the left pane, double-click **Remote Servers**.
3. Select a remote server, and in the right pane click the **Proxy Tables** tab.
4. From the **File** menu choose **New** » **Proxy Table**.
5. Follow the instructions in the **Create Proxy Table Wizard**.

Create proxy tables with the CREATE EXISTING TABLE statement

The CREATE EXISTING TABLE statement creates a proxy table that maps to an existing table on the remote server. SQL Anywhere derives the column attributes and index information from the object at the remote location.

To create a proxy table with the CREATE EXISTING TABLE statement (SQL)

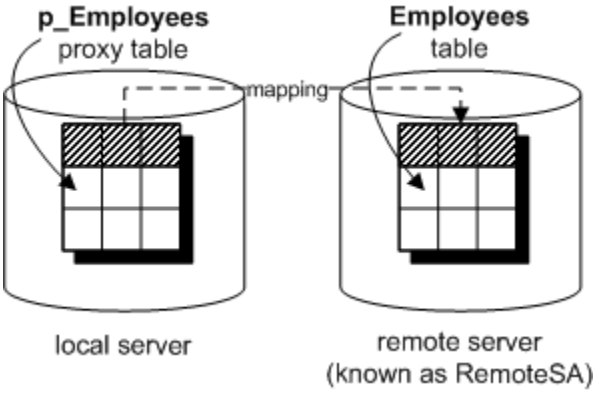
1. Connect to the host database as a user with DBA authority.
2. Execute a CREATE EXISTING TABLE statement.

For more information, see "CREATE EXISTING TABLE statement" [*SQL Anywhere Server - SQL Reference*].

Example 1

To create a proxy table called p_Employees on the current server to a remote table named Employees on the server named RemoteSA, use the following syntax:

```
CREATE EXISTING TABLE p_Employees
AT 'RemoteSA..GROUPO.Employees';
```



Example 2

The following statement maps the proxy table a1 to the Microsoft Access file *mydbfile.mdb*. In this example, the AT keyword uses the semicolon (;) as a delimiter. The server defined for Microsoft Access is named access.

```
CREATE EXISTING TABLE a1
AT 'access;d:\mydbfile.mdb;;a1';
```

Create a proxy table with the CREATE TABLE statement

The CREATE TABLE statement creates a new table on the remote server, and defines the proxy table for that table when you use the AT option. Columns are defined using SQL Anywhere data types. SQL Anywhere automatically converts the data into the remote server's native types.

If you use the CREATE TABLE statement to create both a local and remote table, and then subsequently use the DROP TABLE statement to drop the proxy table, the remote table is also dropped. You can, however, use the DROP TABLE statement to drop a proxy table created using the CREATE EXISTING TABLE statement. In this case, the remote table is not dropped.

For more information, see "CREATE TABLE statement" [*SQL Anywhere Server - SQL Reference*] and "CREATE EXISTING TABLE statement" [*SQL Anywhere Server - SQL Reference*].

Example

The following statement creates a table named Employees on the remote server RemoteSA, and creates a proxy table named Members that maps to the remote table:

```
CREATE TABLE Members
( membership_id INTEGER NOT NULL,
```

```
member_name CHAR( 30 ) NOT NULL,
office_held CHAR( 20 ) NULL )
AT 'RemoteSA..GROUPO.Employees';
```

List the columns on a remote table

Before you execute a CREATE EXISTING TABLE statement, it may be helpful to get a list of the columns that are available on a remote table. The sp_remote_columns system procedure produces a list of the columns on a remote table and a description of those data types. The following is the syntax for the sp_remote_columns system procedure:

```
sp_remote_columns servername, tablename [, owner ]
[, database]
```

If a table name, owner, or database name is given, the list of columns is limited to only those that match.

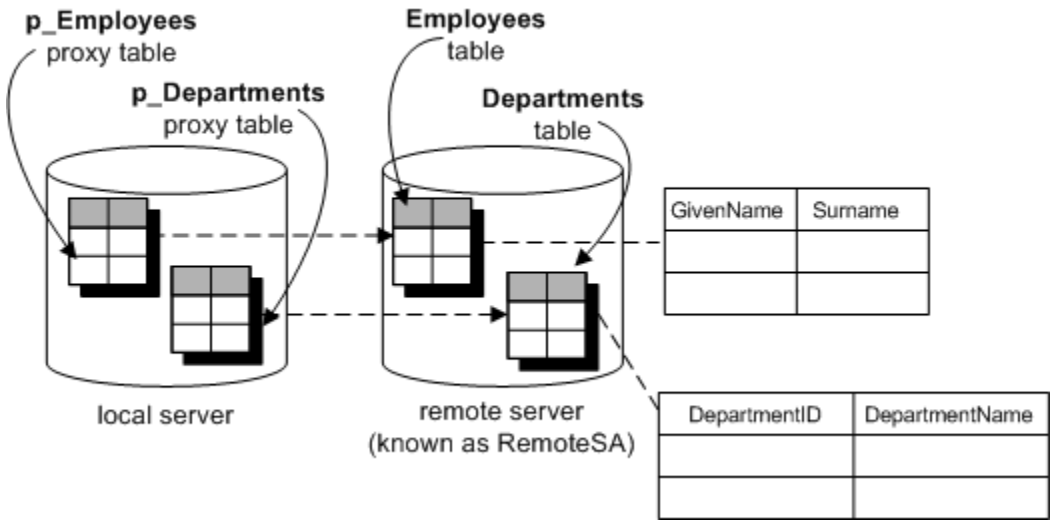
For example, the following returns a list of the columns in the sysobjects table in the production database on an Adaptive Server Enterprise server named asetest:

```
CALL sp_remote_columns  asetest, sysobjects, null, production;
```

For more information, see "sp_remote_columns system procedure" [*SQL Anywhere Server - SQL Reference*].

Join remote tables

The following figure illustrates proxy tables on a local database server mapped to the remote tables Employees and Departments of the SQL Anywhere sample database on the remote server RemoteSA mapped.



You can use joins between tables on different SQL Anywhere databases. The following example is a simple case using just one database to illustrate the principles.

To perform a join between two remote tables (SQL)

1. Create a new database named *empty.db*.

   This database holds no data. It is used only to define the remote objects, and to access the SQL Anywhere sample database.

2. Start a database server running the *empty.db*. You can do this using the following command line:

   ```
   dbsrv12 empty
   ```

3. From Interactive SQL, connect to *empty.db* as user DBA.

4. In the new database, create a remote server named RemoteSA. Its server class is saodbc, and the connection string refers to the SQL Anywhere 12 Demo ODBC data source:

   ```
   CREATE SERVER RemoteSA
   CLASS 'saodbc'
   USING 'SQL Anywhere 12 Demo';
   ```

5. In this example, you use the same user ID and password on the remote database as on the local database, so no external logins are needed.

   Sometimes you must provide a user ID and password when connecting to the database at the remote server. In the new database, you could create an external login to the remote server. For simplicity in our example, the local login name and the remote user ID are both DBA:

   ```
   CREATE EXTERNLOGIN DBA
   TO RemoteSA
   REMOTE LOGIN DBA
   IDENTIFIED BY sql;
   ```

6. Define the p_Employees proxy table:

   ```
   CREATE EXISTING TABLE p_Employees
   AT 'RemoteSA..GROUPO.Employees';
   ```

7. Define the p_Departments proxy table:

   ```
   CREATE EXISTING TABLE p_Departments
   AT 'RemoteSA..GROUPO.Departments';
   ```

8. Use the proxy tables in the SELECT statement to perform the join.

   ```
   SELECT GivenName, Surname, DepartmentName
   FROM p_Employees JOIN p_Departments
   ON p_Employees.DepartmentID = p_Departments.DepartmentID
   ORDER BY Surname;
   ```

Join tables from multiple local databases

A SQL Anywhere server may have several local databases running at one time. By defining tables in other local SQL Anywhere databases as remote tables, you can perform cross-database joins.

For more information about specifying multiple databases, see "USING parameter in the CREATE SERVER statement" on page 775.

Example

Suppose you are using database db1, and you want to access data in tables in database db2. You need to set up proxy table definitions that point to the tables in database db2. For example, on a SQL Anywhere server named RemoteSA, you might have three databases available, db1, db2, and db3.

1. If you are using ODBC, create an ODBC data source name for each database you will be accessing.
2. Connect to one of the databases from which you will be performing. For example, connect to db1.
3. Perform a CREATE SERVER statement for each other local database you will be accessing. This sets up a **loopback** connection to your SQL Anywhere server.

   ```
   CREATE SERVER remote_db2
   CLASS 'saodbc'
   USING 'RemoteSA_db2';
   CREATE SERVER remote_db3
   CLASS 'saodbc'
   USING 'RemoteSA_db3';
   ```

4. Create proxy table definitions by executing CREATE EXISTING TABLE statements for the tables in the other databases you want to access.

   ```
   CREATE EXISTING TABLE Employees
   AT 'remote_db2...Employees';
   ```

Send native statements to remote servers

Use the FORWARD TO statement to send one or more statements to the remote server in its native syntax. This statement can be used in two ways:

- To send a statement to a remote server.
- To place SQL Anywhere into passthrough mode for sending a series of statements to a remote server.

The FORWARD TO statement can be used to verify that a server is configured correctly. If you send a statement to the remote server and SQL Anywhere does not return an error message, the remote server is configured correctly.

The FORWARD TO statement cannot be used within procedures or batches.

If a connection cannot be made to the specified server, a message is returned to the user. If a connection is made, any results are converted into a form that can be recognized by the client program.

For more information, see "FORWARD TO statement" [*SQL Anywhere Server - SQL Reference*].

Example 1

The following statement verifies connectivity to the server named RemoteASE by selecting the version string:

```
FORWARD TO RemoteASE {SELECT @@version};
```

Example 2

The following statements show a passthrough session with the server named RemoteASE:

```
FORWARD TO RemoteASE
SELECT * FROM titles
SELECT * FROM authors
FORWARD TO;
```

Using remote procedure calls (RPCs)

SQL Anywhere users can issue procedure calls to remote servers that support the feature.

This functionality is supported by SQL Anywhere, Adaptive Server Enterprise, Oracle, and IBM DB2. Issuing a remote procedure call is similar to using a local procedure call.

SQL Anywhere supports fetching result sets from remote procedures, including fetching multiple result sets. As well, remote functions can be used to fetch return values from remote procedures and functions. Remote procedures can be used in the FROM clause of a SELECT statement.

Create remote procedures

Use one of the following procedures to issue a remote procedure call.

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 always return one, then the local procedure definition must contain a RESULT clause.

To create a remote procedure (Sybase Central)

1. Use the SQL Anywhere 12 plug-in to connect to the host database as a user with DBA authority.
2. In the left pane, double-click **Remote Servers**.
3. Select the remote server, and in the right pane click the **Remote Procedures** tab.
4. From the **File** menu, choose **New** » **Remote Procedure**.
5. Follow the instructions in the **Create Remote Procedure Wizard**.

To create a remote procedure (SQL)

1. Connect to the host database as a user with DBA authority.
2. Execute a statement to define the procedure to SQL Anywhere. For example:

   ```
   CREATE PROCEDURE RemoteWho()
   AT 'bostonase.master.dbo.sp_who';
   ```

   The syntax is similar to a local procedure definition. The location string defines the location of the procedure.

For more information, see "CREATE PROCEDURE statement" [*SQL Anywhere Server - SQL Reference*].

To issue a remote procedure call (SQL)

1. Connect to the database as a user with DBA authority.
2. Execute the procedure as follows:

   ```
   CALL RemoteWho();
   ```

Example

This example specifies a parameter when calling a remote procedure:

```
CREATE PROCEDURE RemoteUser ( IN username CHAR( 30 ) )
AT 'bostonase.master.dbo.sp_helpuser';
CALL RemoteUser( 'joe' );
```

Data types for remote procedures

The following data types are allowed for remote procedure call parameters:

- [UNSIGNED] SMALLINT
- [UNSIGNED] INT
- [UNSIGNED] BIGINT
- TINYINT
- REAL
- DOUBLE
- CHAR
- BIT
- NUMERIC and DECIMAL data types are allowed for IN parameters, but not for OUT or INOUT parameters

Drop remote procedures

Use one of the following procedures to delete a remote procedure.

To delete a remote procedure (Sybase Central)

1. Use the SQL Anywhere 12 plug-in to connect to the host database as a user with DBA authority.
2. In the left pane, double-click **Remote Servers**.
3. Select the remote server, and in the right pane click the **Remote Procedures** tab.
4. Select the remote procedure, and then choose **File** » **Delete**.
5. Click **Yes**.

To delete a remote procedure (SQL)

1. Connect to the host database as a user with DBA authority.
2. Execute a DROP PROCEDURE statement.

For more information, see "DROP PROCEDURE statement" [*SQL Anywhere Server - SQL Reference*].

Example

Delete a remote procedure called remoteproc.

```
DROP PROCEDURE remoteproc;
```

Transaction management and remote data

Transactions provide a way to group SQL statements so that they are treated as a unit—either all work performed by the statements is committed to the database, or none of it is.

For the most part, transaction management with remote tables is the same as transaction management for local tables in SQL Anywhere, but there are some differences. They are discussed in the following section.

For a general discussion of transactions, see "Using transactions and isolation levels" on page 89.

Remote transaction management overview

The method for managing transactions involving remote servers uses a two-phase commit protocol. SQL Anywhere implements a strategy that ensures transaction integrity for most scenarios. However, when more than one remote server is invoked in a transaction, there is still a chance that a distributed unit of work will be left in an undetermined state. Even though two-phase commit protocol is used, no recovery process is included.

The general logic for managing a user transaction is as follows:

1. SQL Anywhere prefaces work to a remote server with a BEGIN TRANSACTION notification.

2. When the transaction is ready to be committed, SQL Anywhere sends a PREPARE TRANSACTION notification to each remote server that has been part of the transaction. This ensures that the remote server is ready to commit the transaction.

3. If a PREPARE TRANSACTION request fails, all remote servers are instructed to roll back the current transaction.

   If all PREPARE TRANSACTION requests are successful, the server sends a COMMIT TRANSACTION request to each remote server involved with the transaction.

Any statement preceded by BEGIN TRANSACTION can begin a transaction. Other statements are sent to a remote server to be executed as a single, remote unit of work.

Restrictions on transaction management

Restrictions on transaction management are as follows:

- Savepoints are not propagated to remote servers.
- If nested BEGIN TRANSACTION and COMMIT TRANSACTION statements are included in a transaction that involves remote servers, only the outermost set of statements is processed. The innermost set, containing the BEGIN TRANSACTION and COMMIT TRANSACTION statements, is not transmitted to remote servers.

Internal operations

This section describes the underlying steps that SQL Anywhere performs on remote servers on behalf of client applications.

Query parsing

When a statement is received from a client, the database server parses it. The database server raises an error if the statement is not a valid SQL Anywhere SQL statement.

Query normalization

Referenced objects in the query are verified and some data type compatibility is checked.

For example, consider the following query:

```
SELECT *
FROM t1
WHERE c1 = 10;
```


The query normalization stage verifies that table t1 with a column c1 exists in the system tables. It also verifies that the data type of column c1 is compatible with the value 10. If the column's data type is TIMESTAMP, for example, this statement is rejected.

Query preprocessing

Query preprocessing prepares the query for optimization. It may change the representation of a statement so that the SQL statement that SQL Anywhere generates for passing to a remote server is syntactically different from the original statement, even though it is semantically equivalent.

Preprocessing performs view expansion so that a query can operate on tables referenced by the view. Expressions may be reordered and subqueries may be transformed to improve processing efficiency. For example, some subqueries may be converted into joins.

Server capabilities

The previous steps are performed on all queries, both local and remote.

The following steps depend on the type of SQL statement and the capabilities of the remote servers involved.

In SQL Anywhere, each remote server has a set of capabilities defined for it. These capabilities are stored in the ISYSCAPABILITIES system table, and are initialized during the first connection to a remote server.

The generic server class odbc relies strictly on information returned from the ODBC driver to determine these capabilities. Other server classes such as db2odbc have more detailed knowledge of the capabilities of a remote server type and use that knowledge to supplement what is returned from the driver.

Once a server is added to ISYSCAPABILITIES, the capability information is retrieved only from the system table.

Since a remote server may not support all the features of a given SQL statement, SQL Anywhere must break the statement into simpler components to the point that the query can be given to the remote server. SQL features not passed off to a remote server must be evaluated by SQL Anywhere itself.

For example, a query may contain an ORDER BY statement. If a remote server cannot perform ORDER BY, the statement is sent to the remote server without it and SQL Anywhere performs the ORDER BY on the result returned, before returning the result to the user. The result is that the user can employ the full range of SQL Anywhere supported SQL without concern for the features of a particular back end.

Complete passthrough of the statement

For efficiency, SQL Anywhere passes off as much of the statement as possible to the remote server. Often, this is the complete statement originally given to SQL Anywhere.

SQL Anywhere will hand off the complete statement when:

- Every table in the statement resides on the same remote server.
- The remote server can process all of the syntax in the statement.

In rare conditions, it may actually be more efficient to let SQL Anywhere do some of the work instead of the remote server doing it. For example, SQL Anywhere may have a better sorting algorithm. In this case, you may consider altering the capabilities of a remote server using the ALTER SERVER statement.

For more information, see "ALTER SERVER statement" [*SQL Anywhere Server - SQL Reference*].

Partial passthrough of the statement

If a statement contains references to multiple servers, or uses SQL features not supported by a remote server, the query is decomposed into simpler parts.

SELECT

SELECT statements are broken down by removing portions that cannot be passed on and letting SQL Anywhere perform the work. For example, suppose a remote server can not process the ATAN2 function in the following statement:

```
SELECT a,b,c
WHERE ATAN2( b, 10 ) > 3
AND c = 10;
```

The statement sent to the remote server would be converted to:

```
SELECT a,b,c WHERE c = 10;
```

Then, SQL Anywhere locally applies `WHERE ATAN2( b, 10 ) > 3` to the intermediate result set.

Joins

When two tables are joined, one table is selected to be the outer table. The outer table is scanned based on the WHERE conditions that apply to it. For every qualifying row found, the other table, known as the inner table is scanned to find a row that matches the join condition.

This same algorithm is used when remote tables are referenced. Since the cost of searching a remote table is usually much higher than a local table (due to network I/O), every effort is made to make the remote table the outermost table in the join.

UPDATE and DELETE

When a qualifying row is found, if SQL Anywhere cannot pass off an UPDATE or DELETE statement entirely to a remote server, it must change the statement into a table scan containing as much of the original WHERE clause as possible, followed by a positioned UPDATE or DELETE statement that specifies WHERE CURRENT OF *cursor-name*.

For example, when the function ATAN2 is not supported by a remote server:

```
UPDATE t1
SET a = atan2( b, 10 )
WHERE b > 5;
```


Would be converted to the following:

```
SELECT a,b
FROM t1
WHERE  b > 5;
```

Each time a row is found, SQL Anywhere would calculate the new value of a and issue:

```
UPDATE t1
SET a = 'new value'
WHERE CURRENT OF CURSOR;
```

If a already has a value that equals the new value, a positioned UPDATE would not be necessary, and would not be sent remotely.

To process an UPDATE or DELETE statement that requires a table scan, the remote data source must support the ability to perform a positioned UPDATE or DELETE (WHERE CURRENT OF *cursor-name*). Some data sources do not support this capability.

Temporary tables cannot be updated
An UPDATE or DELETE cannot be performed if an intermediate temporary table is required. This occurs in queries with ORDER BY and some queries with subqueries.

Troubleshooting remote data access

This section provides some hints for troubleshooting access to remote servers.

Features not supported for remote data

The following SQL Anywhere features are not supported on remote data:

- ALTER TABLE statement against remote tables.
- triggers defined on proxy tables.
- SQL Remote.
- foreign keys that refer to remote tables.
- READTEXT, WRITETEXT, and TEXTPTR functions.
- positioned UPDATE and DELETE statements.
- UPDATE and DELETE statements requiring an intermediate temporary table.
- backward scrolling on cursors opened against remote data. Fetch statements must be NEXT or RELATIVE 1.

- calls to functions that contain an expression that references a proxy table.
- If a column on a remote table has a name that is a keyword on the remote server, you cannot access data in that column. You can execute a CREATE EXISTING TABLE statement, and import the definition but you cannot select that column.

Case sensitivity

The case sensitivity setting of your SQL Anywhere database should match the settings used by any remote servers accessed.

SQL Anywhere databases are created case insensitive by default. With this configuration, unpredictable results may occur when selecting from a case-sensitive database. Different results will occur depending on whether ORDER BY or string comparisons are pushed off to a remote server, or evaluated by the local SQL Anywhere server.

Connectivity tests

Take the following steps to be sure you can connect to a remote server:

- Determine that you can connect to a remote server using a client tool such as Interactive SQL before configuring SQL Anywhere.
- Perform a simple passthrough statement to a remote server to check your connectivity and remote login configuration. For example:

  ```
  FORWARD TO RemoteSA {SELECT @@version};
  ```

- Turn on remote tracing for a trace of the interactions with remote servers. For example:

  ```
  SET OPTION cis_option = 7;
  ```

  Once you have turned on remote tracing, the tracing information appears in the database server messages window. You can log this output to a file by specifying the -o server option when you start the database server.

  For more information about the cis_option option, see “cis_option option” [*SQL Anywhere Server - Database Administration*].

  For more information about the -o server option, see “-o dbeng12/dbsrv12 server option” [*SQL Anywhere Server - Database Administration*].

General problems with queries

If SQL Anywhere is having difficulty handling a query against a remote table, it is usually helpful to understand how SQL Anywhere is executing the query. You can display remote tracing, and a description of the query execution plan:


```
SET OPTION cis_option = 7;
```

Once you have turned on remote tracing, the tracing information appears in the database server messages window. You can log this output to a file by specifying the -o server option when you start the database server.

For more information about using the cis_option option for debugging queries when using remote data access, see "cis_option option" [*SQL Anywhere Server - Database Administration*].

For more information about the -o server option, see "-o dbeng12/dbsrv12 server option" [*SQL Anywhere Server - Database Administration*].

Queries blocked on themselves

You must have enough threads available to support the individual tasks that are being run by a query. Failure to provide the number of required tasks can lead to a query becoming blocked on itself. See "Transaction blocking and deadlock" on page 107.

Managing remote data access connections via ODBC

If you access remote databases via ODBC, the connection to the remote server is given a name. You can use the name to drop the connection to cancel a remote request.

The connections are named ASACIS_*conn-name*, where *conn-name* is the connection ID of the local connection. The connection ID can be obtained from the sa_conn_info stored procedure. See "sa_conn_info system procedure" [*SQL Anywhere Server - SQL Reference*].

Server classes for remote data access

The server class you specify in the CREATE SERVER statement determines the behavior of a remote connection. The server classes give SQL Anywhere detailed server capability information. SQL Anywhere formats SQL statements specific to a server's capabilities.

There are two categories of server classes:

- ODBC-based server classes
- JDBC-based server classes (deprecated)

Each server class has a set of unique characteristics that you need to know to configure the server for remote data access.

You should refer both to information generic to the server class category (JDBC-based or ODBC-based), and to the information specific to the individual server class.

ODBC-based server classes

The ODBC-based server classes include:

- saodbc
- ulodbc
- adsodbc
- aseodbc
- db2odbc
- iqodbc
- msaccessodbc
- mssodbc
- mysqlodbc
- odbc
- oraodbc

Note
When using remote data access, if you use an ODBC driver that does not support Unicode, then character set conversion is not performed on data coming from that ODBC driver.

Defining ODBC external servers

The most common way of defining an ODBC-based server is to base it on an ODBC data source. To do this, you can create a data source using the ODBC Administrator.

For more information, see "Creating ODBC data sources" [*SQL Anywhere Server - Database Administration*].

Once you have defined the data source, the USING clause in the CREATE SERVER statement should match the ODBC data source name.

For example, to configure a IBM DB2 server named mydb2 whose data source name is also mydb2, use:

```
CREATE SERVER mydb2
CLASS 'db2odbc'
USING 'mydb2';
```

For more information, see "CREATE SERVER statement" [*SQL Anywhere Server - SQL Reference*].

Using connection strings instead of data sources

An alternative, which avoids using data sources, is to supply a connection string in the USING clause of the CREATE SERVER statement. To do this, you must know the connection parameters for the ODBC driver you are using. For example, a connection to a SQL Anywhere database may be as follows:

```
CREATE SERVER TestSA
CLASS 'saodbc'
USING 'DRIVER=SQL Anywhere 12;HOST=myhost;Server=TestSA;DBN=sample';
```


This defines a connection to a SQL Anywhere database server named TestSA, running on a computer called myhost, and a database named sample using the TCP/IP protocol.

See also

For information specific to particular ODBC server classes, see:

- "Server class saodbc" on page 775
- "Server class ulodbc" on page 776
- "Server class adsodbc" on page 776
- "Server class aseodbc" on page 776
- "Server class db2odbc" on page 779
- "Server class msaccessodbc" on page 781
- "Server class mssodbc" on page 782
- "Server class mysqlodbc" on page 784
- "Server class odbc" on page 785
- "Server class oraodbc" on page 787

USING parameter in the CREATE SERVER statement

You must run a separate CREATE SERVER statement for each SQL Anywhere database you intend to access. For example, if a SQL Anywhere server named TestSA is running on the computer banana and owns three databases (db1, db2, db3), you would configure the local SQL Anywhere database server similar to this:

```
CREATE SERVER TestSAdb1
CLASS 'saodbc'
USING 'banana:2638/db1'
CREATE SERVER TestSAdb2
CLASS 'saodbc'
USING 'banana:2638/db2'
CREATE SERVER TestSAdb3
CLASS 'saodbc'
USING 'banana:2638/db3';
```

If you do not specify a */database-name* value, the remote connection uses the remote SQL Anywhere default database.

For more information, see "CREATE SERVER statement" [*SQL Anywhere Server - SQL Reference*].

Server class saodbc

A server with server class saodbc is a SQL Anywhere database server. No special requirements exist for the configuration of a SQL Anywhere data source.

To access SQL Anywhere database servers that support multiple databases, create an ODBC data source name defining a connection to each database. Issue a CREATE SERVER statement for each of these ODBC data source names.

Server class ulodbc

A server with server class ulodbc is an UltraLite database. Create an ODBC data source name defining a connection to the UltraLite database. Issue a CREATE SERVER statement for the ODBC data source name.

There is a one to one mapping between the UltraLite and SQL Anywhere data types because UltraLite supports a subset of the data types available in SQL Anywhere. See "Data types in UltraLite" [*UltraLite - Database Management and Reference*].

Note
You cannot create a remove server for an UltraLite database running on Mac OS X.

Server class adsodbc

When you issue a CREATE TABLE statement, SQL Anywhere automatically converts the data types to the corresponding Advantage Database Server data types using the following data type conversions.

SQL Anywhere data type	ADS default data type
BIT	Logical
TINYINT, SMALLINT, INT, INTEGER	Integer
BIGINT	Numeric(32)
DECIMAL(p,s), NUMERIC(p,s)	Numeric(p+3)
DATE	Date
TIME	Time
DATETIME, TIMESTAMP	TimeStamp
MONEY, SMALLMONEY	Money
FLOAT, REAL	Double
CHAR(*n*), VARCHAR(*n*), LONG VARCHAR	Char(*n*)
BINARY(*n*), VARBINARY(*n*), LONG BINARY	Blob

Server class aseodbc

A server with server class aseodbc is a Sybase SQL Server and Adaptive Server Enterprise (version 10 and later) database server. SQL Anywhere requires the installation of the Adaptive Server Enterprise

ODBC driver and Open Client connectivity libraries to connect to a remote Adaptive Server Enterprise server with class aseodbc, but the performance is better than with the ASEJDBC class.

Notes

- Open Client should be version 11.1.1, EBF 7886 or later. Install Open Client and verify connectivity to the Adaptive Server Enterprise server before you install ODBC and configure SQL Anywhere. The Sybase ODBC driver should be version 11.1.1, EBF 7911 or later.
- The local setting of the quoted_identifier option controls the use of quoted identifiers for Adaptive Server Enterprise. For example, if you set the quoted_identifier option to Off locally, then quoted identifiers are turned off for Adaptive Server Enterprise.
- Configure a user data source in the **Configuration Manager** with the following attributes:
  - **General tab** Type any value for **Data Source Name**. This value is used in the USING clause of the CREATE SERVER statement.

    The server name should match the name of the server in the Sybase interfaces file.

    For more information about the interfaces file, see "The interfaces file" [*SQL Anywhere Server - Database Administration*].
  - **Advanced tab** Select the **Application Using Threads** and **Enable Quoted Identifiers** options.
  - **Connection tab** Set the charset field to match your SQL Anywhere character set.

    Set the language field to your preferred language for error messages.
  - **Performance tab** Set the **Prepare Method** to **2-Full**.

    Set the **Fetch Array Size** as large as possible for the best performance. This increases memory requirements since this is the number of rows that must be cached in memory. Adaptive Server Enterprise recommends using a value of 100.

    Set **Select Method** to **0-Cursor**.

    Set **Packet Size** to as large a value as possible. Adaptive Server Enterprise recommends using a value of -1.

    Set **Connection Cache** to 1.

Data type conversions: ODBC and Adaptive Server Enterprise

When you issue a CREATE TABLE statement, SQL Anywhere automatically converts the data types to the corresponding Adaptive Server Enterprise data types. The following table describes the SQL Anywhere to Adaptive Server Enterprise data type conversions.

SQL Anywhere data type	Adaptive Server Enterprise default data type
BIT	bit

SQL Anywhere data type	Adaptive Server Enterprise default data type
TINYINT	tinyint
SMALLINT	smallint
INT	int
INTEGER	integer
DECIMAL [defaults p=30, s=6]	numeric(30,6)
DECIMAL(128,128)	not supported
NUMERIC [defaults p=30 s=6]	numeric(30,6)
NUMERIC(128,128)	not supported
FLOAT	real
REAL	real
DOUBLE	float
SMALLMONEY	numeric(10,4)
MONEY	numeric(19,4)
DATE	datetime
TIME	datetime
TIMESTAMP	datetime
SMALLDATETIME	datetime
DATETIME	datetime
CHAR(*n*)	varchar(*n*)
CHARACTER(*n*)	varchar(*n*)
VARCHAR(*n*)	varchar(*n*)
CHARACTER VARYING(*n*)	varchar(*n*)
LONG VARCHAR	text
TEXT	text

SQL Anywhere data type	Adaptive Server Enterprise default data type
BINARY(*n*)	binary(*n*)
LONG BINARY	image
IMAGE	image
BIGINT	numeric(20,0)

Server class db2odbc

A server with server class db2odbc is IBM DB2.

Notes

- Sybase certifies the use of IBM's DB2 Connect version 5, with fix pack WR09044. Configure and test your ODBC configuration using the instructions for that product. SQL Anywhere has no specific requirements for the configuration of IBM DB2 data sources.
- The following is an example of a CREATE EXISTING TABLE statement for a IBM DB2 server with an ODBC data source named mydb2:

```
CREATE EXISTING TABLE ibmcol
AT 'mydb2..sysibm.syscolumns';
```

Data type conversions: IBM DB2

When you issue a CREATE TABLE statement, SQL Anywhere automatically converts the data types to the corresponding IBM DB2 data types. The following table describes the SQL Anywhere to IBM DB2 data type conversions.

SQL Anywhere data type	IBM DB2 default data type
BIT	smallint
TINYINT	smallint
SMALLINT	smallint
INT	int
INTEGER	int
BIGINT	decimal(20,0)
CHAR(1-254)	varchar(*n*)

SQL Anywhere data type	IBM DB2 default data type
CHAR(255-4000)	varchar(*n*)
CHAR(4001-32767)	long varchar
CHARACTER(1-254)	varchar(*n*)
CHARACTER(255-4000)	varchar(*n*)
CHARACTER(4001-32767)	long varchar
VARCHAR(1-4000)	varchar(*n*)
VARCHAR(4001-32767)	long varchar
CHARACTER VARYING(1-4000)	varchar(*n*)
CHARACTER VARYING(4001-32767)	long varchar
LONG VARCHAR	long varchar
TEXT	long varchar
BINARY(1-4000)	varchar for bit data
BINARY(4001-32767)	long varchar for bit data
LONG BINARY	long varchar for bit data
IMAGE	long varchar for bit data
DECIMAL [defaults p=30, s=6]	decimal(30,6)
NUMERIC [defaults p=30 s=6]	decimal(30,6)
DECIMAL(128, 128)	NOT SUPPORTED
NUMERIC(128, 128)	NOT SUPPORTED
REAL	real
FLOAT	float
DOUBLE	float
SMALLMONEY	decimal(10,4)
MONEY	decimal(19,4)

SQL Anywhere data type	IBM DB2 default data type
DATE	date
TIME	time
SMALLDATETIME	timestamp
DATETIME	timestamp
TIMESTAMP	timestamp

Server class iqodbc

A server with server class iqodbc is a Sybase IQ server. No special requirements exist for the configuration of a Sybase IQ data source.

To access Sybase IQ servers that support multiple databases, create an ODBC data source name defining a connection to each database. Issue a CREATE SERVER statement for each of these ODBC data source names. See "USING parameter in the CREATE SERVER statement" on page 775.

Server class msaccessodbc

Access databases are stored in a *.mdb* file. Using the ODBC manager, create an ODBC data source and map it to one of these files. A new *.mdb* file can be created through the ODBC manager. This database file becomes the default if you don't specify a different default when you create a table through SQL Anywhere.

Assuming an ODBC data source named access, you can use any of the following statements to access data:

- ```
  CREATE TABLE tab1 (a int, b char(10))
  AT 'access...tab1';
  ```
- ```
  CREATE TABLE tab1 (a int, b char(10))
  AT 'access;d:\pcdb\data.mdb;;tab1';
  ```
- ```
  CREATE EXISTING TABLE tab1
  AT 'access;d:\pcdb\data.mdb;;tab1';
  ```

Access does not support the owner name qualification; leave it empty.

Data type conversions: Microsoft Access

SQL Anywhere data type	Microsoft Access default data type
BIT, TINYINT	TINYINT
  ```

SQL Anywhere data type	Microsoft Access default data type
SMALLINT	SMALLINT
INT, INTEGER	INTEGER
BIGINT	DECIMAL(19,0)
DECIMAL(p,s), NUMERIC(p,s)	DECIMAL(p,s)
DATE, TIME, DATETIME, TIMESTAMP	DATETIME
MONEY, SMALLMONEY	MONEY
FLOAT	FLOAT
REAL	REAL
CHAR(*n*), VARCHAR(*n*)	CHARACTER(*n*) if *n* is less than 254 TEXT if *n* is greater than or equal to 254
LONG VARCHAR	TEXT
BINARY, VARBINARY	BINARY(*n*) if *n* is less than 4000 IMAGE if *n* is greater than or equal to 4000
LONG BINARY	IMAGE

Server class mssodbc

A server with server class mssodbc is Microsoft SQL Server version 6.5, Service Pack 4.

Notes

- Sybase certifies the use of Microsoft SQL Server version 3.60.0319 ODBC driver (included in the MDAC 2.0 release). Configure and test your ODBC configuration using the instructions for that product.
- The following is an example of a CREATE EXISTING TABLE statement for a Microsoft SQL Server named mymssql:

```
CREATE EXISTING TABLE accounts,
AT 'mymssql.database.owner.accounts';
```

- The local setting of the quoted_identifier option controls the use of quoted identifiers for Microsoft SQL Server. For example, if you set the quoted_identifier option to Off locally, then quoted identifiers are turned off for Microsoft SQL Server.

Data type conversions: Microsoft SQL Server

When you issue a CREATE TABLE statement, SQL Anywhere automatically converts the data types to the corresponding Microsoft SQL Server data types using the following data type conversions.

SQL Anywhere data type	Microsoft SQL Server default data type
BIT	bit
TINYINT	tinyint
SMALLINT	smallint
INT	int
BIGINT	numeric(20,0)
DECIMAL [defaults p=30, s=6]	decimal(prec, scale)
NUMERIC [defaults p=30 s=6]	numeric(prec, scale)
FLOAT	if (prec) float(prec) else float
REAL	real
SMALLMONEY	smallmoney
MONEY	money
DATE	datetime
TIME	datetime
TIMESTAMP	datetime
SMALLDATETIME	datetime
DATETIME	datetime
CHAR(*n*)	if (length > 255) text else varchar(length)
CHARACTER(*n*)	char(*n*)
VARCHAR(*n*)	if (length > 255) text else varchar(length)
LONG VARCHAR	text
BINARY(*n*)	if (length > 255) image else binary(length)
LONG BINARY	image

SQL Anywhere data type	Microsoft SQL Server default data type
DOUBLE	float
UNIQUEIDENTIFIERSTR	uniqueidentifier

Server class mysqlodbc

When you issue a CREATE TABLE statement, SQL Anywhere automatically converts the data types to the corresponding MySQL data types using the following data type conversions.

SQL Anywhere data type	MySQL default data type
BIT	bit(1)
TINYINT	tinyint unsigned
SMALLINT	smallint
INT, INTEGER	int
BIGINT	bigint
DECIMAL(p,s), NUMERIC(p,s)	decimal(p,s)
DATE	date
TIME	time
DATETIME, TIMESTAMP	datetime
MONEY	decimal(19,4)
SMALLMONEY	decimal(10,4)
FLOAT	float
REAL	real
CHAR(*n*)	char(*n*) if *n* is less than 254 varchar(*n*) if *n* is greater than or equal to 254 but less than 4000 longtext if *n* is greater than or equal to 4000

SQL Anywhere data type	MySQL default data type
VARCHAR(*n*)	varchar(*n*) if *n* is less than 4000 longtext if *n* is greater than or equal to 4000
LONG VARCHAR	longtext
BINARY(*n*), VARBINARY(*n*)	varbinary(*n*) if *n* is less than 4000 longblob if *n* is greater than or equal to 4000
LONG BINARY	longblob

Server class odbc

ODBC data sources that do not have their own server class use server class **odbc**. You can use any ODBC driver. Sybase certifies the following ODBC data sources:

- "Microsoft Excel (Microsoft 3.51.171300)" on page 785
- "Microsoft FoxPro (Microsoft 3.51.171300)" on page 786
- "Lotus Notes SQL 2.0" on page 786

The latest versions of Microsoft ODBC drivers can be obtained through the Microsoft Data Access Components (MDAC) distribution found at the Microsoft Download Center. The Microsoft driver versions listed above are part of MDAC 2.0.

Microsoft Excel (Microsoft 3.51.171300)

With Excel, each Excel workbook is logically considered to be a database holding several tables. Tables are mapped to sheets in a workbook. When you configure an ODBC data source name in the ODBC driver manager, you specify a default workbook name associated with that data source. However, when you issue a CREATE TABLE statement, you can override the default and specify a workbook name in the location string. This allows you to use a single ODBC DSN to access all of your excel workbooks.

In this example, a remote server named excel was created. To create a workbook named *work1.xls* with a sheet (table) called mywork:

```
CREATE TABLE mywork (a int, b char(20))
AT 'excel;d:\work1.xls;;mywork';
```

To create a second sheet (or table) execute a statement such as:

```
CREATE TABLE mywork2 (x float, y int)
AT 'excel;d:\work1.xls;;mywork2';
```

You can import existing worksheets into SQL Anywhere using CREATE EXISTING, under the assumption that the first row of your spreadsheet contains column names.

```
CREATE EXISTING TABLE mywork
AT'excel;d:\work1;;mywork';
```

If SQL Anywhere reports that the table is not found, you may need to explicitly state the column and row range you want to map to. For example:

```
CREATE EXISTING TABLE mywork
AT 'excel;d:\work1;;mywork$';
```

Adding the $ to the sheet name indicates that the entire worksheet should be selected.

Note in the location string specified by AT that a semicolon is used instead of a period for field separators. This is because periods occur in the file names. Excel does not support the owner name field so leave this blank.

Deletes are not supported. Also some updates may not be possible since the Excel driver does not support positioned updates.

Microsoft FoxPro (Microsoft 3.51.171300)

You can store FoxPro tables together inside a single FoxPro database file (*.dbc*), or, you can store each table in its own separate *.dbf* file. When using *.dbf* files, be sure the file name is filled into the location string; otherwise the directory that SQL Anywhere was started in is used.

```
CREATE TABLE fox1 (a int, b char(20))
AT 'foxpro;d:\pcdb;;fox1';
```

This statement creates a file named *d:\pcdb\fox1.dbf* when you choose the **Free Table Directory** option in the ODBC Driver Manager.

Lotus Notes SQL 2.0

To obtain this driver, go to the Lotus web site at http://www.lotus.com/. Read the documentation that is included with it for an explanation of how Notes data maps to relational tables. You can easily map SQL Anywhere tables to Notes forms.

Here is how to set up SQL Anywhere to access the Address sample file.

- Create an ODBC data source using the NotesSQL driver. The database will be the sample names file: *c:\notes\data\names.nsf*. The **Map Special Characters** option should be turned on. For this example, the **Data Source Name** is *my_notes_dsn*.
- Create a server in SQL Anywhere:

  ```
  CREATE SERVER names
  CLASS 'odbc'
  USING 'my_notes_dsn';
  ```

- Map the Person form into a SQL Anywhere table:


```
CREATE EXISTING TABLE Person
AT 'names...Person';
```

- Query the table:

```
SELECT * FROM Person;
```

Avoiding password prompts

Lotus Notes does not support sending a user name and password through the ODBC API. If you try to access Lotus notes using a password protected ID, a window appears on the computer where SQL Anywhere is running, and prompts you for a password. Avoid this behavior in multi-user server environments.

To access Lotus Notes unattended, without ever receiving a password prompt, you must use a non-password-protected ID. You can remove password protection from your ID by clearing it (choose **File** » **Tools** » **User ID** » **Clear Password**), unless your Domino administrator required a password when your ID was created. In this case, you will not be able to clear it.

Server class oraodbc

A server with server class oraodbc is Oracle version 8.0 or later.

Notes

- Sybase certifies the use of the Oracle version 8.0.03 ODBC driver. Configure and test your ODBC configuration using the instructions for that product.
- The following is an example of a CREATE EXISTING TABLE statement for an Oracle server named myora:

```
CREATE EXISTING TABLE employees
AT 'myora.database.owner.employees';
```

- As a result of Oracle ODBC driver restrictions, you cannot issue a CREATE EXISTING TABLE statement for system tables. A message returns stating that the table or columns cannot be found.

Data type conversions: Oracle

When you issue a CREATE TABLE statement, SQL Anywhere automatically converts the data types to the corresponding Oracle data types using the following data type conversions.

SQL Anywhere data type	Oracle data type
BIT	number(1,0)
TINYINT	number(3,0)
SMALLINT	number(5,0)

SQL Anywhere data type	Oracle data type
INT	number(11,0)
BIGINT	number(20,0)
DECIMAL(prec, scale)	number(prec, scale)
NUMERIC(prec, scale)	number(prec, scale)
FLOAT	float
REAL	real
SMALLMONEY	numeric(13,4)
MONEY	number(19,4)
DATE	date
TIME	date
TIMESTAMP	date
SMALLDATETIME	date
DATETIME	date
CHAR(*n*)	if (n > 255) long else varchar(*n*)
VARCHAR(*n*)	if (n > 2000) long else varchar(*n*)
LONG VARCHAR	long or clob
BINARY(*n*)	if (n > 255) long raw else raw(*n*)
VARBINARY(*n*)	if (n > 255) long raw else raw(*n*)
LONG BINARY	long raw

JDBC-based server classes (deprecated)

Support for the JDBC-based server classes is deprecated. You should update your applications to use the ODBC-based server classes. See “Defining ODBC external servers” on page 774.

JDBC-based server classes are used when SQL Anywhere internally uses a Java VM and jConnect 5.5 to connect to the remote server. The JDBC-based server classes are:

- **SAJDBC** SQL Anywhere.
- **ASEJDBC** Sybase SQL Server and Adaptive Server Enterprise (version 10 and later).
- **IQJDBC** Sybase IQ.

Configuration notes for JDBC classes

When you access remote servers defined with JDBC-based classes, consider that:

- For optimum performance, an ODBC-based class is recommended (saodbc or aseodbc).
- Any remote server that you access using the ASEJDBC or SAJDBC server class must be set up to handle a jConnect 6.x-based client.
- If a JDBC remote server connection is disconnected or lost, you only find out that the server is unavailable if you attempt to use the JDBC remote server to access a proxy object, such as a proxy table or proxy procedure. ODBC does not have this limitation.

Server class SAJDBC (deprecated)

This server class is deprecated. You should update your application to use server class saodbc. See "Server class saodbc" on page 775.

A server with server class SAJDBC is a SQL Anywhere server. No special requirements exist for the configuration of a SQL Anywhere data source.

Server class ASEJDBC (deprecated)

This server class is deprecated. You should update your application to use server class aseodbc. See "Server class aseodbc" on page 776.

A server with server class ASEJDBC is a Sybase SQL Server and Adaptive Server Enterprise (version 10 and later) server. No special requirements exist for the configuration of an Adaptive Server Enterprise data source.

Notes

- The local setting of the quoted_identifier option controls the use of quoted identifiers for Adaptive Server Enterprise. For example, if you set the quoted_identifier option to Off locally, then quoted identifiers are turned off for Adaptive Server Enterprise.

Data type conversions: JDBC and Adaptive Server Enterprise

When you issue a CREATE TABLE statement, SQL Anywhere automatically converts the data types to the corresponding Adaptive Server Enterprise data types using the following data type conversions.

SQL Anywhere data type	Adaptive Server Enterprise default data type
BIT	bit
TINYINT	tinyint
SMALLINT	smallint
INT	int
INTEGER	integer
DECIMAL [defaults p=30, s=6]	numeric(30,6)
DECIMAL(128,128)	not supported
NUMERIC [defaults p=30 s=6]	numeric(30,6)
NUMERIC(128,128)	not supported
FLOAT	real
REAL	real
DOUBLE	float
SMALLMONEY	numeric(10,4)
MONEY	numeric(19,4)
DATE	datetime
TIME	datetime
TIMESTAMP	datetime
SMALLDATETIME	datetime
DATETIME	datetime
CHAR(*n*)	varchar(*n*)
CHARACTER(*n*)	varchar(*n*)
VARCHAR(*n*)	varchar(*n*)
CHARACTER VARYING(*n*)	varchar(*n*)
LONG VARCHAR	text

SQL Anywhere data type	Adaptive Server Enterprise default data type
TEXT	text
BINARY(*n*)	binary(*n*)
LONG BINARY	image
IMAGE	image
BIGINT	numeric(19,0)

Server class IQJDBC (deprecated)

This server class is deprecated. You should update your application to use server class iqodbc. See "Server class iqodbc" on page 781.

A server with server class IQJDBC is a Sybase IQ server. No special requirements exist for the configuration of a Sybase IQ data source.

Stored procedures and triggers

This section describes how to build logic into your database using SQL stored procedures and triggers. Storing logic in the database makes it available automatically to all applications, providing consistency, performance, and security benefits. This section also provides information for using the SQL Anywhere debugger to identify logic issues in your application design.

Using procedures, triggers, and batches

Procedures and triggers store procedural SQL statements in a database for use by all applications. They can include control statements that allow repetition (LOOP statement) and conditional execution (IF statement and CASE statement) of SQL statements. Batches are sets of SQL statements submitted to the database server as a group. Many features available in procedures and triggers, such as control statements, are also available in batches.

Procedures are invoked with a CALL statement, and use parameters to accept values and return values to the calling environment. SELECT statements can also operate on procedure result sets by including the procedure name in the FROM clause.

Procedures can return result sets to the caller, call other procedures, or fire triggers. For example, a user-defined function is a type of stored procedure that returns a single value to the calling environment. User-defined functions do not modify parameters passed to them, but rather, they broaden the scope of functions available to queries and other SQL statements.

Triggers are associated with specific database tables. They fire automatically whenever someone inserts, updates or deletes rows of the associated table. Triggers can call procedures and fire other triggers, but they have no parameters and cannot be invoked by a CALL statement.

SQL Anywhere debugger

You can debug stored procedures and triggers using the SQL Anywhere debugger. See “Debugging procedures, functions, triggers, and events” on page 840.

You can profile stored procedures to analyze performance characteristics in Sybase Central. See “Procedure profiling using system procedures” on page 189.

Benefits of procedures and triggers

Procedures and triggers enhance the security, efficiency, and standardization of databases.

Definitions for procedures and triggers appear in the database, separately from any one database application. This separation provides several advantages.

Standardization

Procedures and triggers standardize actions performed by more than one application program. By coding the action once and storing it in the database for future use, applications need only call the procedure or

fire the trigger to achieve the desired result repeatedly. And since changes occur in only one place, all applications using the action automatically acquire the new functionality if the implementation of the action changes.

Efficiency

Procedures and triggers used in a network database server environment can access data in the database without requiring network communication. This means they execute faster and with less impact on network performance than if they had been implemented in an application on one of the client machines.

When you create a procedure or trigger, it is automatically checked for correct syntax, and then stored in the system tables. The first time any application calls or fires a procedure or trigger, it is compiled from the system tables into the server's virtual memory and executed from there. Since one copy of the procedure or trigger remains in memory after the first execution, repeated executions of the same procedure or trigger happen instantly. As well, several applications can use a procedure or trigger concurrently, or one application can use it recursively.

Security

Procedures and triggers provide security by allowing users limited access to data in tables that they cannot directly examine or modify.

Triggers, for example, execute under the table permissions of the owner of the associated table, but any user with permissions to insert, update or delete rows in the table can fire them. Similarly, procedures (including user-defined functions) execute with permissions of the procedure owner, but any user granted permissions can call them. This means that procedures and triggers can (and usually do) have different permissions than the user ID that invoked them.

Introduction to procedures

Creating procedures

In Sybase Central, the **Create Procedure Wizard** provides the option of using procedure templates. Alternatively, you can use Interactive SQL to execute a CREATE PROCEDURE statement to create a procedure. You must have DBA or RESOURCE authority to create procedure.

To create a new procedure (Sybase Central)

1. Use the SQL Anywhere 12 plug-in to connect to the database as a user with DBA or Resource authority.
2. In the left pane, double-click **Procedures & Functions**.
3. Choose **File** » **New** » **Procedure**.
4. Follow the instructions in the **Create Procedure Wizard**.
5. In the right pane, click the **SQL** tab to complete the procedure code.

The new procedure appears in **Procedures & Functions**.

Example

The following simple example creates the procedure NewDepartment, which performs an INSERT into the Departments table of the SQL Anywhere sample database, creating a new department.

```
CREATE PROCEDURE NewDepartment(
   IN id INT,
   IN name CHAR(35),
   IN head_id INT )
BEGIN
   INSERT
   INTO Departments ( DepartmentID,
       DepartmentName, DepartmentHeadID )
   VALUES ( id, name, head_id );
END;
```

The body of a procedure is a compound statement. The compound statement starts with a BEGIN statement and concludes with an END statement. For NewDepartment the compound statement is a single INSERT bracketed by BEGIN and END statements.

Parameters to procedures can be marked as one of IN, OUT, or INOUT. By default, parameters are INOUT parameters. All parameters to the NewDepartment procedure are IN parameters, as they are not changed by the procedure. You should set parameters to IN if they are not used to return values to the caller.

Temporary procedures

To create a temporary procedure, you must use the CREATE TEMPORARY PROCEDURE statement, an extension of the CREATE PROCEDURE statement. Temporary procedures are not permanently stored in the database. Instead, they are dropped at the end of a connection, or when specifically dropped, whichever occurs first. See “CREATE PROCEDURE statement” [*SQL Anywhere Server - SQL Reference*].

Remote procedures

To create a remote procedure, you must have at least one remote server. See:

- “Create remote procedures” on page 765
- “Create remote servers” on page 750

See also

- “SQL Anywhere database connections” [*SQL Anywhere Server - Database Administration*]
- “CREATE PROCEDURE statement” [*SQL Anywhere Server - SQL Reference*]
- “ALTER PROCEDURE statement” [*SQL Anywhere Server - SQL Reference*]
- “Using compound statements” on page 816
- “Create remote procedures” on page 765
- “CALL statement” [*SQL Anywhere Server - SQL Reference*]

Altering procedures

You can modify an existing procedure using either Sybase Central or Interactive SQL. You must have DBA authority or be the owner of the procedure.

In Sybase Central, you cannot rename an existing procedure directly. Instead, you must create a new procedure with the new name, copy the previous code to it, and then delete the old procedure.

In Interactive SQL, you can execute an ALTER PROCEDURE statement to modify an existing procedure. You must include the entire new procedure in this statement (in the same syntax as in the CREATE PROCEDURE statement that created the procedure).

To alter the code of a procedure (Sybase Central)

1. Use the SQL Anywhere 12 plug-in to connect to the database as a user with DBA or Resource authority.
2. In the left pane, double-click **Procedures & Functions**.
3. Select the procedure.
4. Use one of the following methods to edit the procedure:
   - In the right pane, click the **SQL** tab.
   - Right-click the procedure and choose **Edit In New Window**.

     Tip
     You can open a separate window for each procedure and copy code between procedures.

   - To add or edit a procedure comment, right-click the procedure and choose **Properties**.

     If you use the **Database Documentation Wizard** to document your SQL Anywhere database, you will have the option to include these comments in the output. See "Documenting a database" [*SQL Anywhere Server - Database Administration*].

See also

- "Set properties for database objects" on page 1
- "Granting permissions on procedures" [*SQL Anywhere Server - Database Administration*]
- "Revoking user permissions and authorities" [*SQL Anywhere Server - Database Administration*]
- "ALTER PROCEDURE statement" [*SQL Anywhere Server - SQL Reference*]
- "CREATE PROCEDURE statement" [*SQL Anywhere Server - SQL Reference*]
- "Creating procedures" on page 794
- "Using Sybase Central to translate stored procedures" on page 659

Calling procedures

CALL statements invoke procedures. Procedures can be called by an application program, or by other procedures and triggers.

The following statement calls the NewDepartment procedure to insert an Eastern Sales department:

```
CALL NewDepartment( 210, 'Eastern Sales', 902 );
```


After this call, you may want to check the Departments table to see that the new department has been added.

All users who have been granted EXECUTE permissions for the procedure can call the NewDepartment procedure, even if they have no permissions on the Departments table.

Another way of calling a procedure that returns a result set is to call it in a query. You can execute queries on result sets of procedures and apply WHERE clauses and other SELECT features to limit the result set.

```
SELECT t.ID, t.QuantityOrdered AS q
FROM ShowCustomerProducts( 149 ) t;
```

See also

- "Database permissions and authorities" [*SQL Anywhere Server - Database Administration*]
- "CALL statement" [*SQL Anywhere Server - SQL Reference*]
- "GRANT statement" [*SQL Anywhere Server - SQL Reference*]
- "FROM clause" [*SQL Anywhere Server - SQL Reference*]

Copying procedures in Sybase Central

To copy procedures between databases in Sybase Central, select the procedure in the left pane and drag it to **Procedures & Functions** of another connected database. A new procedure is then created, and the original procedure's code is copied to it.

Only the procedure code is copied to the new procedure and the other procedure properties (permissions, and so on) are not copied. A procedure can be copied to the same database, provided you give it a new name.

Deleting procedures

Once you create a procedure, it remains in the database until someone explicitly removes it. Only the owner of the procedure or a user with DBA authority can drop the procedure from the database.

To delete a procedure (Sybase Central)

1. Use the SQL Anywhere 12 plug-in to connect to the database as a user with DBA authority or as the owner of the procedure.
2. In the left pane, double-click **Procedures & Functions**.
3. Select the procedure and choose **Edit** » **Delete**.
4. Click **Yes**.

To delete a procedure (SQL)

1. Connect to a database as a user with DBA authority or as the owner of the procedure.
2. Execute a DROP PROCEDURE statement.

Example

The following statement removes the procedure NewDepartment from the database:

```
DROP PROCEDURE NewDepartment;
```

See also

- "SQL Anywhere database connections" [*SQL Anywhere Server - Database Administration*]
- "DROP PROCEDURE statement" [*SQL Anywhere Server - SQL Reference*]

Returning procedure results in parameters

Procedures return results to the calling environment in one of the following ways:

- Individual values are returned as OUT or INOUT parameters.
- Result sets can be returned.
- Procedures can return a single result using a RETURN statement.

To create and run a procedure, and display its output (SQL)

1. Using Interactive SQL, connect to the SQL Anywhere sample database as the DBA.
2. In the SQL Statements pane, type the following to create a procedure (AverageSalary) that returns the average salary of employees as an OUT parameter:

   ```
   CREATE PROCEDURE AverageSalary( OUT average_salary NUMERIC(20,3) )
   BEGIN
      SELECT AVG( Salary )
      INTO average_salary
      FROM Employees;
   END;
   ```

3. Create a variable to hold the procedure output. In this case, the output variable is numeric, with three decimal places, so create a variable as follows:

   ```
   CREATE VARIABLE Average NUMERIC(20,3);
   ```

4. Call the procedure using the created variable to hold the result:

   ```
   CALL AverageSalary( Average );
   ```

   If the procedure was created and run properly, the Interactive SQL **Messages** tab does not display any errors.

5. To inspect the value of the variable, execute the following statement:

   ```
   SELECT Average;
   ```

   Look at the value of the output variable Average. The **Results** tab in the **Results** pane displays the value 49988.623 for this variable, the average employee salary.

See also

- "SQL Anywhere database connections" [*SQL Anywhere Server - Database Administration*]

Returning procedure results in result sets

In addition to returning results to the calling environment in individual parameters, procedures can return information in result sets. A result set is typically the result of a query. The following procedure returns a result set containing the salary for each employee in a given department:

```
CREATE PROCEDURE SalaryList( IN department_id INT )
RESULT ( "Employee ID" INT, Salary NUMERIC(20,3) )
BEGIN
   SELECT EmployeeID, Salary
    FROM Employees
    WHERE Employees.DepartmentID = department_id;
END;
```

If Interactive SQL calls this procedure, the names in the RESULT clause are matched to the results of the query and used as column headings in the displayed results.

To test this procedure from Interactive SQL, you can CALL it, specifying one of the departments of the company. In Interactive SQL, the results appear on the **Results** tab in the **Results** pane.

Example

To list the salaries of employees in the R & D department (department ID 100), type the following:

```
CALL SalaryList( 100 );
```

Employee ID	Salary
102	45700.000
105	62000.000
160	57490.000
243	72995.000
...	...

Interactive SQL can only return multiple result sets if you have this option enabled on the **Results** tab of the **Options** window. Each result set appears on a separate tab in the **Results** pane.

See also

- "Returning multiple result sets from procedures" on page 823

Introduction to user-defined functions

User-defined functions are a class of procedures that return a single value to the calling environment. This section introduces creating, using, and dropping user-defined functions.

Note
SQL Anywhere does not make any assumptions about whether user-defined functions are thread-safe. This is the responsibility of the application developer.

Creating user-defined functions

You use the CREATE FUNCTION statement to create user-defined functions. You must have RESOURCE authority to execute this statement.

The following simple example creates a function that concatenates two strings, together with a space, to form a full name from a first name and a last name.

```
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 CREATE FUNCTION syntax differs slightly from that of the CREATE PROCEDURE statement. The following are distinctive differences:

- No IN, OUT, or INOUT keywords are required, as all parameters are IN parameters.
- The RETURNS clause is required to specify the data type being returned.
- The RETURN statement is required to specify the value being returned.

You can also create user-defined functions from Sybase Central.

To create a user-defined function (Sybase Central)

1. Use the SQL Anywhere 12 plug-in to connect to the database as a user with DBA or Resource authority.
2. In the left pane, click **Procedures & Functions**.
3. Choose **File** » **New** » **Function**.
4. Follow the instructions in the **Create Function Wizard**.
5. In the right pane, click the **SQL** tab to complete the procedure code.

   The new function appears in **Procedures & Functions**.

See also

- "CREATE FUNCTION statement" [*SQL Anywhere Server - SQL Reference*]
- "CREATE FUNCTION statement (external procedures)" [*SQL Anywhere Server - SQL Reference*]
- "CREATE FUNCTION statement (external procedures)" [*SQL Anywhere Server - SQL Reference*]

Calling user-defined functions

A user-defined function can be used, subject to permissions, in any place you would use a built-in non-aggregate function.

The following statement in Interactive SQL returns a full name from two columns containing a first and last name:

```
SELECT FullName(GivenName, Surname)
 AS "Full Name"
 FROM Employees;
```

Full Name
Fran Whitney
Matthew Cobb
Philip Chin
...

The following statement in Interactive SQL returns a full name from a supplied first and last name:

```
SELECT FullName('Jane', 'Smith')
 AS "Full Name";
```

Full Name
Jane Smith

Any user who has been granted EXECUTE permissions for the function can use the FullName function.

Example

The following user-defined function illustrates local declarations of variables.

The Customers table includes Canadian and US customers. The user-defined function Nationality forms a 3-letter country code based on the Country column.

```
CREATE FUNCTION Nationality( CustomerID INT )
RETURNS CHAR( 3 )
BEGIN
    DECLARE nation_string CHAR(3);
    DECLARE nation country_t;
    SELECT DISTINCT Country INTO nation
```

```
    FROM Customers
    WHERE ID = CustomerID;
    IF nation = 'Canada' THEN
            SET nation_string = 'CDN';
    ELSE IF nation = 'USA' OR nation = ' ' THEN
            SET nation_string = 'USA';
        ELSE
            SET nation_string = 'OTH';
        END IF;
    END IF;
RETURN ( nation_string );
END;
```

This example declares a variable nation_string to hold the nationality string, uses a SET statement to set a value for the variable, and returns the value of the nation_string string to the calling environment.

The following query lists all Canadian customers in the Customers table:

```
SELECT *
FROM Customers
WHERE Nationality(ID) = 'CDN';
```

Declarations of cursors and exceptions are discussed in later sections.

Notes

While this function is useful for illustration, it may perform very poorly if used in a SELECT involving many rows. For example, if you used the function in the SELECT list of a query on a table containing 100,000 rows, of which 10,000 are returned, the function will be called 10,000 times. If you use it in the WHERE clause of the same query, it would be called 100,000 times.

Dropping user-defined functions

Once you create a user-defined function, it remains in the database until someone explicitly removes it. Only the owner of the function or a user with DBA authority can drop a function from the database.

The following statement removes the function FullName from the database:

```
DROP FUNCTION FullName;
```

See also

- "DROP FUNCTION statement" [*SQL Anywhere Server - SQL Reference*]

Permissions to execute user-defined functions

Ownership of a user-defined function belongs to the user who created it, and that user can execute it without permission. The owner of a user-defined function can grant permissions to other users with the GRANT EXECUTE command.

For example, the creator of the function FullName could allow another user to use FullName with the statement:


```
GRANT EXECUTE ON Nationality TO BobS;
```

The following statement revokes permissions to use the function:

```
REVOKE EXECUTE ON Nationality FROM BobS;
```

See "Granting permissions on procedures" [*SQL Anywhere Server - Database Administration*].

Advanced information on user-defined functions

SQL Anywhere treats all user-defined functions as **idempotent** unless they are declared NOT DETERMINISTIC. Idempotent functions return a consistent result for the same parameters and are free of side effects. Two successive calls to an idempotent function with the same parameters return the same result, and have no unwanted side-effects on the query's semantics.

For more information about non-deterministic and deterministic functions, see "Function caching" on page 587.

Introduction to triggers

A trigger is a special form of stored procedure that is executed automatically when a statement that modifies data is executed. You use triggers whenever referential integrity and other declarative constraints are insufficient. See "Ensuring data integrity" on page 65, and "CREATE TABLE statement" [*SQL Anywhere Server - SQL Reference*].

You may want to enforce a more complex form of referential integrity involving more detailed checking, or you may want to enforce checking on new data, but allow legacy data to violate constraints. Another use for triggers is in logging the activity on database tables, independent of the applications using the database.

Note
There are three special statements that triggers do not fire after: LOAD TABLE, TRUNCATE, and WRITETEXT. See "LOAD TABLE statement" [*SQL Anywhere Server - SQL Reference*], "TRUNCATE statement" [*SQL Anywhere Server - SQL Reference*], and "WRITETEXT statement [T-SQL]" [*SQL Anywhere Server - SQL Reference*].

Trigger execution permissions

Triggers execute with the permissions of the owner of the associated table or view, not the user ID whose actions cause the trigger to fire. A trigger can modify rows in a table that a user could not modify directly.

You can prevent triggers from being fired by specifying the -gf server option, or by setting the fire_triggers option. See:

- "-gf dbeng12/dbsrv12 server option" [*SQL Anywhere Server - Database Administration*]
- "fire_triggers option" [*SQL Anywhere Server - Database Administration*]

Trigger types

SQL Anywhere supports the following trigger types:

- **BEFORE trigger** A BEFORE trigger fires before a triggering action is performed. BEFORE triggers can be defined for tables, but not views.
- **AFTER trigger** An AFTER trigger fires after the triggering action is complete. AFTER triggers can be defined for tables, but not views.
- **INSTEAD OF trigger** An INSTEAD OF trigger is a conditional trigger that fires instead of the triggering action. INSTEAD OF triggers can be defined for tables and views (except materialized views). See "INSTEAD OF triggers" on page 810.

For a full description of the syntax for defining a trigger, see "CREATE TRIGGER statement" [*SQL Anywhere Server - SQL Reference*].

Trigger events

Triggers can be defined on one or more of the following triggering events:

Action	Description
INSERT	Invokes the trigger whenever a new row is inserted into the table associated with the trigger.
DELETE	Invokes the trigger whenever a row of the associated table is deleted.
UPDATE	Invokes the trigger whenever a row of the associated table is updated.
UPDATE OF *column-list*	Invokes the trigger whenever a row of the associated table is updated such that a column in the *column-list* is modified.

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. See "Trigger operation conditions" [*SQL Anywhere Server - SQL Reference*].

Trigger times

Triggers can be either **row-level** or **statement-level**:

- A row-level trigger executes once for each row that is changed. Row-level triggers execute BEFORE or AFTER the row is changed.

  Column values for the new and old images of the affected row are made available to the trigger via variables.

- A statement-level trigger executes after the entire triggering statement is completed. Rows affected by the triggering statement are made available to the trigger via temporary tables representing the new and old images of the rows. SQL Anywhere does not support statement-level BEFORE triggers.

Flexibility in trigger execution time is useful for triggers that rely on referential integrity actions such as cascaded updates or deletes being performed (or not) as they execute.

If an error occurs while a trigger is executing, the operation that fired the trigger fails. INSERT, UPDATE, and DELETE are atomic operations. When they fail, all effects of the statement (including the effects of triggers and any procedures called by triggers) revert to their preoperative state. See "Atomic compound statements" on page 817.

Creating triggers

You create triggers using either Sybase Central or Interactive SQL. In Sybase Central, you can use a wizard to provide necessary information. In Interactive SQL, you can use a CREATE TRIGGER statement. For both tools, you must have DBA or RESOURCE authority to create a trigger and you must have ALTER permissions on the table associated with the trigger.

The body of a trigger consists of a compound statement: a set of semicolon-delimited SQL statements bracketed by a BEGIN and an END statement.

You cannot use COMMIT and ROLLBACK and some ROLLBACK TO SAVEPOINT statements within a trigger.

To create a trigger for a given table (Sybase Central)

1. Use the SQL Anywhere 12 plug-in to connect to the database as a user with DBA or Resource authority.
2. In the left pane, click **Triggers**.
3. Choose **File** » **New** » **Trigger**.
4. Follow the instructions in the **Create Trigger Wizard**.
5. To complete the code, in the right pane click the **SQL** tab.

To create a trigger for a given table (SQL)

1. Connect to a database as a user with DBA authority. You must also have ALTER permissions on the table associated with the trigger.
2. Execute a CREATE TRIGGER statement.

Example 1: A row-level INSERT trigger

The following trigger is an example of a row-level INSERT trigger. It checks that the birth date entered for a new employee is reasonable:

```
CREATE TRIGGER check_birth_date
   AFTER INSERT ON Employees
REFERENCING NEW AS new_employee
FOR EACH ROW
BEGIN
     DECLARE err_user_error EXCEPTION
```

```
   FOR SQLSTATE '99999';
   IF new_employee.BirthDate > 'June 6, 2001' THEN
          SIGNAL err_user_error;
   END IF;
END;
```

Note
You may already have a trigger with the name check_birth_date in your SQL Anywhere sample database. If so, and you attempt to run the above SQL statement, an error is returned indicating that the trigger definition conflicts with existing triggers.

This trigger fires after any row is inserted into the Employees table. It detects and disallows any new rows that correspond to birth dates later than June 6, 2001.

The phrase REFERENCING NEW AS new_employee allows statements in the trigger code to refer to the data in the new row using the alias new_employee.

Signaling an error causes the triggering statement, and any previous trigger effects, to be undone.

For an INSERT statement that adds many rows to the Employees table, the check_birth_date trigger fires once for each new row. If the trigger fails for any of the rows, all effects of the INSERT statement roll back.

You can specify that the trigger fires before the row is inserted, rather than after, by changing the second line of the example to say :

```
BEFORE INSERT ON Employees
```

The REFERENCING NEW clause refers to the inserted values of the row; it is independent of the timing (BEFORE or AFTER) of the trigger.

Sometimes it is easier to enforce constraints using declarative referential integrity or CHECK constraints, rather than triggers. For example, implementing the above example with a column check constraint proves more efficient and concise:

```
CHECK (@col <= 'June 6, 2001')
```

Example 2: A row-level DELETE trigger example

The following CREATE TRIGGER statement defines a row-level DELETE trigger:

```
CREATE TRIGGER mytrigger
BEFORE DELETE ON Employees
REFERENCING OLD AS oldtable
FOR EACH ROW
BEGIN
   ...
END;
```

The REFERENCING OLD clause is independent of the timing (BEFORE or AFTER) of the trigger, and enables the delete trigger code to refer to the values in the row being deleted using the alias oldtable.

Example 3: A statement-level UPDATE trigger example

The following CREATE TRIGGER statement is appropriate for statement-level UPDATE triggers:


```
CREATE TRIGGER mytrigger AFTER UPDATE ON Employees
REFERENCING NEW AS table_after_update
            OLD AS table_before_update
FOR EACH STATEMENT
BEGIN
   ...
END;
```

The REFERENCING NEW and REFERENCING OLD clause allows the UPDATE trigger code to refer to both the old and new values of the rows being updated. The table alias table_after_update refers to columns in the new row and the table alias table_before_update refers to columns in the old row.

The REFERENCING NEW and REFERENCING OLD clause has a slightly different meaning for statement-level and row-level triggers. For statement-level triggers the REFERENCING OLD or NEW aliases are table aliases, while in row-level triggers they refer to the row being altered.

See also

- "SQL Anywhere database connections" [*SQL Anywhere Server - Database Administration*]
- "COMMIT statement" [*SQL Anywhere Server - SQL Reference*]
- "ROLLBACK TO SAVEPOINT statement" [*SQL Anywhere Server - SQL Reference*]
- "CREATE TRIGGER statement" [*SQL Anywhere Server - SQL Reference*]
- "Using compound statements" on page 816

Executing triggers

Triggers execute automatically whenever an INSERT, UPDATE, or DELETE operation is performed on the table named in the trigger. A row-level trigger fires once for each row affected, while a statement-level trigger fires once for the entire statement.

When an INSERT, UPDATE, or DELETE fires a trigger, the order of operation is as follows, depending on the trigger type (BEFORE or AFTER):

1. BEFORE triggers fire.
2. The operation itself is performed.
3. Referential actions are performed.
4. AFTER triggers fire.

Note
When creating a trigger using the CREATE TRIGGER statement, if a trigger-type is not specified, the default is AFTER.

If any of the steps encounter an error not handled within a procedure or trigger, the preceding steps are undone, the subsequent steps are not performed, and the operation that fired the trigger fails.

Altering triggers

You can modify an existing trigger using either Sybase Central or Interactive SQL. You must be the owner of the table on which the trigger is defined, or be DBA, or have ALTER permissions on the table and have RESOURCE authority.

In Sybase Central, you cannot rename an existing trigger directly. Instead, you must create a new trigger with the new name, copy the previous code to it, and then delete the old trigger.

Alternatively, you can use an ALTER TRIGGER statement to modify an existing trigger. You must include the entire new trigger in this statement (in the same syntax as in the CREATE TRIGGER statement that created the trigger).

To alter the code of a trigger (Sybase Central)

1. Use the SQL Anywhere 12 plug-in to connect to the database as a user with DBA authority or as the owner of the trigger.
2. In the left pane, double-click **Triggers**.
3. Select the trigger.
4. Use one of the following methods to alter the trigger:
   - In the right pane, click the **SQL** tab.
   - Right-click the trigger and choose **Edit In New Window**.

     Tip
     You can open a separate window for each procedure and copy code between triggers.

   - To add or edit a procedure comment, right-click the trigger and choose **Properties**.

     If you use the **Database Documentation Wizard** to document your SQL Anywhere database, you will have the option to include these comments in the output. See "Documenting a database" [*SQL Anywhere Server - Database Administration*].

To alter the code of a trigger (SQL)

1. Connect to the database as a user with DBA authority or as the owner of the trigger.
2. Execute an ALTER TRIGGER statement. Include the entire new trigger in this statement.

See also

- "Set properties for database objects" on page 1
- "SQL Anywhere database connections" [*SQL Anywhere Server - Database Administration*]
- "Using Sybase Central to translate stored procedures" on page 659
- "ALTER TRIGGER statement" [*SQL Anywhere Server - SQL Reference*]

Dropping triggers

Once you create a trigger, it remains in the database until someone explicitly removes it. You must have ALTER permissions on the table associated with the trigger to drop the trigger.

To delete a trigger (Sybase Central)

1. Use the SQL Anywhere 12 plug-in to connect to the database as a user with DBA authority or as the owner of the trigger.
2. In the left pane, double-click **Triggers**.
3. Select the trigger and choose **Edit** » **Delete**.
4. Click **Yes**.

To delete a trigger (SQL)

1. Connect to a database as a user with DBA authority or as the owner of the trigger.
2. Execute a DROP TRIGGER statement.

Example

The following statement removes the mytrigger trigger from the database:

```
DROP TRIGGER mytrigger;
```

See also

- "SQL Anywhere database connections" [*SQL Anywhere Server - Database Administration*]
- "DROP TRIGGER statement" [*SQL Anywhere Server - SQL Reference*]

Trigger execution permissions

You cannot grant permissions to execute a trigger, since users cannot execute triggers: SQL Anywhere fires them in response to actions on the database. Nevertheless, a trigger does have permissions associated with it as it executes, defining its right to perform certain actions.

Triggers execute using the permissions of the owner of the table on which they are defined, not the permissions of the user who caused the trigger to fire, and not the permissions of the user who created the trigger.

When a trigger refers to a table, it uses the group memberships of the table creator to locate tables with no explicit owner name specified. For example, if a trigger on user_1.Table_A references Table_B and does not specify the owner of Table_B, then either Table_B must have been created by user_1 or user_1 must be a member of a group (directly or indirectly) that is the owner of Table_B. If neither condition is met, the database server returns a message, when the trigger fires, indicating that the table cannot be found.

Also, user_1 must have permissions to perform the operations specified in the trigger.

See also

- "Database permissions and authorities" [*SQL Anywhere Server - Database Administration*]

Advanced information on triggers

One aspect of triggers that can be difficult to understand is the order in which triggers fire if several triggers are impacted by the same triggering action. Whether competing triggers are fired, and the order in which they are fired, depends on two things: trigger type (BEFORE, INSTEAD OF, or AFTER), and trigger scope (row-level or statement-level).

For row-level triggers, BEFORE triggers fire before INSTEAD OF triggers, which fire before AFTER triggers. All row-level triggers for a given row fire before any triggers fire for a subsequent row.

For statement-level triggers, INSTEAD OF triggers fire before AFTER triggers. Statement-level BEFORE triggers are not supported.

If there are competing statement-level and row-level AFTER triggers, the statement-level AFTER triggers fire after all row-level triggers have completed.

If there are competing statement-level and row-level INSTEAD OF triggers, the row-level triggers do not fire.

INSTEAD OF triggers

INSTEAD OF triggers differ from BEFORE and AFTER triggers because when an INSTEAD OF trigger fires, the triggering action is skipped and the specified action is performed instead.

The following is a list of capabilities and restrictions that are unique to INSTEAD OF triggers:

- There can only be one INSTEAD OF trigger for each trigger event on a given table.
- INSTEAD OF triggers can be defined for a table or a view. However, INSTEAD OF triggers cannot be defined on materialized views since you cannot execute DML operations, such as INSERT, DELETE, and UPDATE statements, on materialized views.
- You cannot specify the ORDER or WHEN clauses when defining an INSTEAD OF trigger.
- You cannot define an INSTEAD OF trigger for an UPDATE OF *column-list* trigger event. See "CREATE TRIGGER statement" [*SQL Anywhere Server - SQL Reference*].
- Whether an INSTEAD OF trigger performs recursion depends on whether the target of the trigger is a base table or a view. Recursion occurs for views, but not for base tables. That is, if an INSTEAD OF trigger performs DML operations on the base table on which the trigger is defined, those operations do not cause triggers to fire (including BEFORE or AFTER triggers). If the target is a view, all triggers fire for the operations performed on the view.

- If a table has an INSTEAD OF trigger defined on it, you cannot execute an INSERT statement with an ON EXISTING clause against the table. Attempting to do so returns a SQLE_INSTEAD_TRIGGER error.
- You cannot execute an INSERT statement against a view that was defined with the WITH CHECK OPTION (or is nested inside another view that was defined this way), and that has an INSTEAD OF INSERT trigger defined against it. This is true for UPDATE and DELETE statements as well. Attempting to do so returns a SQLE_CHECK_TRIGGER_CONFLICT error.
- If an INSTEAD OF trigger is fired as a result of a positioned update, positioned delete, PUT statement, or wide insert operation, a SQLE_INSTEAD_TRIGGER_POSITIONED error is returned.

Updating non-updatable views using INSTEAD OF triggers

INSTEAD OF triggers allow you to execute INSERT, UPDATE, or DELETE statements against a view that is not inherently updatable. The body of the trigger defines what it means to execute the corresponding INSERT, UPDATE, or DELETE statement. For example, suppose you create the following view:

```
CREATE VIEW V1 ( Surname, GivenName, State )
   AS SELECT DISTINCT Surname, GivenName, State
        FROM Contacts;
```

You cannot delete rows from V1 because the DISTINCT keyword makes V1 not inherently updatable. In other words, the database server cannot unambiguously determine what it means to delete a row from V1. However, you could define an INSTEAD OF DELETE trigger that implements a delete operation on V1. For example, the following trigger deletes all rows from Contacts with a given Surname, GivenName, and State when that row is deleted from V1:

```
CREATE TRIGGER V1_Delete
  INSTEAD OF DELETE ON V1
  REFERENCING OLD AS old_row
  FOR EACH ROW
BEGIN
    DELETE FROM Contacts
      WHERE Surname = old_row.Surname
        AND GivenName = old_row.GivenName
        AND State = old_row.State
END;
```

Once the V1_Delete trigger is defined, you can delete rows from V1. You can also define other INSTEAD OF triggers to allow INSERT and UPDATE statements to be performed on V1.

If a view with an INSTEAD OF DELETE trigger is nested in another view, it is treated like a base table for checking the updatability of a DELETE. This is true for INSERT and UPDATE operations as well. Continuing from the previous example, create another view:

```
CREATE VIEW V2 ( Surname, GivenName ) AS
  SELECT Surname, GivenName from V1;
```

Without the V1_Delete trigger, you cannot delete rows from V2 because V1 is not inherently updatable, so neither is V2. However, if you define an INSTEAD OF DELETE trigger on V1, you can delete rows from V2. Each row deleted from V2 results in a row being deleted from V1, which causes the V1_Delete trigger to fire.

Be careful when defining an INSTEAD OF trigger on a nested view, since the firing of the trigger can have unintended consequences. To make the intended behavior explicit, define the INSTEAD OF triggers on any view referencing the nested view.

The following trigger could be defined on V2 to cause the desired behavior for a DELETE statement:

```
CREATE TRIGGER V2_Delete
  INSTEAD OF DELETE ON V2
  REFERENCING OLD AS old_row
  FOR EACH ROW
BEGIN
    DELETE FROM Contacts
      WHERE Surname = old_row.Surname
        AND GivenName = old_row.GivenName
END;
```

The V2_Delete trigger ensures that the behavior of a delete operation on V2 remains the same, even if the INSTEAD OF DELETE trigger on V1 is removed or changed.

Introduction to batches

A batch is a set of SQL statements submitted together and executed as a group, one after the other. The control statements used in procedures (CASE, IF, LOOP, and so on) can also be used in batches. If the batch consists of a compound statement enclosed in a BEGIN/END, then it can also contain host variables, local declarations for variables, cursors, temporary tables and exceptions. Host variable references are permitted within batches with the following restrictions:

- only one statement in the batch can refer to host variables
- the statement which uses host variables cannot be preceded by a statement which returns a result set

Use of BEGIN/END is recommended to clearly indicate when a batch is being used.

Statements within the batch may be delimited with semicolons, in which case the batch is conforming to the Watcom SQL dialect. A multi-statement batch that does not use semicolons to delimit statements conforms to the Transact-SQL dialect. The dialect of the batch determines which statements are permitted within the batch, and also determines how errors within the batch are handled. For more information about Transact-SQL batches, see “Transact-SQL batch overview” on page 658.

In many ways, batches are similar to stored procedures; however, there are some differences:

- batches do not have names
- batches do not accept parameters
- batches are not stored persistently in the database
- batches cannot be shared by different connections

A simple batch consists of a set of SQL statements with no delimiters followed by a separate line with just the word go on it. The following example creates an Eastern Sales department and transfers all sales reps from Massachusetts to that department. It is an example of a Transact-SQL batch.

```
INSERT
INTO Departments ( DepartmentID, DepartmentName )
VALUES ( 220, 'Eastern Sales' )

UPDATE Employees
SET DepartmentID = 220
WHERE DepartmentID = 200
AND State = 'MA'

COMMIT
go
```

The word go is recognized by Interactive SQL and causes it to send the previous statements as a single batch to the server. See "Executing multiple SQL statements" [*SQL Anywhere Server - Database Administration*].

The following example, while similar in appearance, is handled quite differently by Interactive SQL. This example does not use the Transact-SQL dialect. Each statement is delimited by a semicolon. Interactive SQL sends each semicolon-delimited statement separately to the server. It is not treated as a batch.

```
INSERT
INTO Departments ( DepartmentID, DepartmentName )
VALUES ( 220, 'Eastern Sales' );

UPDATE Employees
SET DepartmentID = 220
WHERE DepartmentID = 200
AND State = 'MA';

COMMIT;
```

To have Interactive SQL treat it as a batch, it can be changed into a compound statement using `BEGIN ... END`. The following is a revised version of the previous example. The three statements in the compound statement are sent as a batch to the server.

```
BEGIN
  INSERT
  INTO Departments ( DepartmentID, DepartmentName )
  VALUES ( 220, 'Eastern Sales' );

  UPDATE Employees
  SET DepartmentID = 220
  WHERE DepartmentID = 200
  AND State = 'MA';

  COMMIT;
END
```

In this particular example, it makes no difference to the end result whether a batch or individual statements are executed by the server. There are situations, though, where it can make a difference. Consider the following example.

```
DECLARE @CurrentID INTEGER;
SET @CurrentID = 207;
```

```
SELECT Surname FROM Employees
  WHERE EmployeeID=@CurrentID;
```

If you execute this example using Interactive SQL, the database server returns an error indicating that the variable cannot be found. This happens because Interactive SQL sends three separate statements to the server. They are not executed as a batch. As you have already seen, the remedy is to use a compound statement to force Interactive SQL to send these statements as a batch to the server. The following example accomplishes this.

```
BEGIN
  DECLARE @CurrentID INTEGER;
  SET @CurrentID = 207;
  SELECT Surname FROM Employees
    WHERE EmployeeID=@CurrentID;
END
```

Putting a BEGIN and END around a set of statements forces Interactive SQL to treat them as a batch.

The IF statement is another example of a compound statement. Interactive SQL sends the following statements as a single batch to the server.

```
IF EXISTS(   SELECT *
             FROM SYSTAB
             WHERE table_name='Employees' )
THEN
   SELECT   Surname AS LastName,
            GivenName AS FirstName
   FROM Employees;
   SELECT Surname, GivenName
   FROM Customers;
   SELECT Surname, GivenName
   FROM Contacts;
ELSE
   MESSAGE 'The Employees table does not exist'
   TO CLIENT;
END IF
```

This situation does not arise when using other techniques to prepare and execute SQL statements. For example, an application that uses ODBC can prepare and execute a series of semicolon-separated statements as a batch.

Care must be exercised when mixing Interactive SQL statements with SQL statements intended for the server. The following is an example of how mixing Interactive SQL statements and SQL statements can be an issue. In this example, since the Interactive SQL OUTPUT statement is embedded in the compound statement, it is sent along with all the other statements to the server as a batch, and results in a syntax error.

```
IF EXISTS(   SELECT *
             FROM SYSTAB
             WHERE table_name='Employees' )
THEN
   SELECT   Surname AS LastName,
            GivenName AS FirstName
   FROM Employees;
   SELECT Surname, GivenName
   FROM Customers;
   SELECT Surname, GivenName
   FROM Contacts;
   OUTPUT TO 'c:\\temp\\query.txt';
```



```
ELSE
   MESSAGE 'The Employees table does not exist'
   TO CLIENT;
END IF
```

The correct placement of the OUTPUT statement is shown below.

```
IF EXISTS(   SELECT *
             FROM SYSTAB
             WHERE table_name='Employees' )
THEN
   SELECT    Surname AS LastName,
             GivenName AS FirstName
   FROM Employees;
   SELECT Surname, GivenName
   FROM Customers;
   SELECT Surname, GivenName
   FROM Contacts;
ELSE
   MESSAGE 'The Employees table does not exist'
   TO CLIENT;
END IF;
OUTPUT TO 'c:\\temp\\query.txt';
```

Control statements

There are several control statements for logical flow and decision making in the body of the procedure or trigger, or in a batch. Available control statements include:

Control statement	Syntax

Compound statements
 	
See "BEGIN statement" [*SQL Anywhere Server - SQL Reference*].
	`BEGIN [ ATOMIC ]`
	`   Statement-list`
	`END`

Conditional execution: IF
 	
See "IF statement" [*SQL Anywhere Server - SQL Reference*].
	`IF condition THEN`
	`   Statement-list`
	`ELSEIF condition THEN`
	`   Statement-list`
	`ELSE`
	`   Statement-list`
	`END IF`

Conditional execution: CASE
 	
See "CASE statement" [*SQL Anywhere Server - SQL Reference*].
	`CASE expression`
	`WHEN value THEN`
	`   Statement-list`
	`WHEN value THEN`
	`   Statement-list`
	`ELSE`
	`   Statement-list`
	`END CASE`

Control statement	Syntax

Repetition: WHILE, LOOP
 	
See "LOOP statement" [*SQL Anywhere Server - SQL Reference*].
	`WHILE condition LOOP`
	`  Statement-list`
	`END LOOP`

Repetition: FOR cursor loop
 	
See "FOR statement" [*SQL Anywhere Server - SQL Reference*].
	`FOR loop-name`
	`  AS cursor-name CURSOR`
	`FOR`
	`  select-statement`
	`DO`
	`  Statement-list`
	`END FOR`

Break: LEAVE See "LEAVE statement" [*SQL Anywhere Server - SQL Reference*].	`LEAVE label`
CALL See "CALL statement" [*SQL Anywhere Server - SQL Reference*].	`CALL procname( arg, ... )`

Using compound statements

A compound statement starts with the keyword BEGIN and concludes with the keyword END. The body of a procedure or trigger is a **compound statement**. Compound statements can also be used in batches. Compound statements can be nested, and combined with other control statements to define execution flow in procedures and triggers or in batches.

A compound statement allows a set of SQL statements to be grouped together and treated as a unit. Delimit SQL statements within a compound statement with semicolons.

For more information about compound statements, see "BEGIN statement" [*SQL Anywhere Server - SQL Reference*].

Declarations in compound statements

Local declarations in a compound statement immediately follow the BEGIN keyword. These local declarations exist only within the compound statement. Within a compound statement you can declare:

- Variables
- Cursors

- Temporary tables
- Exceptions (error identifiers)

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 called from the compound statement.

Atomic compound statements

An **atomic** statement is a statement that is 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 changed rows revert back to their original state. The UPDATE statement is atomic.

All non-compound SQL statements are atomic. You can make a compound statement atomic by adding the keyword ATOMIC after the BEGIN keyword.

```
BEGIN ATOMIC
   UPDATE Employees
   SET ManagerID = 501
   WHERE EmployeeID = 467;
    UPDATE Employees
   SET BirthDate = 'bad_data';
END
```

In this example, the two update statements are part of an atomic compound statement. They must either succeed or fail as one. The first update statement would succeed. The second one causes a data conversion error since the value being assigned to the BirthDate column cannot be converted to a date.

The atomic compound statement fails and the effect of both UPDATE statements is undone. Even if the currently executing transaction is eventually committed, neither statement in the atomic compound statement takes effect.

If an atomic compound statement succeeds, the changes made within the compound statement take effect only if the currently executing transaction is committed. In the case when an atomic compound statement succeeds but the transaction in which it occurs gets rolled back, the atomic compound statement also gets rolled back. A savepoint is established at the start of the atomic compound statement. Any errors within the statement result in a rollback to that savepoint.

When an atomic compound statement is executed in autocommit (unchained) mode, the commit mode changes to manual (chained) until statement execution is complete. In manual mode, DML statements executed within the atomic compound statement do not cause an immediate COMMIT or ROLLBACK. If the atomic compound statement completes successfully, a COMMIT statement is executed; otherwise, a ROLLBACK statement is executed. For more information about autocommit behavior, see “Setting autocommit or manual commit mode” [*SQL Anywhere Server - Programming*] and “Controlling autocommit behavior” [*SQL Anywhere Server - Programming*].

You cannot use COMMIT and ROLLBACK and some ROLLBACK TO SAVEPOINT statements within an atomic compound statement. See "Transactions and savepoints in procedures and triggers" on page 836.

There is a case where some, but not all, statements within an atomic compound statement are executed. This happens when an exception handler within the compound statement deals with an error.

For more information, see "Using exception handlers in procedures and triggers" on page 833.

The structure of procedures and triggers

The body of a procedure or trigger consists of a compound statement as discussed in "Using compound statements" on page 816. A compound statement consists of a BEGIN and an END, enclosing a set of SQL statements. Semicolons delimit each statement.

Declaring parameters for procedures

Procedure parameters appear as a list in the CREATE PROCEDURE statement. Parameter names must conform to the rules for other database identifiers such as column names. They must have valid data types, and can be prefixed with one of the keywords IN, OUT or INOUT. By default, parameters are INOUT parameters. These keywords have the following meanings:

- **IN** The argument is an expression that provides a value to the procedure.
- **OUT** The argument is a variable that could be given a value by the procedure.
- **INOUT** The argument is a variable that provides a value to the procedure, and could be given a new value by the procedure.

You can assign default values to procedure parameters in the CREATE PROCEDURE statement. The default value must be a constant, which may be NULL. For example, the following procedure uses the NULL default for an IN parameter to avoid executing a query that would have no meaning:

```
CREATE PROCEDURE CustomerProducts(
      IN customer_ID
                  INTEGER DEFAULT NULL )
RESULT ( product_ID INTEGER,
         quantity_ordered INTEGER )
BEGIN
   IF 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;
```

The following statement assigns the DEFAULT NULL, and the procedure RETURNs instead of executing the query.

```
CALL CustomerProducts();
```

See also

- "SQL data types" [*SQL Anywhere Server - SQL Reference*]

Passing parameters to procedures

You can take advantage of default values of stored procedure parameters with either of two forms of the CALL statement.

If the optional parameters are at the end of the argument list in the CREATE PROCEDURE statement, they may be omitted from the CALL statement. As an example, consider a procedure with three INOUT parameters:

```
CREATE PROCEDURE SampleProcedure(
       INOUT var1 INT DEFAULT 1,
                  INOUT var2 int DEFAULT 2,
                  INOUT var3 int DEFAULT 3 )
...
```

This example assumes that the calling environment has set up three variables to hold the values passed to the procedure:

```
CREATE VARIABLE V1 INT;
CREATE VARIABLE V2 INT;
CREATE VARIABLE V3 INT;
```

The procedure SampleProcedure may be called supplying only the first parameter as follows:

```
CALL SampleProcedure( V1 );
```

in which case the default values are used for *var2* and *var3*.

A more flexible method of calling procedures with optional arguments is to pass the parameters by name. The SampleProcedure procedure may be called as follows:

```
CALL SampleProcedure( var1 = V1, var3 = V3 );
```

or as follows:

```
CALL SampleProcedure( var3 = V3, var1 = V1 );
```

Passing parameters to functions

User-defined functions are not invoked with the CALL statement, but are used in the same manner that built-in functions are. For example, the following statement uses the FullName function defined in "Creating user-defined functions" on page 800 to retrieve the names of employees:

To list the names of all employees

- In Interactive SQL, type the following:

```
SELECT FullName( GivenName, Surname ) AS Name
   FROM Employees;
```

Name
Fran Whitney
Matthew Cobb
Philip Chin
Julie Jordan
...

Notes

- Default parameters can be used in calling functions. However, parameters cannot be passed to functions by name.
- Parameters are passed by value, not by reference. Even if the function changes the value of the parameter, this change is not returned to the calling environment.
- Output parameters cannot be used in user-defined functions.
- User-defined functions cannot return result sets.

Returning results from procedures

Procedures can return results in the form of a single row of data, or multiple rows. Results consisting of a single row of data can be passed back as arguments to the procedure. Results consisting of multiple rows of data are passed back as result sets. Procedures can also return a single value given in the RETURN statement.

For simple examples of how to return results from procedures, see "Introduction to procedures" on page 794.

Returning a value using the RETURN statement

The RETURN statement returns a single integer value to the calling environment, causing an immediate exit from the procedure. The RETURN statement takes the form:

```
RETURN expression
```

The value of the supplied expression is returned to the calling environment. To save the return value in a variable, use an extension of the CALL statement:

```
CREATE VARIABLE returnval INTEGER;
returnval = CALL myproc();
```

See also

- "RETURN statement" [*SQL Anywhere Server - SQL Reference*]

Returning results as procedure parameters

Procedures can return results to the calling environment in the parameters to the procedure.

Within a procedure, parameters and variables can be assigned values using:

- the SET statement.
- a SELECT statement with an INTO clause.

Using the SET statement

The following procedure returns a value in an OUT parameter assigned using a SET statement:

```
CREATE PROCEDURE greater(
   IN a INT,
   IN b INT,
   OUT c INT )
BEGIN
   IF a > b THEN
      SET c = a;
   ELSE
      SET c = b;
   END IF ;
END;
```

Using single-row SELECT statements

Single-row queries retrieve at most one row from the database. This type of query uses a SELECT statement with an INTO clause. The INTO clause follows the select list and precedes the FROM clause. It contains a list of variables to receive the value for each select list item. There must be the same number of variables as there are select list items.

When a SELECT statement executes, the server retrieves the results of the SELECT statement and places the results in the variables. If the query results contain more than one row, the server returns an error. For queries returning more than one row, you must use cursors. For information about returning more than one row from a procedure, see "Returning result sets from procedures" on page 822.

If the query results in no rows being selected, the variables are not updated, and a warning is returned.

The following procedure returns the results of a single-row SELECT statement in the procedure parameters.

To return the number of orders placed by a given customer

- Type the following:

```
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;
```

You can test this procedure in Interactive SQL using the following statements, which show the number of orders placed by the customer with ID 102:

```
CREATE VARIABLE orders INT;
CALL OrderCount ( 102, orders );
SELECT orders;
```

Notes

- The customer_ID parameter is declared as an IN parameter. This parameter holds the customer ID passed in to the procedure.
- The Orders parameter is declared as an OUT parameter. It holds the value of the orders variable returned to the calling environment.
- No DECLARE statement is necessary for the Orders variable, as it is declared in the procedure argument list.
- The SELECT statement returns a single row and places it into the variable Orders.

Returning result sets from procedures

Result sets allow a procedure to return more than one row of results to the calling environment.

The following procedure returns a list of customers who have placed orders, together with the total value of the orders placed. The procedure does not list customers who have not placed orders.

```
CREATE PROCEDURE ListCustomerValue()
RESULT ("Company" CHAR(36), "Value" INT)
BEGIN
   SELECT CompanyName,
      CAST( sum(  SalesOrderItems.Quantity *
                  Products.UnitPrice)
                  AS INTEGER ) AS value
   FROM Customers
      INNER JOIN SalesOrders
      INNER JOIN SalesOrderItems
      INNER JOIN Products
```



```
    GROUP BY CompanyName
    ORDER BY value DESC;
END;
```

- Run the following statement:

```
CALL ListCustomerValue ( );
```

Company	Value
The Hat Company	5016
The Igloo	3564
The Ultimate	3348
North Land Trading	3144
Molly's	2808
...	...

Notes

- The number of variables in the RESULT list must match the number of the SELECT list items. Automatic data type conversion is performed where possible if data types do not match.
- The RESULT clause is part of the CREATE PROCEDURE statement, and does not have a command delimiter.
- The names of the SELECT list items do not need to match those of the RESULT list.
- When testing this procedure, Interactive SQL displays only the first result set by default. You can configure Interactive SQL to display more than one result set by setting the **Show Multiple Result Sets** option on the **Results** tab of the **Options** window.
- You can modify procedure result sets, unless they are generated from a view. The user calling the procedure requires the appropriate permissions on the underlying table to modify procedure results. This is different than the usual permissions for procedure execution, where the procedure owner must have permissions on the table. See "Editing result sets in Interactive SQL" [*SQL Anywhere Server - Database Administration*].
- If a stored procedure or user-defined function returns a result, it cannot also set output parameters or return a return value.

Returning multiple result sets from procedures

Before Interactive SQL can return multiple result sets, you need to enable this option on the **Results** tab of the **Options** window. By default, this option is disabled. If you change the setting, it takes effect in newly created connections (such as new windows).

To enable multiple result set functionality (Interactive SQL)

1. Connect to the database as a user with DBA authority.
2. In Interactive SQL, choose **Tools** » **Options**.
3. Click **SQL Anywhere**.
4. On the **Results** tab, select **Show All Result Sets**.
5. Click **OK**.

To enable multiple result set functionality (SQL)

1. Connect to the database as a user with DBA authority.
2. Run the following:

```
SET OPTION isql_show_multiple_result_sets=On
```

After you enable this option, a procedure can return more than one result set to the calling environment. If a RESULT clause is employed, the result sets must be compatible: they must have the same number of items in the SELECT lists, and the data types must all be of types that can be automatically converted to the data types listed in the RESULT list.

Example

The following procedure lists the names of all employees, customers, and contacts listed in the database:

```
CREATE PROCEDURE ListPeople()
RESULT ( Surname CHAR(36), GivenName CHAR(36) )
BEGIN
   SELECT Surname, GivenName
   FROM Employees;
   SELECT Surname, GivenName
   FROM Customers;
   SELECT Surname, GivenName
   FROM Contacts;
END;
```

To test this procedure and view multiple result sets in Interactive SQL, enter the following statement in the SQL Statements pane:

```
CALL ListPeople ();
```

Returning variable result sets from procedures

The RESULT clause is optional in procedures. Omitting the result clause allows you to write procedures that return different result sets, with different numbers or types of columns, depending on how they are executed.

If you do not use the variable result sets feature, you should use a RESULT clause for performance reasons.

For example, the following procedure returns two columns if the input variable is Y, but only one column otherwise:

```
CREATE PROCEDURE Names( IN formal char(1) )
BEGIN
   IF formal = 'y' THEN
      SELECT Surname, GivenName
      FROM Employees
   ELSE
      SELECT GivenName
      FROM Employees
   END IF
END;
```

When you create a procedure without a RESULT clause and the procedure returns a variable result set, a DESCRIBE of the SELECT statement referencing the procedure may fail. To prevent the failure of the DESCRIBE, it is recommended that you include a WITH clause that describes the expected result set schema.

The use of variable result sets in procedures is subject to some limitations, depending on the interface used by the client application.

- **Embedded SQL** To get the proper shape of result set, you must DESCRIBE the procedure call after the cursor for the result set is opened, but before any rows are returned.

  For more information about the DESCRIBE statement, see "DESCRIBE statement [Interactive SQL]" [*SQL Anywhere Server - SQL Reference*].

- **ODBC** Variable result set procedures can be used by ODBC applications. The SQL Anywhere ODBC driver performs the proper description of the variable result sets.

- **Open Client applications** Open Client applications can use variable result set procedures. SQL Anywhere performs the proper description of the variable result sets.

Using cursors in procedures and triggers

Cursors retrieve rows one at a time from a query or stored procedure with multiple rows in its result set. A cursor is a handle or an identifier for the query or procedure, and for a current position within the result set.

Cursor management overview

Managing a cursor is similar to managing a file in a programming language. The following steps manage cursors:

1. Declare a cursor for a particular SELECT statement or procedure using the DECLARE statement.
2. Open the cursor using the OPEN statement.
3. Use the FETCH statement to retrieve results one row at a time from the cursor.

4. The warning `Row Not Found` signals the end of the result set.

5. Close the cursor using the CLOSE statement.

By default, cursors are automatically closed at the end of a transaction (on COMMIT or ROLLBACK statements). Cursors opened using the WITH HOLD clause will stay open for subsequent transactions until explicitly closed.

For more information about positioning cursors, see "Cursor positioning" [*SQL Anywhere Server - Programming*].

Using cursors on SELECT statements in procedures

The following procedure uses a cursor on a SELECT statement. Based on the same query used in the ListCustomerValue procedure described in "Returning result sets from procedures" on page 822, it illustrates several features of the stored procedure language.

```
CREATE PROCEDURE TopCustomerValue(
      OUT TopCompany CHAR(36),
      OUT TopValue INT )
BEGIN
   -- 1. Declare the "row not found" exception
   DECLARE err_notfound
      EXCEPTION FOR SQLSTATE '02000';
   -- 2.  Declare variables to hold
   --     each company name and its value
   DECLARE ThisName CHAR(36);
   DECLARE ThisValue INT;
   -- 3.  Declare the cursor ThisCompany
   --      for the query
   DECLARE ThisCompany CURSOR FOR
   SELECT CompanyName,
         CAST( sum( SalesOrderItems.Quantity *
               Products.UnitPrice ) AS INTEGER )
         AS value
   FROM Customers
      INNER JOIN SalesOrders
      INNER JOIN SalesOrderItems
      INNER JOIN Products
   GROUP BY CompanyName;
   -- 4. Initialize the values of TopValue
   SET TopValue = 0;
   -- 5. Open the cursor
   OPEN ThisCompany;
   -- 6. Loop over the rows of the query
   CompanyLoop:
   LOOP
      FETCH NEXT ThisCompany
         INTO ThisName, ThisValue;
      IF SQLSTATE = err_notfound THEN
         LEAVE CompanyLoop;
      END IF;
      IF ThisValue > TopValue THEN
         SET TopCompany = ThisName;
         SET TopValue = ThisValue;
      END IF;
   END LOOP CompanyLoop;
```



```
   -- 7. Close the cursor
   CLOSE ThisCompany;
END;
```

Notes

The TopCustomerValue procedure has the following notable features:

- The `Row Not Found` exception is declared. This exception signals, later in the procedure, when a loop over the results of a query completes.

  For more information about exceptions, see "Errors and warnings in procedures and triggers" on page 828.

- Two local variables ThisName and ThisValue are declared to hold the results from each row of the query.
- The cursor ThisCompany is declared. The SELECT statement produces a list of company names and the total value of the orders placed by that company.
- The value of TopValue is set to an initial value of 0, for later use in the loop.
- The ThisCompany cursor opens.
- The LOOP statement loops over each row of the query, placing each company name in turn into the variables ThisName and ThisValue. If ThisValue is greater than the current top value, TopCompany and TopValue are reset to ThisName and ThisValue.
- The cursor closes at the end of the procedure.
- You can also write this procedure without a loop by adding an ORDER BY value DESC clause to the SELECT statement. Then, only the first row of the cursor needs to be fetched.

The LOOP construct in the TopCompanyValue procedure is a standard form, exiting after the last row is processed. You can rewrite this procedure in a more compact form using a FOR loop. The FOR statement combines several aspects of the above procedure into a single statement.

```
CREATE PROCEDURE TopCustomerValue2(
      OUT TopCompany CHAR(36),
      OUT TopValue INT )
BEGIN
   -- 1. Initialize the TopValue variable
   SET TopValue = 0;
   -- 2. Do the For Loop
   FOR CompanyFor AS ThisCompany
      CURSOR FOR
      SELECT CompanyName AS ThisName,
         CAST( sum( SalesOrderItems.Quantity *
               Products.UnitPrice ) AS INTEGER )
         AS ThisValue
      FROM Customers
         INNER JOIN SalesOrders
         INNER JOIN SalesOrderItems
         INNER JOIN Products
      GROUP BY ThisName
   DO
      IF ThisValue > TopValue THEN
         SET TopCompany = ThisName;
```

```
            SET TopValue = ThisValue;
        END IF;
    END FOR;
END;
```

Updating a cursor inside a stored procedure

The following procedure uses an updatable cursor on a SELECT statement. It illustrates how to perform an UPDATE on a row using the stored procedure language.

```
CREATE PROCEDURE UpdateSalary(
  IN employeeIdent INT,
  IN salaryIncrease NUMERIC(10,3) )
BEGIN
-- Procedure to increase (or decrease) an employee's salary
  DECLARE err_notfound
      EXCEPTION FOR SQLSTATE '02000';
  DECLARE oldSalary NUMERIC(20,3);
  DECLARE employeeCursor
    CURSOR FOR SELECT Salary from Employees
               WHERE EmployeeID = employeeIdent
    FOR UPDATE;
  OPEN employeeCursor;
  FETCH employeeCursor INTO oldSalary FOR UPDATE;
  IF SQLSTATE = err_notfound THEN
    MESSAGE 'No such employee' TO CLIENT;
  ELSE
    UPDATE Employees SET Salary = oldSalary + salaryIncrease
      WHERE CURRENT OF employeeCursor;
  END IF;
  CLOSE employeeCursor;
END;
```

The following statement calls the above stored procedure:

```
CALL UpdateSalary( 105, 220.00 );
```

Errors and warnings in procedures and triggers

After an application program executes a SQL statement, it can examine a **status code**. This status code (or return code) indicates whether the statement executed successfully or failed and gives the reason for the failure. You can use the same mechanism to indicate the success or failure of a CALL statement to a procedure.

Error reporting uses either the SQLCIt is often desirable to intercept certainODE or SQLSTATE status descriptions. For full descriptions of SQLCODE and SQLSTATE error and warning values and their meanings, see “Error Messages”.

Whenever a SQL statement executes, a value appears in special procedure variables called SQLSTATE and SQLCODE. The special value indicates whether there were any unusual conditions encountered when the statement was executed. You can check the value of SQLSTATE or SQLCODE in an IF statement following a SQL statement, and take actions depending on whether the statement succeeded or failed.

For example, the SQLSTATE variable can be used to indicate if a row is successfully fetched. The TopCustomerValue procedure presented in section "Using cursors on SELECT statements in procedures" on page 826 used the SQLSTATE test to detect that all rows of a SELECT statement had been processed.

Default error handling in procedures and triggers

This section describes how SQL Anywhere handles errors that occur during a procedure execution, if you have no error handling built in to the procedure.

For different behavior, you can use exception handlers, described in "Using exception handlers in procedures and triggers" on page 833.

Warnings are handled in a slightly different manner from errors: for a description, see "Default handling of warnings in procedures and triggers" on page 832.

There are two ways of handling errors without using explicit error handling:

- **Default error handling** The procedure or trigger fails and returns an error code to the calling environment.
- **ON EXCEPTION RESUME** If the ON EXCEPTION RESUME clause appears in the CREATE PROCEDURE statement, the procedure carries on executing after an error, resuming at the statement following the one causing the error.

  The precise behavior for procedures that use ON EXCEPTION RESUME is dictated by the on_tsql_error option setting. See "on_tsql_error option" [*SQL Anywhere Server - Database Administration*].

Default error handling

Generally, if a SQL statement in a procedure or trigger fails, the procedure or trigger stops executing and control returns to the application program with an appropriate setting for the SQLSTATE and SQLCODE values. This is true even if the error occurred in a procedure or trigger invoked directly or indirectly from the first one. For triggers the operation causing the trigger is also undone and the error is returned to the application.

The following demonstration procedures show what happens when an application calls the procedure OuterProc, and OuterProc in turn calls the procedure InnerProc, which then encounters an error.

```
CREATE PROCEDURE OuterProc()
BEGIN
   MESSAGE 'Hello from OuterProc.' TO CLIENT;
   CALL InnerProc();
   MESSAGE 'SQLSTATE set to ',
      SQLSTATE,' in OuterProc.' TO CLIENT
END;
CREATE PROCEDURE InnerProc()
   BEGIN
      DECLARE column_not_found
         EXCEPTION FOR SQLSTATE '52003';
      MESSAGE 'Hello from InnerProc.' TO CLIENT;
```

```
      SIGNAL column_not_found;
   MESSAGE 'SQLSTATE set to ',
      SQLSTATE, ' in InnerProc.' TO CLIENT;
END;

CALL OuterProc();
```

The Interactive SQL Messages tab displays the following:

```
Hello from OuterProc.
Hello from InnerProc.
```

The DECLARE statement in InnerProc declares a symbolic name for one of the predefined SQLSTATE values associated with error conditions already known to the server.

The MESSAGE statement sends a message to the Interactive SQL Messages tab.

The SIGNAL statement generates an error condition from within the InnerProc procedure.

None of the statements following the SIGNAL statement in InnerProc execute: InnerProc immediately passes control back to the calling environment, which in this case is the procedure OuterProc. None of the statements following the CALL statement in OuterProc execute. The error condition returns to the calling environment to be handled there. For example, Interactive SQL handles the error by displaying a message window describing the error.

The TRACEBACK function provides a list of the statements that were executing when the error occurred. You can use the TRACEBACK function from Interactive SQL by entering the following statement:

```
SELECT TRACEBACK();
```

Error handling with ON EXCEPTION RESUME

If the ON EXCEPTION RESUME clause appears in the CREATE PROCEDURE statement, the procedure checks the following statement when an error occurs. If the statement handles the error, then the procedure continues executing, resuming at the statement after the one causing the error. It does not return control to the calling environment when an error occurred.

The behavior for procedures that use ON EXCEPTION RESUME can be modified by the on_tsql_error option setting. See “on_tsql_error option” [*SQL Anywhere Server - Database Administration*].

Error-handling statements include the following:

- IF
- SELECT @variable =
- CASE
- LOOP
- LEAVE
- CONTINUE
- CALL
- EXECUTE
- SIGNAL
- RESIGNAL
- DECLARE
- SET VARIABLE

The following demonstration procedures show what happens when an application calls the procedure OuterProc; and OuterProc in turn calls the procedure InnerProc, which then encounters an error. These demonstration procedures are based on those used earlier in this section:

```
DROP PROCEDURE OuterProc;
DROP PROCEDURE InnerProc;

CREATE PROCEDURE OuterProc()
ON EXCEPTION RESUME
BEGIN
   DECLARE res CHAR(5);
   MESSAGE 'Hello from OuterProc.' TO CLIENT;
   CALL InnerProc();
   SET res=SQLSTATE;
   IF res='52003' THEN
      MESSAGE 'SQLSTATE set to ',
         res, ' in OuterProc.' TO CLIENT;
   END IF
END;

CREATE PROCEDURE InnerProc()
ON EXCEPTION RESUME
BEGIN
   DECLARE column_not_found
      EXCEPTION FOR SQLSTATE '52003';
   MESSAGE 'Hello from InnerProc.' TO CLIENT;
   SIGNAL column_not_found;
   MESSAGE 'SQLSTATE set to ',
   SQLSTATE, ' in InnerProc.' TO CLIENT;
END;

CALL OuterProc();
```

The Interactive SQL Messages tab then displays the following:

```
Hello from OuterProc.
Hello from InnerProc.
SQLSTATE set to 52003 in OuterProc.
```

The execution path taken is as follows:

1. OuterProc executes and calls InnerProc.

2. In InnerProc, the SIGNAL statement signals an error.
3. The MESSAGE statement is not an error-handling statement, so control is passed back to OuterProc and the message is not displayed.
4. In OuterProc, the statement following the error assigns the SQLSTATE value to the variable named **res**. This is an error-handling statement, and so execution continues and the OuterProc message appears.

Default handling of warnings in procedures and triggers

Errors and warnings are handled differently. While the default action for errors is to set a value for the SQLSTATE and SQLCODE variables, and return control to the calling environment in the event of an error, the default action for warnings is to set the SQLSTATE and SQLCODE values and continue execution of the procedure.

The following demonstration procedures illustrate default handling of warnings. These demonstration procedures are based on those used in "Default error handling in procedures and triggers" on page 829.

In this case, the SIGNAL statement generates a `row not found` condition, which is a warning rather than an error.

```
DROP PROCEDURE OuterProc;
DROP PROCEDURE InnerProc;

CREATE PROCEDURE OuterProc()
BEGIN
   MESSAGE 'Hello from OuterProc.' TO CLIENT;
   CALL InnerProc();
   MESSAGE 'SQLSTATE set to ',
      SQLSTATE,' in OuterProc.' TO CLIENT;
END;
CREATE PROCEDURE InnerProc()
BEGIN
   DECLARE row_not_found
      EXCEPTION FOR SQLSTATE '02000';
   MESSAGE 'Hello from InnerProc.' TO CLIENT;
   SIGNAL row_not_found;
   MESSAGE 'SQLSTATE set to ',
   SQLSTATE, ' in InnerProc.' TO CLIENT;
END;

CALL OuterProc();
```

The Interactive SQL Messages tab then displays the following:

```
Hello from OuterProc.
Hello from InnerProc.
SQLSTATE set to 02000 in InnerProc.
SQLSTATE set to 00000 in OuterProc.
```

The procedures both continued executing after the warning was generated, with SQLSTATE set by the warning (02000).

Execution of the second MESSAGE statement in InnerProc resets the warning. Successful execution of any SQL statement resets SQLSTATE to 00000 and SQLCODE to 0. If a procedure needs to save the

error status, it must do an assignment of the value immediately after execution of the statement which caused the error or warning.

Using exception handlers in procedures and triggers

It is often desirable to intercept certain types of errors and handle them within a procedure or trigger, rather than pass the error back to the calling environment. This is done through the use of an **exception handler**.

You define an exception handler with the EXCEPTION part of a compound statement. See “Using compound statements” on page 816.

Whenever an error occurs in the compound statement, the exception handler executes. Unlike errors, warnings do not cause exception handling code to be executed. Exception handling code also executes if an error appears in a nested compound statement or in a procedure or trigger invoked anywhere within the compound statement.

An exception handler for the interrupt error SQL_INTERRUPT, SQLSTATE 57014 should only contain non-interruptible statements such as ROLLBACK and ROLLBACK TO SAVEPOINT. If the exception handler contains interruptible statements that are invoked when the connection is interrupted, the database server stops the exception handler at the first interruptible statement and returns the interrupt error.

An exception handler can use the SQLSTATE or SQLCODE special values to determine why a statement failed. Alternatively, the ERRORMSG function can be used without an argument to return the error condition associated with a SQLSTATE. Only the first statement in each WHEN clause can specify this information and the statement cannot be a compound statement.

The demonstration procedures used to illustrate exception handling are based on those used in “Default error handling in procedures and triggers” on page 829.

In this example, additional code handles the **column not found** error in the InnerProc procedure.

```
DROP PROCEDURE OuterProc;
DROP PROCEDURE InnerProc;

CREATE PROCEDURE OuterProc()
BEGIN
   MESSAGE 'Hello from OuterProc.' TO CLIENT;
   CALL InnerProc();
   MESSAGE 'SQLSTATE set to ',
      SQLSTATE,' in OuterProc.' TO CLIENT
END;
CREATE PROCEDURE InnerProc()
BEGIN
   DECLARE column_not_found
    EXCEPTION FOR SQLSTATE '52003';
   MESSAGE 'Hello from InnerProc.' TO CLIENT;
  SIGNAL column_not_found;
  MESSAGE 'Line following SIGNAL.' TO CLIENT;
   EXCEPTION
      WHEN column_not_found THEN
      MESSAGE 'Column not found handling.' TO CLIENT;
        WHEN OTHERS THEN
```

```
      RESIGNAL ;
END;

CALL OuterProc();
```

The Interactive SQL **Messages** tab then displays the following:

```
Hello from OuterProc.
Hello from InnerProc.
Column not found handling.
SQLSTATE set to 00000 in OuterProc.
```

The EXCEPTION clause declares the exception handler. The lines following EXCEPTION do not execute unless an error occurs. Each WHEN clause specifies an exception name (declared with a DECLARE statement) and the statement or statements to be executed in the event of that exception. The WHEN OTHERS THEN clause specifies the statement(s) to be executed when the exception that occurred does not appear in the preceding WHEN clauses.

In the above example, the statement RESIGNAL passes the exception on to a higher-level exception handler. RESIGNAL is the default action if WHEN OTHERS THEN is not specified in an exception handler.

Additional notes

- The EXCEPTION handler executes, rather than the lines following the SIGNAL statement in InnerProc.
- As the error encountered was a **column not found** error, the MESSAGE statement included to handle the error executes, and SQLSTATE resets to zero (indicating no errors).
- After the exception handling code executes, control passes back to OuterProc, which proceeds as if no error was encountered.
- You should not use ON EXCEPTION RESUME together with explicit exception handling. The exception handling code is not executed if ON EXCEPTION RESUME is included.
- If the error handling code for the **column not found** exception is a RESIGNAL statement, control returns to the OuterProc procedure with SQLSTATE still set at the value 52003. This is just as if there were no error handling code in InnerProc. Since there is no error handling code in OuterProc, the procedure fails.

Exception handling and atomic compound statements

When an exception is handled inside a compound statement, the compound statement completes without an active exception and the changes before the exception are not reversed. This is true even for atomic compound statements. If an error occurs within an atomic compound statement and is explicitly handled, some, but not all, the statements in the atomic compound statement are executed.

See also

- "SQLCODE special value" [*SQL Anywhere Server - SQL Reference*]
- "SQLSTATE special value" [*SQL Anywhere Server - SQL Reference*]
- "ERRORMSG function [Miscellaneous]" [*SQL Anywhere Server - SQL Reference*]
- "RESIGNAL statement" [*SQL Anywhere Server - SQL Reference*]

Nested compound statements and exception handlers

The code following a statement that causes an error executes only if an ON EXCEPTION RESUME clause appears in a procedure definition.

You can use nested compound statements to give you more control over which statements execute following an error and which do not.

The following example illustrates how nested compound statements can be used to control flow. The procedure is based on that used as an example in "Default error handling in procedures and triggers" on page 829.

```
DROP PROCEDURE OuterProc;
DROP PROCEDURE InnerProc;

CREATE PROCEDURE InnerProc()
BEGIN
   BEGIN
      DECLARE column_not_found
      EXCEPTION FOR SQLSTATE VALUE '52003';
      MESSAGE 'Hello from InnerProc' TO CLIENT;
      SIGNAL column_not_found;
      MESSAGE 'Line following SIGNAL' TO CLIENT
      EXCEPTION
         WHEN column_not_found THEN
            MESSAGE 'Column not found handling' TO
            CLIENT;
         WHEN OTHERS THEN
            RESIGNAL;
   END;
   MESSAGE 'Outer compound statement' TO CLIENT;
END;

CALL InnerProc();
```

The Interactive SQL **Messages** tab then displays the following:

```
Hello from InnerProc
Column not found handling
Outer compound statement
```

When the SIGNAL statement that causes the error is encountered, control passes to the exception handler for the compound statement, and the `Column not found handling` message prints. Control then passes back to the outer compound statement and the `Outer compound statement` message prints.

If an error other than `column not found` is encountered in the inner compound statement, the exception handler executes the RESIGNAL statement. The RESIGNAL statement passes control directly back to the calling environment, and the remainder of the outer compound statement is not executed.

Using the EXECUTE IMMEDIATE statement in procedures

The EXECUTE IMMEDIATE statement allows statements to be constructed inside procedures using a combination of literal strings (in quotes) and variables. For example, the following procedure includes an EXECUTE IMMEDIATE statement that creates a table.

```
CREATE PROCEDURE CreateTableProcedure(
      IN tablename char(128) )
BEGIN
   EXECUTE IMMEDIATE 'CREATE TABLE '
   || tablename
   || '(column1 INT PRIMARY KEY)'
END;
```

The EXECUTE IMMEDIATE statement can be used with queries that return result sets. For example:

```
CREATE PROCEDURE DynamicResult(
   IN Columns LONG VARCHAR,
   IN TableName CHAR(128),
   IN Restriction LONG VARCHAR DEFAULT NULL )
BEGIN
    DECLARE Command LONG VARCHAR;
    SET Command = 'SELECT ' || Columns || ' FROM ' || TableName;
    IF ISNULL( Restriction,'') <> '' THEN
         SET Command = Command || ' WHERE ' || Restriction;
    END IF;
    EXECUTE IMMEDIATE WITH RESULT SET ON Command;
END;
```

The following statement calls this procedure:

```
CALL DynamicResult(
   'table_id,table_name',
   'SYSTAB',
   'table_id <= 10');
```

table_id	table_name
1	ISYSTAB
2	ISYSTABCOL
3	ISYSIDX
...	...

In ATOMIC compound statements, you cannot use an EXECUTE IMMEDIATE statement that causes a COMMIT, as COMMITs are not allowed in that context.

See "EXECUTE IMMEDIATE statement [SP]" [*SQL Anywhere Server - SQL Reference*].

Transactions and savepoints in procedures and triggers

SQL statements in a procedure or trigger are part of the current transaction. You can call several procedures within one transaction or have several transactions in one procedure.

COMMIT and ROLLBACK are not allowed within any atomic statement. Note that triggers are fired due to an INSERT, UPDATE, or DELETE which are atomic statements. COMMIT and ROLLBACK are not allowed in a trigger or in any procedures called by a trigger.

Savepoints can be used within a procedure or trigger, but a ROLLBACK TO SAVEPOINT statement can never refer to a savepoint before the atomic operation started. Also, all savepoints within an atomic operation are released when the atomic operation completes.

See also

- "Using transactions and isolation levels" on page 89
- "Atomic compound statements" on page 817
- "Savepoints within transactions" on page 92

Tips for writing procedures

This section provides some pointers for developing procedures.

Check if you need to change the command delimiter

You do not need to change the command delimiter in Interactive SQL or Sybase Central when you write procedures. However, if you create and test procedures and triggers from some other browsing tool, you may need to change the command delimiter from the semicolon to another character.

Each statement within the procedure ends with a semicolon. For some browsing applications to parse the CREATE PROCEDURE statement itself, you need the command delimiter to be something other than a semicolon.

If you are using an application that requires changing the command delimiter, a good choice is to use two semicolons as the command delimiter (;;) or a question mark (?) if the system does not permit a multi-character delimiter.

Remember to delimit statements within your procedure

You should end each statement within the procedure with a semicolon. Although you can leave off semicolons for the last statement in a statement list, it is good practice to use semicolons after each statement.

The CREATE PROCEDURE statement itself contains both the RESULT specification and the compound statement that forms its body. No semicolon is needed after the BEGIN or END keywords, or after the RESULT clause.

Use fully-qualified names for tables in procedures

If a procedure has references to tables in it, you should always preface the table name with the name of the owner (creator) of the table.

When a procedure refers to a table, it uses the group memberships of the procedure creator to locate tables with no explicit owner name specified. For example, if a procedure created by user_1 references Table_B and does not specify the owner of Table_B, then either Table_B must have been created by user_1 or user_1 must be a member of a group (directly or indirectly) that is the owner of Table_B. If neither condition is met, a `table not found` message results when the procedure is called.

You can minimize the inconvenience of long fully qualified names by using a correlation name to provide a convenient name to use for the table within a statement. Correlation names are described in "FROM clause" [*SQL Anywhere Server - SQL Reference*].

Specifying dates and times in procedures

When dates and times are sent to the database from procedures, they are sent as strings. The date part of the string is interpreted according to the current setting of the date_order database option. As different connections may set this option to different values, some strings may be converted incorrectly to dates, or the database may not be able to convert the string to a date.

You should use the unambiguous date format *yyyy-mm-dd* or *yyyy/mm/dd* when using date strings within procedures. The server interprets these strings unambiguously as dates, regardless of the date_order database option setting. See "Date and time data types" [*SQL Anywhere Server - SQL Reference*].

Verifying that procedure input arguments are passed correctly

One way to verify input arguments is to display the value of the parameter on the Interactive SQL **Messages** tab using the MESSAGE statement. For example, the following procedure simply displays the value of the input parameter *var*:

```
CREATE PROCEDURE message_test( IN var char(40) )
BEGIN
   MESSAGE var TO CLIENT;
END;
```

You can also use the debugger to verify that procedure input arguments were passed correctly. See "Lesson 2: Debug a stored procedure" on page 843.

Statements allowed in procedures, triggers, events, and batches

Most SQL statements are acceptable in batches, with the exception of the following:

- ALTER DATABASE (syntax 3 and 4)
- CONNECT
- CREATE DATABASE
- CREATE DECRYPTED FILE
- CREATE ENCRYPTED FILE
- DISCONNECT
- DROP CONNECTION
- DROP DATABASE
- FORWARD TO
- Interactive SQL commands such as INPUT or OUTPUT
- PREPARE TO COMMIT
- STOP SERVER

You can use COMMIT, ROLLBACK, and SAVEPOINT statements within procedures, triggers, events, and batches with certain restrictions. See "Transactions and savepoints in procedures and triggers" on page 836.

For more information, see the Usage for each SQL statement in "SQL statements" [*SQL Anywhere Server - SQL Reference*].

Using SELECT statements in batches

You can include one or more SELECT statements in a batch. For example:

```
IF EXISTS( SELECT *
           FROM SYSTAB
           WHERE table_name='Employees' )
THEN
   SELECT   Surname AS LastName,
            GivenName AS FirstName
   FROM Employees;
   SELECT Surname, GivenName
   FROM Customers;
   SELECT Surname, GivenName
   FROM Contacts;
END IF;
```

The alias for the result set is necessary only in the first SELECT statement, as the server uses the first SELECT statement in the batch to describe the result set.

A RESUME statement is necessary following each query to retrieve the next result set.

Hiding the contents of procedures, functions, triggers and views

You may want to distribute an application and a database without disclosing the logic contained within procedures, functions, triggers and views. As an added security measure, you can obscure the contents of these objects using the SET HIDDEN clause of the ALTER PROCEDURE, ALTER FUNCTION, ALTER TRIGGER, and ALTER VIEW statements.

The SET HIDDEN clause obfuscates the contents of the associated objects and makes them unreadable, while still allowing the objects to be used. You can also unload and reload the objects into another database.

The modification is irreversible, and deletes the original text of the object. Preserving the original source for the object outside the database is required.

Debugging using the debugger will not show the procedure definition, nor will procedure profiling display the source.

Running one of the above statements on an object that is already hidden has no effect.

To hide the text for all objects of a particular type, you can use a loop similar to the following:

```
BEGIN
    FOR hide_lp as hide_cr cursor FOR
        SELECT proc_name, user_name
        FROM SYS.SYSPROCEDURE p, SYS.SYSUSER u
        WHERE p.creator = u.user_id
        AND p.creator NOT IN (0,1,3)
    DO
        MESSAGE 'altering ' || proc_name;
        EXECUTE IMMEDIATE 'ALTER PROCEDURE "' ||
            user_name || '"."' || proc_name
            || '" SET HIDDEN'
    END FOR
END;
```

See also

- "ALTER FUNCTION statement" [*SQL Anywhere Server - SQL Reference*]
- "ALTER PROCEDURE statement" [*SQL Anywhere Server - SQL Reference*]
- "ALTER TRIGGER statement" [*SQL Anywhere Server - SQL Reference*]
- "ALTER VIEW statement" [*SQL Anywhere Server - SQL Reference*]

Debugging procedures, functions, triggers, and events

Introduction to the SQL Anywhere debugger

You can use the SQL Anywhere debugger to debug SQL stored procedures, triggers, event handlers, and user-defined functions you create.

You can also use the debugger to:

- **Debug event handlers** Event handlers are an extension of SQL stored procedures. The material in this section about debugging stored procedures applies equally to debugging event handlers.
- **Browse stored procedures and classes** You can browse the source code of SQL procedures.
- **Trace execution** Step line by line through the code of a stored procedure. You can also look up and down the stack of functions that have been called.
- **Set breakpoints** Run the code until you hit a breakpoint, and stop at that point in the code.
- **Set break conditions** Breakpoints include lines of code, but you can also specify conditions when the code is to break. For example, you can stop at a line the tenth time it is executed, or only if a variable has a particular value.
- **Inspect and modify local variables** When execution is stopped at a breakpoint, you can inspect the values of local variables and alter their value.

- **Inspect and break on expressions** When execution is stopped at a breakpoint, you can inspect the value of a wide variety of expressions.
- **Inspect and modify row variables** Row variables are the OLD and NEW values of row-level triggers. You can inspect and modify these values.
- **Execute queries** You can execute queries when execution is stopped at a breakpoint in a SQL procedure. This permits you to look at intermediate results held in temporary tables, check values in base tables, and to view the query execution plan.

Requirements for using the debugger

To use the debugger, you must either have DBA authority or be granted permissions in the SA_DEBUG group. This group is added to all databases when they are created. Only one user can debug a database at a time.

When using the debugger over HTTP/SOAP connections, you should change the port timeout options on the server. For example, `-xs http{to=600;kto=0;port=8081)` sets the timeout to 6 minutes and turns off keep-alive timeout for port 8081. Note that timeout (to) is the period of time between received packets. Keep-alive timeout (kto) is the total time that the connection is allowed to run. When you set kto to 0, it is equivalent to setting it to never time out. See "-xs dbeng12/dbsrv12 server option" [*SQL Anywhere Server - Database Administration*].

If using a SQL Anywhere HTTP/SOAP client procedure to call into the SQL Anywhere HTTP/SOAP service you are debugging, you should set the client's remote_idle_timeout database option to a large value such as 150 (the default is 15 seconds) to avoid timing out during the debugging session. See "remote_idle_timeout option" [*SQL Anywhere Server - Database Administration*].

Tutorial: Getting started with the debugger

This tutorial describes how to connect to a database, how start the debugger, and how to debug a simple stored procedure.

Lesson 1: Connect to a database and start the debugger

To start the debugger

1. Create a directory to hold the copy of the sample database you will use in this tutorial, for example *c:\demodb*.
2. Copy and rename the sample database from *samples-dir\demo.db* to *c:\demodb*.

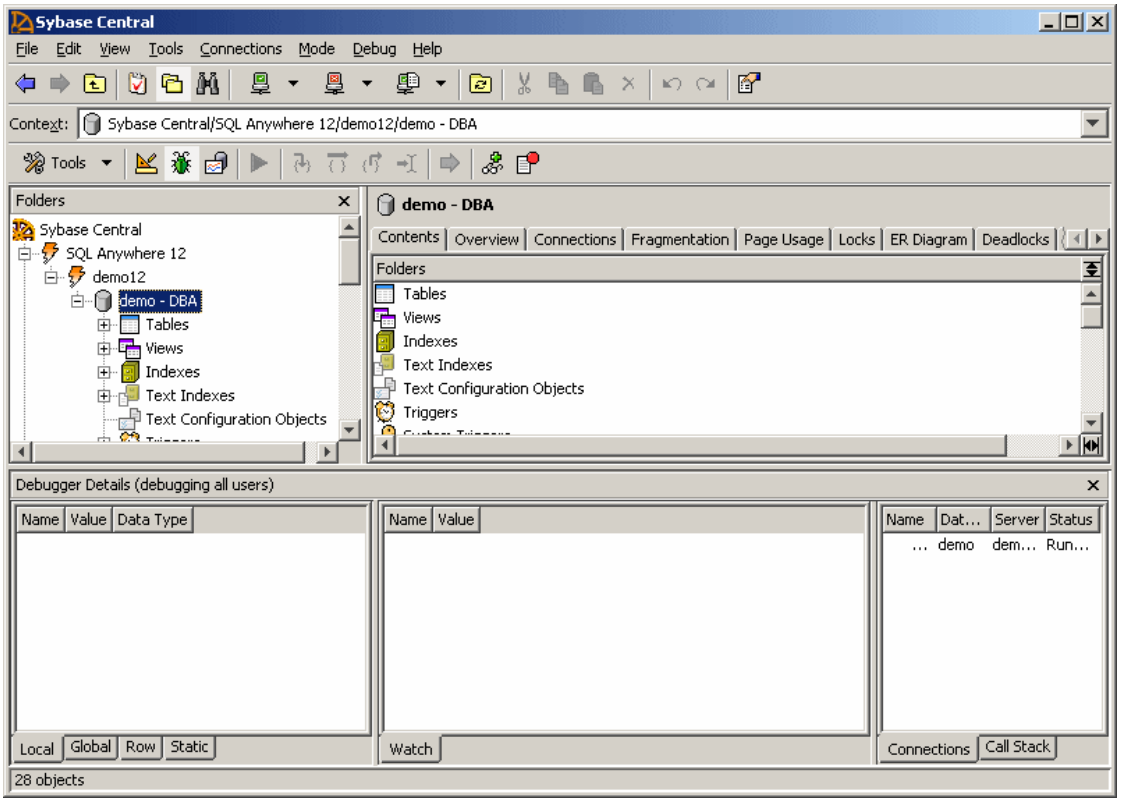
   For information about *samples-dir*, see "Samples directory" [*SQL Anywhere Server - Database Administration*].

3. Click **Start** » **Programs** » **SQL Anywhere 12** » **Administration Tools** » **Sybase Central**.
4. Choose **Connections** » **Connect With SQL Anywhere 12**.
5. In the **User ID** field, type **DBA** and in the **Password** field, type **sql**.
6. From the **Action** dropdown list, choose **Start A Database On This Computer**.
7. Click **Database File**, and then in the box below, *c:\demodb*.
8. Click **Connect**.
9. Choose **Mode** » **Debug**.
10. In the **Which User Would You Like To Debug** field, type ***** and click **OK**.

    If you want to debug a different user, you must exit Debug mode, and then re-enter Debug mode.

    The **Debugger Details** pane appears at the bottom of Sybase Central and the Sybase Central toolbar displays a set of debugger tools.

    When you provide a user name, information for connections with that user name is captured and appears on the **Connections** tab.



11. (optional) Restore the sample database (*demo.db*) to its original state by following the steps found here: "Recreate the sample database (demo.db)" [*SQL Anywhere 12 - Introduction*].

Lesson 2: Debug a stored procedure

This lesson illustrates how to use the debugger to identify errors in stored procedures. To set the stage, you introduce a deliberate error into the debugger_tutorial, which is part of the SQL Anywhere sample database.

The debugger_tutorial procedure should return a result set that contains the name of the company that has placed the highest value of orders, and the value of their orders. It computes these values by looping over the result set of a query that lists companies and orders. (This result could be achieved without adding the logic into the procedure by using a SELECT FIRST query. The procedure is used to create a convenient example.) The procedure has an intentional bug in it. In this tutorial you diagnose and fix the bug.

Run the debugger_tutorial procedure

The debugger_tutorial procedure should return a result set consisting of the top company and the value of products they have ordered. As a result of a bug, it does not return this result set. In this lesson, you run the stored procedure.

To run the debugger_tutorial stored procedure

1. In the left pane of Sybase Central, double-click **Procedures & Functions**.
2. Right-click **Debugger_Tutorial (GROUPO)** and choose **Execute From Interactive SQL**.

   Interactive SQL opens and the following result set appears:

top_company	top_value
(NULL)	(NULL)

   This is an incorrect result. The remainder of the tutorial diagnoses the error that produced this result.
3. Close Interactive SQL.

Diagnose the bug

To diagnose the bug in the procedure, set breakpoints in the procedure and step through the code, watching the value of variables as the procedure is executed.

Here, you set a breakpoint at the first executable statement in the procedure.

To diagnose the bug

1. Choose **Mode** » **Debug**.
2. In the right pane, double-click **Debugger_Tutorial (GROUPO)**.
3. In the right pane, locate the following statement:

   ```
   OPEN cursor_this_customer;
   ```

4. To add a breakpoint, click the vertical gray area to the left of the statement. The breakpoint appears as a red circle.
5. In the left pane, right-click **Debugger_Tutorial (GROUPO)** and choose **Execute From Interactive SQL**.

   In the **Connections** tab of Sybase Central, a yellow arrow indicating the breakpoint appears.
6. In the **Debugger Details** window, click the **Local** tab to display a list of local variables in the procedure together with their current value and data type. The top_company, top_value, this_value, and this_company variables are all uninitialized and are therefore NULL.
7. Press F11 to scroll through the procedure. The value of the variables changes when you reach the following line:

   ```
   IF this_value > top_value THEN
   ```

8. Press F11 once more to determine which branch the execution takes. The yellow arrow moves back to the following text:

   ```
   customer_loop: loop
   ```

   The `IF` test did not return true. The test failed because a comparison of any value to NULL returns NULL. A value of NULL fails the test and the code inside the `IF...END IF` statement is not executed.

   At this point, you may realize that the problem is the fact that top_value is not initialized.

Confirm the diagnosis and fix the bug

You can test the hypothesis that the problem is the lack of initialization for top_value right in the debugger, without changing the procedure code.

To test the hypothesis

1. In the **Debugger Details** window, click the **Local** tab.
2. Click the **top_value** variable and type **3000** in the **Value** field.
3. Press F11 repeatedly until the **Value** field of the **This_Value** variable is greater than 3000.
4. Click the breakpoint so that it turns gray.

5. Press F5 to execute the procedure.

   The Interactive SQL window appears again. It shows the correct results.

top_company	top_value
Chadwicks	8076

The hypothesis is confirmed. The problem is that the top_value is not initialized.

To fix the bug

1. Choose **Mode** » **Design**.
2. In the right pane, locate the following statement:

   ```
   OPEN cursor_this_customer;
   ```

3. Type a new line that initializes the top_value variable:

   ```
   SET top_value = 0;
   ```

4. Choose **File** » **Save**.
5. Execute the procedure again, and confirm that Interactive SQL displays the correct results.

You have now completed the lesson. Close any open Interactive SQL windows.

Working with breakpoints

Breakpoints control when the debugger interrupts execution of your source code.

When you are running in Debug mode, and a connection hits a breakpoint, the behavior changes depending on the connection that is selected:

- If you do not have a connection selected, the connection is automatically selected and the source for the procedure is shown to debug it.
- If you already have a connection selected and it is the same connection that hit the breakpoint, the source for the procedure is shown to debug it.
- If you already have a connection selected, but it is not the connection that hit the breakpoint, a window appears that prompts you to change to the connection that encountered the breakpoint.

Set breakpoints

A breakpoint instructs the debugger to interrupt execution at a specified line. By default, a breakpoint applies to all connections.

To set a breakpoint

1. Use the SQL Anywhere 12 plug-in to connect to the database as a user with DBA authority.
2. In the left pane, double-click **Procedures & Functions**.
3. Select a procedure.
4. Choose **Mode** » **Debug**.
5. In the **Which User Would You Like To Debug** field, type ***** to debug all users, or type the name of the database user you want to debug.
6. In the right pane, click the line where you want to insert the breakpoint.

   A cursor appears in the line where you clicked.
7. Press F9.

   A red circle appears to the left of the line of code.

To set a breakpoint (Debug menu)

1. Choose **Debug** » **Breakpoints**.
2. Click **New**.
3. In the **Procedure** list, select a procedure.
4. If required, complete the **Condition** and **Count** fields.

   The Condition is a SQL expression that must evaluate to true for the breakpoint to interrupt execution. For example, you can set a breakpoint to apply to a connection made by a specified user, by entering the following condition:

   ```
   CURRENT USER = 'user-name'
   ```

   The Count is the number of times the breakpoint is hit before it stops execution. A value of 0 means that the breakpoint always stops execution.
5. Click **OK**. The breakpoint is set on the first executable statement in the procedure.

Disable and enable breakpoints

You can change the status of a breakpoint from the Sybase Central right pane or from the **Breakpoints** window.

To change the status of a breakpoint

1. Use the SQL Anywhere 12 plug-in to connect to the database as a user with DBA authority.
2. In the left pane, double-click **Procedures & Functions**.
3. Select a procedure.
4. Choose **Mode** » **Debug**.
5. In the right pane, click the breakpoint indicator to the left of the line you want to edit. The breakpoint changes from active to inactive.

To change the status of a breakpoint (Breakpoints window)

1. Use the SQL Anywhere 12 plug-in to connect to the database as a user with DBA authority.
2. In the left pane, double-click **Procedures & Functions**.
3. Select a procedure.
4. Choose **Mode** » **Debug**.
5. Choose **Debug** » **Breakpoints**.
6. Select the breakpoint and click **Edit**, **Disable** or **Remove**.
7. Click **Close**.

Edit breakpoint conditions

You can add conditions to breakpoints to instruct the debugger to interrupt execution at that breakpoint only when a certain condition or count is satisfied. For procedures and triggers, it must be a SQL search condition.

For example, to make a breakpoint apply to a specific connection only, set a condition on the breakpoint.

To set a condition or count on a breakpoint

1. Use the SQL Anywhere 12 plug-in to connect to the database as a user with DBA authority.
2. In the left pane, double-click **Procedures & Functions**.
3. Select a procedure.
4. Choose **Mode** » **Debug**.
5. Choose **Debug** » **Breakpoints**.
6. Select the breakpoint you want to edit and then click **Edit**.

7. In the **Condition** list, click a condition. For example, to set the breakpoint so that it applies only to connections from a specific user ID, enter the following condition:

   ```
   CURRENT USER='user-name'
   ```

   In this condition, *user-name* is the user ID for which the breakpoint is to be active.

8. Click **OK** and then click **Close**.

Working with variables

The debugger lets you view and edit the behavior of your variables while stepping through your code. The debugger provides a **Debugger Details** pane to display the different kinds of variables used in stored procedures. The **Debugger Details** pane appears at the bottom of Sybase Central when Sybase Central is running in Debug mode.

View variable values

To view variable values

1. Use the SQL Anywhere 12 plug-in to connect to the database as a user with DBA authority.
2. In the left pane, double-click **Procedures & Functions**.
3. Select a procedure.
4. Choose **Mode** » **Debug**.
5. In the **Which User Would You Like To Debug** field, type ***** to debug all users, or type the name of the database user you want to debug.
6. In the right pane, click the line where you want to insert the breakpoint.

   A cursor appears in the line where you clicked.

7. Press F9.

   A red circle appears to the left of the line of code.

8. In the **Debugger Details** pane, click the **Local** tab.
9. In the left pane, right-click the procedure and choose **Execute From Interactive SQL**. The variables, along with their values, appear on the **Local** tab.

Viewing global variables

Global variables are defined by SQL Anywhere and hold information about the current connection, database, and other settings. They appear in the **Debugger Details** pane on the **Global** tab.

For a list of global variables, see "Global variables" [*SQL Anywhere Server - SQL Reference*].

Row variables are used in triggers to hold the values of rows affected by the triggering statement. They appear in the **Debugger Details** pane on the **Row** tab.

For more information about triggers, see "Introduction to triggers" on page 803.

Static variables are used in Java classes. They are appear on the **Statics** tab.

Display the call stack

It is useful to examine the sequence of calls that has been made when you are debugging nested procedures. You can view a listing of the procedures on the **Call Stack** tab.

To display the call stack

1. Use the SQL Anywhere 12 plug-in to connect to the database as a user with DBA authority.
2. In the left pane, double-click **Procedures & Functions**.
3. Select a procedure.
4. Choose **Mode** » **Debug**.
5. In the **Which User Would You Like To Debug** field, type ***** to debug all users, or type the name of the database user you want to debug.
6. In the right pane, click the line where you want to insert the breakpoint.

   A cursor appears in the line where you clicked.
7. Press F9.

   A red circle appears to the left of the line of code.
8. In the **Debugger Details** pane, click the **Local** tab.
9. In the left pane, right-click the procedure and choose **Execute From Interactive SQL**.
10. In the **Debugger Details** pane, click the **Call Stack** tab.

    The names of the procedures appear on the Calls Stack tab. The current procedure is shown at the top of the list. The procedure that called it is immediately below.

Working with connections

The **Connections** tab displays the connections to the database. At any time, multiple connections may be running. Some may be stopped at a breakpoint, and others may not.

To switch connections, double-click a connection on the **Connections** tab.

A useful technique is to set a breakpoint so that it interrupts execution for a single user ID. You can do this by setting a breakpoint condition of the following form:

```
CURRENT USER = 'user-name'
```

The SQL special value CURRENT USER holds the user ID of the connection.

For more information, see "Edit breakpoint conditions" on page 847, and "CURRENT USER special value" [*SQL Anywhere Server - SQL Reference*].

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

