Copyright and trademarks

Copyright © 2008 iAnywhere Solutions, Inc. Portions copyright © 2008 Sybase, Inc. All rights reserved.

This documentation is provided AS IS, without warranty or liability of any kind (unless provided by a separate written agreement between you and iAnywhere).

You may use, print, reproduce, and distribute this documentation (in whole or in part) subject to the following conditions: 1) you must retain this and all other proprietary notices, on all copies of the documentation or portions thereof, 2) you may not modify the documentation, 3) you may not do anything to indicate that you or anyone other than iAnywhere is the author or source of the documentation.

iAnywhere®, Sybase®, and the marks listed at http://www.sybase.com/detail?id=1011207 are trademarks of Sybase, Inc. or its subsidiaries. ® indicates registration in the United States of America.

All other company and product names mentioned may be trademarks of the respective companies with which they are associated.
# Contents

About this book ........................................................................................................... ix

About the SQL Anywhere documentation ................................................................ x

I. Designing and Creating Databases ................................................................. 1

1. Designing and creating your database ................................................................. 3
   Introduction to database design ................................................................................. 4
   Database design concepts ......................................................................................... 5
   The design process ................................................................................................... 11
   Designing the database table properties ............................................................... 24
   Tutorial: Creating a SQL Anywhere database ....................................................... 28

2. Working with database objects ............................................................................... 35
   Introduction to working with database objects ...................................................... 36
   Working with tables ................................................................................................ 38
   Working with temporary tables .............................................................................. 52
   Working with views .................................................................................................. 54
   Working with indexes .............................................................................................. 86

3. Ensuring data integrity ......................................................................................... 93
   Introduction to data integrity .................................................................................... 94
   Using column defaults .......................................................................................... 97
   Using table and column constraints .................................................................. 103
   Using domains ...................................................................................................... 107
   Enforcing entity and referential integrity ........................................................... 110
   Integrity rules in the system tables ..................................................................... 117

4. Using transactions and isolation levels ............................................................ 119
   Introduction to transactions ................................................................................... 120
   Isolation levels and consistency ........................................................................... 124
   Transaction blocking and deadlock ....................................................................... 138
   How locking works ............................................................................................... 142
   Choosing isolation levels ...................................................................................... 156
   Isolation level tutorials ....................................................................................... 160
   Particular concurrency issues .............................................................................. 178
   Summary ................................................................................................................ 180
II. Monitoring and Improving Database Performance ........................................ 183

5. Monitoring and improving performance ..................................................... 185
   Introduction to monitoring and improving performance .............................. 186
   Application profiling .................................................................................. 188
   Advanced application profiling using diagnostic tracing ............................ 199
   Other diagnostic tools and techniques ........................................................ 217
   Monitoring database performance .............................................................. 223
   Performance improvement tips .................................................................... 239

6. Application profiling tutorials ................................................................. 263
   Setting up for the application profiling tutorials .......................................... 264
   Tutorial: Diagnosing deadlocks ................................................................... 265
   Tutorial: Diagnosing slow statements ......................................................... 270
   Tutorial: Diagnosing index fragmentation .................................................... 275
   Tutorial: Diagnosing table fragmentation .................................................... 279
   Tutorial: Baselining with procedure profiling ............................................. 284

III. Querying and Modifying Data ................................................................. 289

7. Querying data ............................................................................................ 291
   Query overview ............................................................................................ 292
   The select list: Specifying columns ............................................................. 295
   The FROM clause: Specifying tables .......................................................... 303
   The WHERE clause: Specifying rows .......................................................... 305
   The ORDER BY clause: Ordering results .................................................... 316
   Summarizing data ....................................................................................... 319
   Full text search ........................................................................................... 323

8. Summarizing, grouping, and sorting query results .................................... 349
   Summarizing query results using aggregate functions ............................... 350
   The GROUP BY clause: Organizing query results into groups ................... 355
   The HAVING clause: selecting groups of data ............................................. 360
   The ORDER BY clause: sorting query results ............................................ 362
   Performing set operations on query results with UNION, INTERSECT, and
   EXCEPT ....................................................................................................... 365
   Standards and compatibility ...................................................................... 371

9. Joins: Retrieving data from several tables ............................................... 373
14. Query optimization and execution ................................................................. 525
   Phases of query processing .............................................................................. 526
   Semantic query transformations ..................................................................... 528
   How the optimizer works .............................................................................. 541
   Query execution algorithms ......................................................................... 565
   Reading execution plans .............................................................................. 589
   Physical data organization and access ......................................................... 615

V. SQL Dialects and Compatibility ..................................................................... 625
15. Other SQL dialects ...................................................................................... 627
   Introduction to SQL Anywhere compliance ............................................... 628
   Testing SQL compliance using the SQL Flagger ......................................... 629
   Features not found in other SQL implementations ....................................... 631
   Transact-SQL Compatibility ........................................................................ 633
   Adaptive Server architectures ..................................................................... 636
   Configuring databases for Transact-SQL compatibility ............................... 641
   Writing compatible SQL statements ............................................................ 647
   Transact-SQL procedure language overview ............................................... 651
   Automatic translation of stored procedures ................................................ 653
   Returning result sets from Transact-SQL procedures .................................. 654
   Variables in Transact-SQL procedures ....................................................... 655
   Error handling in Transact-SQL procedures ................................................ 656

VI. XML in the Database ................................................................................... 659
16. Using XML in the database ......................................................................... 661
   Introduction to XML ..................................................................................... 662
   Storing XML documents in relational databases ....................................... 663
   Exporting relational data as XML ............................................................... 664
   Importing XML documents as relational data .......................................... 665
   Obtaining query results as XML ............................................................... 672
   Using SQL/XML to obtain query results as XML ..................................... 690

VII. Remote Data and Bulk Operations ............................................................. 699
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>17.</td>
<td>Importing and exporting data</td>
<td>701-704, 722-724, 733-735, 745-746, 750-752</td>
</tr>
<tr>
<td>19.</td>
<td>Server classes for remote data access</td>
<td>783-784, 785-799</td>
</tr>
<tr>
<td>VIII.</td>
<td>Stored Procedures and Triggers</td>
<td>803-804, 806-814</td>
</tr>
</tbody>
</table>
About this book

Subject
This book describes how to design and create databases; how to import, export, and modify data; how to retrieve data; and how to build stored procedures and triggers.

Audience
This book is for all users of SQL Anywhere.

Before you begin
This book assumes that you have an elementary familiarity with database management systems and SQL Anywhere in particular. If you do not have such a familiarity, you should consider reading SQL Anywhere 11 - Introduction before reading this book.
About the SQL Anywhere documentation

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

- **HTML Help**  The online Help contains the complete SQL Anywhere documentation, including the books and the context-sensitive help for SQL Anywhere tools. The online Help is updated with each release of the product.

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

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

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

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

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

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

About the books in the documentation set

SQL Anywhere documentation

The SQL Anywhere documentation consists of the following books:

- **SQL Anywhere 11 - Introduction**  This book introduces SQL Anywhere 11—a comprehensive package that provides data management and data exchange, enabling the rapid development of database-powered applications for server, desktop, mobile, and remote office environments.

- **SQL Anywhere 11 - Changes and Upgrading**  This book describes new features in SQL Anywhere 11 and in previous versions of the software.

- **SQL Anywhere Server - Database Administration**  This book describes how to run, manage, and configure SQL Anywhere databases. It describes database connections, the database server, database files, backup procedures, security, high availability, and replication with Replication Server, as well as administration utilities and options.

- **SQL Anywhere Server - Programming**  This book describes how to build and deploy database applications using the C, C++, Java, PHP, Perl, Python, and .NET programming languages such as Visual Basic and Visual C#. A variety of programming interfaces such as ADO.NET and ODBC are described.

- **SQL Anywhere Server - SQL Reference**  This book provides reference information for system procedures, and the catalog (system tables and views). It also provides an explanation of the SQL Anywhere implementation of the SQL language (search conditions, syntax, data types, and functions).
• **SQL Anywhere Server - SQL Usage**  This book describes how to design and create databases; how to import, export, and modify data; how to retrieve data; and how to build stored procedures and triggers.

• **MobiLink - Getting Started**  This book introduces MobiLink, a session-based relational-database synchronization system. MobiLink technology allows two-way replication and is well suited to mobile computing environments.

• **MobiLink - Client Administration**  This book describes how to set up, configure, and synchronize MobiLink clients. MobiLink clients can be SQL Anywhere or UltraLite databases. This book also describes the Dbmlsync API, which allows you to integrate synchronization seamlessly into your C++ or .NET client applications.

• **MobiLink - Server Administration**  This book describes how to set up and administer MobiLink applications.

• **MobiLink - Server-Initiated Synchronization**  This book describes MobiLink server-initiated synchronization, a feature of MobiLink that allows you to initiate synchronization or other remote actions from the consolidated database.

• **QAnywhere**  This book describes QAnywhere, which is a messaging platform for mobile and wireless clients, as well as traditional desktop and laptop clients.

• **SQL Remote**  This book describes the SQL Remote data replication system for mobile computing, which enables sharing of data between a SQL Anywhere consolidated database and many SQL Anywhere remote databases using an indirect link such as email or file transfer.

• **UltraLite - Database Management and Reference**  This book introduces the UltraLite database system for small devices.

• **UltraLite - C and C++ Programming**  This book describes UltraLite C and C++ programming interfaces. With UltraLite, you can develop and deploy database applications to handheld, mobile, or embedded devices.

• **UltraLite - M-Business Anywhere Programming**  This book describes UltraLite for M-Business Anywhere. With UltraLite for M-Business Anywhere you can develop and deploy web-based database applications to handheld, mobile, or embedded devices, running Palm OS, Windows Mobile, or Windows.

• **UltraLite - .NET Programming**  This book describes UltraLite.NET. With UltraLite.NET you can develop and deploy database applications to computers, or handheld, mobile, or embedded devices.

• **UltraLiteJ**  This book describes UltraLiteJ. With UltraLiteJ, you can develop and deploy database applications in environments that support Java. UltraLiteJ supports BlackBerry smartphones and Java SE environments. UltraLiteJ is based on the iAnywhere UltraLite database product.

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

## Documentation conventions

This section lists the typographic and graphical conventions used in this documentation.
Syntax conventions

The following conventions are used in the SQL syntax descriptions:

- **Keywords**  All SQL keywords appear in uppercase, like the words ALTER TABLE in the following example:

  ```sql
  ALTER TABLE [ owner.]table-name
  ```

- **Placeholders**  Items that must be replaced with appropriate identifiers or expressions are shown like the words `owner` and `table-name` in the following example:

  ```sql
  ALTER TABLE [ owner.]table-name
  ```

- **Repeating items**  Lists of repeating items are shown with an element of the list followed by an ellipsis (three dots), like `column-constraint` in the following example:

  ```sql
  ADD column-definition [ column-constraint, ... ]
  ```

  One or more list elements are allowed. In this example, if more than one is specified, they must be separated by commas.

- **Optional portions**  Optional portions of a statement are enclosed by square brackets.

  ```sql
  RELEASE SAVEPOINT [ savepoint-name ]
  ```

  These square brackets indicate that the `savepoint-name` is optional. The square brackets should not be typed.

- **Options**  When none or only one of a list of items can be chosen, vertical bars separate the items and the list is enclosed in square brackets.

  ```sql
  [ ASC | DESC ]
  ```

  For example, you can choose one of ASC, DESC, or neither. The square brackets should not be typed.

- **Alternatives**  When precisely one of the options must be chosen, the alternatives are enclosed in curly braces and a bar is used to separate the options.

  ```sql
  [ QUOTES { ON | OFF } ]
  ```

  If the QUOTES option is used, one of ON or OFF must be provided. The brackets and braces should not be typed.

Operating system conventions

SQL Anywhere runs on a variety of platforms, including Windows, Windows Mobile, Unix, Linux, and Mac OS X. To simplify references to operating systems, the documentation groups the supported operating systems as follows:

- **Windows**  The Microsoft Windows family of operating systems for desktop and laptop computers. The Windows family includes Windows Vista and Windows XP.

- **Windows Mobile**  Windows Mobile provides a Windows user interface and additional functionality, such as small versions of applications like Word and Excel. Windows Mobile is most commonly used on mobile devices.
Limitations or variations in SQL Anywhere are commonly based on the underlying operating system, and seldom on the particular variant used (Windows Mobile).

- **Unix** Unless specified, Unix refers to Linux, Mac OS X, and Unix platforms.

**File name conventions**

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

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

- **install-dir** During the installation process, you choose where to install SQL Anywhere. The documentation refers to this location using the convention \install-dir\.

  After installation is complete, the environment variable SQLANY11 specifies the location of the installation directory containing the SQL Anywhere components (\install-dir\).

  For example, the documentation may refer to a file as \install-dir\readme.txt. On Windows platforms, this reference is equivalent to %SQLANY11%\readme.txt. On Unix platforms, this reference is equivalent to $SQLANY11/readme.txt.

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

- **Uppercase and lowercase directory names** On Windows, directory names often use mixed case. References to directory names can use any case, since the Windows file system is not case sensitive. Unix file systems are case sensitive. The use of mixed-case is less common.

  The SQL Anywhere installation program follows operating system conventions for its directory structure. On Windows, the installation contains directories such as Bin32 and Documentation. On Unix, these directories are called bin32 and documentation.

  The documentation often uses the mixed case forms of directory names. Usually, you can convert a mixed case directory name to lowercase for the equivalent directory name on Unix platforms. For example, the directory MobiLink is mobilink on Unix platforms.

- **Slashes separating parts of directory names** The documentation uses backslashes as the directory separator. For example, the PDF form of the documentation is found in the directory \install-dir\Documentation\en\pdf, which is the Windows form.

  On Unix platforms, replace the backslash with the forward slash. The PDF documentation is found in the directory 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 platforms, executable file names have no suffix.

  For example, on Windows, the network database server is dbsrv11.exe. On Unix, Linux, and Mac OS X, it is dbsrv11.
samples-dir  The installation process allows you to choose where to install the samples that are included with SQL Anywhere, and the documentation refers to this location using the convention `samples-dir`.

After installation is complete, the environment variable SQLANYSAMP11 specifies the location of the directory containing the samples (`samples-dir`). From the Windows Start menu, choosing Programs » SQL Anywhere 11 » Sample Applications And Projects opens a Windows Explorer window in this directory.

For more information about the default location of `samples-dir`, by operating system, see “Samples directory” [SQL Anywhere Server - Database Administration].

Environment variables  The documentation refers to setting environment variables. On Windows, environment variables are referred to using the syntax `%ENVVAR%`. On Unix, Linux, and Mac OS X, environment variables are referred to using the syntax `$ENVVAR` or `${ENVVAR}`.

Unix, Linux, and Mac OS X environment variables are stored in shell and login startup files, such as `.cshrc` or `.tcshrc`.

Graphic icons

The following icons are used in this documentation.

- A client application.

- A database server, such as Sybase SQL Anywhere.

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

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

Contacting the documentation team

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

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

Finding out more and requesting technical support

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

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

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

The newsgroups are located on the forums.sybase.com news server. The newsgroups include the following:

- sybase.public.sqlanywhere.general
- sybase.public.sqlanywhere.linux
- sybase.public.sqlanywhere.mobilink
- sybase.public.sqlanywhere.product_futures_discussion
- sybase.public.sqlanywhere.replication
- sybase.public.sqlanywhere.ultralite
- ianywhere.public.sqlanywhere.qanywhere
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, as well as other staff, assist on the newsgroup service when they have time available. They offer their help on a volunteer basis and may not be available on a regular basis to provide solutions and information. Their ability to help is based on their workload.
Part I. Designing and Creating Databases

This part describes key concepts and strategies for designing and building databases. It covers issues of database design as well as the mechanics of working with tables, views, and indexes. It also includes material on referential integrity and transactions.
CHAPTER 1

Designing and creating your database

Contents

Introduction to database design ................................................................. 4
Database design concepts ................................................................................. 5
The design process ......................................................................................... 11
Designing the database table properties ......................................................... 24
Tutorial: Creating a SQL Anywhere database ............................................... 28
Introduction to database design

While designing a database is not a difficult task for small and medium sized databases, it is an important one. Bad database design can lead to an inefficient and possibly unreliable database system. Because client applications are built to work on specific parts of a database, and rely on the database design, a bad design can be difficult to revise at a later date.

For more information, you may also want to consult an introductory book on database design.
Database design concepts

In designing a database, you plan what things you want to store information about, and what information you will keep about each one. You also determine how these things are related. In the common language of database design, what you are creating during this step is a conceptual database model.

Entities and relationships

The distinguishable objects or things that you want to store information about are called entities. The associations between them are called relationships. In the language of database description, you can think of entities as nouns and relationships as verbs.

Conceptual models are useful because they make a clean distinction between the entities and relationships. These models hide the details involved in implementing a design in any particular database management system. They allow you to focus on fundamental database structure. Hence, they also form a common language for the discussion of database design.

Entity-relationship diagrams

The main component of a conceptual database model is a diagram that shows the entities and relationships. This diagram is commonly called an entity-relationship diagram. Many people use the name entity-relationship modeling to refer to the task of creating a conceptual database model.

Conceptual database design is a top-down design method. Tools such as Sybase PowerDesigner that help you pursue this method. This chapter is an introductory chapter only, but it does contain enough information for the design of straightforward databases.

Entities

An entity is the database equivalent of a noun. Distinguishable objects such as employees, order items, departments, and products are all examples of entities. In a database, a table represents each entity. The entities that you build into your database arise from the activities for which you will be using the database, such as tracking sales calls and maintaining employee information.

Attributes

Each entity contains a number of attributes. Attributes are particular characteristics of the things that you would like to store. For example, in an employee entity, you might want to store an employee ID number, first and last names, an address, and other information that pertains to a particular employee. Attributes are also known as properties.

You depict an entity using a rectangular box. Inside, you list the attributes associated with that entity.
An **identifier** is one or more attributes on which all the other attributes depend. It uniquely identifies an item in the entity. Underline the names of attributes that you want to form part of an identifier.

In the Employee entity, above, the Employee Number uniquely identifies an employee. All the other attributes store information that pertains only to that one employee. For example, an employee number uniquely determines an employee's name and address. Two employees might have the same name or the same address, but you can make sure that they don't have the same employee number. Employee Number is underlined to show that it is an identifier.

It is good practice to create an identifier for each entity. As will be explained later, these identifiers become primary keys within your tables. Primary key values must be unique and cannot be NULL or undefined. They identify each row in a table uniquely and improve the performance of the database server.

### Relationships

A **relationship** between entities is the database equivalent of a verb. An employee is a member of a department, or an office is located in a city. As will be explained later, relationships in a database may appear as foreign key relationships between tables, or may appear as separate tables themselves.

The relationships in the database are an encoding of rules or practices that govern the data in the entities. If each department has one department head, you can create a one-to-one relationship between departments and employees to identify the department head.

Once a relationship is built into the structure of the database, there is no provision for exceptions. There is nowhere to put a second department head. Duplicating the department entry would involve duplicating the department ID, which is the identifier. Duplicate identifiers are not allowed.

**Tip**

Strict database structure can benefit you because it can eliminate inconsistencies, such as a department with two managers. On the other hand, you as the designer should make your design flexible enough to allow some expansion for unforeseen uses. Extending a well-designed database is usually not too difficult, but modifying the existing table structure can render an entire database and its client applications obsolete.

**Cardinality of relationships**

There are three kinds of relationships between tables. These correspond to the **cardinality** (number) of the entities involved in the relationship.

- **One-to-one relationships** You depict a relationship by drawing a line between two entities. The line may have other markings on it such as the two little circles shown below. Later sections explain the purpose of these marks. In the following diagram, one employee manages one department.
- **One-to-many relationships**  The fact that one item contained in Entity 1 can be associated with multiple entities in Entity 2 is denoted by the multiple lines forming the attachment to Entity 2. In the following diagram, one office can have many phones.

- **Many-to-many relationships**  In this case, draw multiple lines for the connections to both entities. This means that one warehouse can hold many different parts, and one type of part can be stored at many warehouses.

**Roles**

You can describe each relationship with two roles. Roles are verbs or phrases that describe the relationship from each point of view. For example, a relationship between employees and departments might be described by the following two roles.

1. An employee is a member of a department.
2. A department contains an employee.

Roles are very important because they afford you a convenient and effective means of verifying your work.

**Tip**

Whether reading from left-to-right or from right-to-left, the following rule makes it easy to read these diagrams: read the name of the first entity, the role next to the first entity, the cardinality from the connection to the second entity, and the name of the second entity.

In the diagram above, reading left to right, each Employee is a member of one Department. Reading right to left, a Department contains many Employees.
Mandatory elements

The little circles just before the end of the line that denotes the relation serve an important purpose. A circle means that an element can exist in the one entity without a corresponding element in the other entity.

If a cross bar appears in place of the circle, that entity must contain at least one element for each element in the other entity. An example will clarify these statements.

![Diagram](image)

This diagram corresponds to the following four statements.
1. A publisher publishes zero or more books.
2. A book is published by exactly one publisher.
3. A book is written by one or more authors.
4. An author writes zero or more books.

Tip

Think of the little circle as the digit 0 and the cross bar as the number one. The circle means at least zero. The cross bar means at least one.

Reflexive relationships

Sometimes, a relationship will exist between entries in a single entity. In this case, the relationship is said to be reflexive. Both ends of the relationship attach to a single entity.

![Diagram](image)
This diagram corresponds to the following two statements.

1. An employee reports to at most one other employee.
2. An employee manages zero or more employees.

Notice that in the case of this relation, it is essential that the relation be optional in both directions. Some employees are not managers. Similarly, at least one employee should head the organization and hence report to no one.

Naturally, you would also like to specify that an employee cannot be his or her own manager. This restriction is a type of business rule. Business rules are discussed later as part of “The design process” on page 11.

**Changing many-to-many relationships into entities**

When you have attributes associated with a relationship, rather than an entity, you can change the relationship into an entity. This situation sometimes arises with many-to-many relationships, when you have attributes that are particular to the relationship and so you cannot reasonably add them to either entity.

Suppose that your parts inventory is located at a number of different warehouses. You have drawn the following diagram.

![](image)

But you want to record the quantity of each part stored at each location. This attribute can only be associated with the relationship. Each quantity depends on both the parts and the warehouse involved. To represent this situation, you can redraw the diagram as follows:

![](image)

Notice the following details of the transformation:

1. Two new relations join the relation entity with each of the two original entities. They inherit their names from the two roles of the original relationship: stored at and contains, respectively.
2. Each entry in the Inventory entity demands one mandatory entry in the Part entity and one mandatory entry in the Warehouse entity. These relationships are mandatory because a storage relationship only makes sense if it is associated with one particular part and one particular warehouse.

3. The new entity is dependent on both the Part entity and on the Warehouse entity, meaning that the new entity is identified by the identifiers of both of these entities. In this new diagram, one identifier from the Part entity and one identifier from the Warehouse entity uniquely identify an entry in the Inventory entity.

Do not add either a Part Number or Warehouse ID attribute to the Inventory entity. Each entry in the Inventory entity does depend on both a particular part and a particular warehouse.
The design process

There are five major steps in the design process.

- “Step 1: Identify entities and relationships” on page 11
- “Step 2: Identify the required data” on page 14
- “Step 3: Normalize the data” on page 15
- “Step 4: Resolve the relationships” on page 19
- “Step 5: Verify the design” on page 22

For more information about implementing the database design, see “Working with database objects” on page 35.

Step 1: Identify entities and relationships

To identify the entities in your design and their relationship to each other

1. Define high-level activities

   Identify the general activities for which you will use this database. For example, you may want to keep track of information about employees.

2. Identify entities

   For the list of activities, identify the subject areas you need to maintain information about. These subjects will become entities. For example, hire employees, assign to a department, and determine a skill level.

3. Identify relationships

   Look at the activities and determine what the relationships will be between the entities. For example, there is a relationship between parts and warehouses. Define two roles to describe each relationship.

4. Break down the activities

   You started out with high-level activities. Now, examine these activities more carefully to see if some of them can be broken down into lower-level activities. For example, a high-level activity such as maintaining employee information can be broken down into:
   - Add new employees.
   - Change existing employee information.
   - Delete terminated employees.

5. Identify business rules

   Look at your business description and see what rules you follow. For example, one business rule might be that a department has one and only one department head. These rules will be built into the structure of the database.
Entity and relationship example

ACME Corporation is a small company with offices in five locations. Currently, 75 employees work for ACME. The company is preparing for rapid growth and has identified nine departments, each with its own department head.

To help in its search for new employees, the personnel department has identified 68 skills that it believes the company will need in its future employee base. When an employee is hired, the employee's level of expertise for each skill is identified.

Define high-level activities

Some of the high-level activities for ACME Corporation are:

- Hire employees.
- Terminate employees.
- Maintain personal employee information.
- Maintain information on skills required for the company.
- Maintain information on which employees have which skills.
- Maintain information on departments.
- Maintain information on offices.

Identify the entities and relationships

Identify the entities (subjects) and the relationships (roles) that connect them. Create a diagram based on the description and high-level activities.

Use boxes to show entities and lines to show relationships. Use the two roles to label each relationship. You should also identify those relationships that are one-to-many, one-to-one, and many-to-many using the appropriate annotation.

Following is a rough entity-relationship diagram. It will be refined throughout the example.
Break down the high-level activities

The following lower-level activities below are based on the high-level activities listed above:

- Add or delete an employee.
- Add or delete an office.
- List employees for a department.
- Add a skill to the skill list.
- Identify the skills of an employee.
- Identify an employee's skill level for each skill.
- Identify all employees that have the same skill level for a particular skill.
- Change an employee's skill level.

These lower-level activities can be used to identify if any new tables or relationships are needed.

Identify business rules

Business rules often identify one-to-many, one-to-one, and many-to-many relationships.

The kind of business rules that may be relevant include the following:

- There are now five offices; expansion plans allow for a maximum of ten.
- Employees can change department or office.
- Each department has one department head.
- Each office has a maximum of three telephone numbers.
- Each telephone number has one or more extensions.
- When an employee is hired, the level of expertise in each of several skills is identified.
- Each employee can have from three to twenty skills.
An employee may or may not be assigned to an office.

**Step 2: Identify the required data**

To identify the required data
1. Identify supporting data.
2. List all the data you need to track.
3. Set up data for each entity.
4. List the available data for each entity. The data that describes an entity (subject) answers the questions who, what, where, when, and why.
5. List any data required for each relationship (verb).
6. List the data, if any, that applies to each relationship.

Identify supporting data
The supporting data you identify will become the names of the attributes of the entity. For example, the data below might apply to the Employee entity, the Skill entity, and the Expert In relationship.

<table>
<thead>
<tr>
<th>Employee</th>
<th>Skill</th>
<th>Expert In</th>
</tr>
</thead>
<tbody>
<tr>
<td>Employee ID</td>
<td>Skill ID</td>
<td>Skill level</td>
</tr>
<tr>
<td>Employee first name</td>
<td>Skill name</td>
<td>Date skill was acquired</td>
</tr>
<tr>
<td>Employee last name</td>
<td>Description of skill</td>
<td></td>
</tr>
<tr>
<td>Employee department</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Employee office</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Employee address</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

If you make a diagram of this data, it will look something like this picture:

![Diagram](image)

Observe that not all of the attributes you listed appear in this diagram. The missing items fall into two categories:
1. Some are contained implicitly in other relationships; for example, Employee department and Employee office are denoted by the relations to the Department and Office entities, respectively.

2. Others are not present because they are associated not with either of these entities, but rather the relationship between them. The above diagram is inadequate.

The first category of items will fall naturally into place when you draw the entire entity-relationship diagram. You can add the second category by converting this many-to-many relationship into an entity.

The new entity depends on both the Employee and the Skill entities. It borrows its identifiers from these entities because it depends on both of them.

**Notes**

- When you are identifying the supporting data, be sure to refer to the activities you identified earlier to see how you will access the data.
  
  For example, you may need to list employees by first name in some situations and by last name in others. To accommodate this requirement, create a First Name attribute and a Last Name attribute, rather than a single attribute that contains both names. With the names separate, you can later create two indexes, one suited to each task.

- Choose consistent names. Consistency makes it easier to maintain your database and easier to read reports and output windows.
  
  For example, if you choose to use an abbreviated name such as Emp_status for one attribute, you should not use a full name, such as Employee_ID, for another attribute. Instead, the names should be Emp_status and Emp_ID.

- At this stage, it is not critical that the data is associated with the correct entity. You can normalize the data to verify it is associated with the correct entity or relationship. See “Step 3: Normalize the data” on page 15.

**Step 3: Normalize the data**

Normalization is a series of tests that eliminate redundancy in the data and make sure the data is associated with the correct entity or relationship. There are five tests. This section presents the first three of them. These three tests are the most important and so are the most frequently used.
Why normalize?
The goals of normalization are to remove redundancy and to improve consistency. For example, if you store a customer's address in multiple locations, it is difficult to update all copies correctly when they move.

For more information about the normalization tests, see a book on database design.

Normal forms
There are several tests for data normalization. When your data passes the first test, it is considered to be in first normal form. When it passes the second test, it is in second normal form, and when it passes the third test, it is in third normal form.

To normalize data in a database
1. List the data.
   ● Identify at least one key for each entity. Each entity must have an identifier.
   ● Identify keys for relationships. The keys for a relationship are the keys from the two entities that it joins.
   ● Check for calculated data in your supporting data list. Calculated data is not normally stored in a relational database.
2. Put the data in first normal form.
   ● If an attribute can have several different values for the same entry, remove these repeated values.
   ● Create one or more entities or relationships with the data that you remove.
3. Put the data in second normal form.
   ● Identify entities and relationships with keys consisting of more than one attribute.
   ● Remove data that depends on only one part of the key.
   ● Create one or more entities and relationships with the data that you remove.
4. Put the data in third normal form.
   ● Remove data that depends on other data in the entity or relationship, not on the key.
   ● Create one or more entities and relationships with the data that you remove.

Data and identifiers
Before you begin to normalize (test your design), simply list the data and identify a unique identifier for each table. The identifier can be made up of one piece of data (attribute) or several (a compound identifier).

The identifier is the set of attributes that uniquely identifies each row in an entity. For example, the identifier for the Employee entity is the Employee ID attribute. The identifier for the Works In relationship consists of the Office Code and Employee ID attributes.

You can make an identifier for each relationship in your database by taking the identifiers from each of the entities that it connects. In the following table, the attributes identified with an asterisk are the identifiers for the entity or relationship.
<table>
<thead>
<tr>
<th>Entity or relationship</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Office</td>
<td>*Office code</td>
</tr>
<tr>
<td></td>
<td>Office address</td>
</tr>
<tr>
<td></td>
<td>Phone number</td>
</tr>
<tr>
<td>Works in</td>
<td>*Office code</td>
</tr>
<tr>
<td></td>
<td>*Employee ID</td>
</tr>
<tr>
<td>Department</td>
<td>*Department ID</td>
</tr>
<tr>
<td></td>
<td>Department name</td>
</tr>
<tr>
<td>Heads</td>
<td>*Department ID</td>
</tr>
<tr>
<td></td>
<td>*Employee ID</td>
</tr>
<tr>
<td>Member of</td>
<td>*Department ID</td>
</tr>
<tr>
<td></td>
<td>*Employee ID</td>
</tr>
<tr>
<td>Skill</td>
<td>*Skill ID</td>
</tr>
<tr>
<td></td>
<td>Skill name</td>
</tr>
<tr>
<td></td>
<td>Skill description</td>
</tr>
<tr>
<td>Expert in</td>
<td>*Skill ID</td>
</tr>
<tr>
<td></td>
<td>*Employee ID</td>
</tr>
<tr>
<td></td>
<td>Skill level</td>
</tr>
<tr>
<td></td>
<td>Date acquired</td>
</tr>
<tr>
<td>Employee</td>
<td>*Employee ID</td>
</tr>
<tr>
<td></td>
<td>Last name</td>
</tr>
<tr>
<td></td>
<td>First name</td>
</tr>
<tr>
<td></td>
<td>Social security number</td>
</tr>
<tr>
<td></td>
<td>Address</td>
</tr>
<tr>
<td></td>
<td>Phone number</td>
</tr>
<tr>
<td></td>
<td>Date of birth</td>
</tr>
</tbody>
</table>

**Putting data in first normal form**

- To test for first normal form, look for attributes that can have repeating values.
Designing and creating your database

- Remove attributes when multiple values can apply to a single item. Move these repeating attributes to a new entity.

In the entity below, Phone number can repeat—an office can have more than one telephone number.

<table>
<thead>
<tr>
<th>Office and phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>Office code</td>
</tr>
<tr>
<td>Office address</td>
</tr>
<tr>
<td>Phone number</td>
</tr>
</tbody>
</table>

Remove the repeating attribute and make a new entity called Telephone. Set up a relationship between Office and Telephone.

Putting data in second normal form

- Remove data that does not depend on the whole key.
- Look only at entities and relationships with an identifier that is composed of more than one attribute. To test for second normal form, remove any data that does not depend on the whole identifier. Each attribute should depend on all of the attributes that comprise the identifier.

In this example, the identifier of the Employee and Department entity is composed of two attributes. Some of the data does not depend on both identifier attributes; for example, the department name depends on only one of those attributes, Department ID, and Employee first name depends only on Employee ID.

<table>
<thead>
<tr>
<th>Employee and Department</th>
</tr>
</thead>
<tbody>
<tr>
<td>Employee ID</td>
</tr>
<tr>
<td>Department ID</td>
</tr>
<tr>
<td>Employee first name</td>
</tr>
<tr>
<td>Employee last name</td>
</tr>
<tr>
<td>Department name</td>
</tr>
</tbody>
</table>

Move the identifier Department ID, which the other employee data does not depend on, to an entity of its own called Department. Also move any attributes that depend on it. Create a relationship between Employee and Department.
Putting data in third normal form

- Remove data that doesn't depend directly on the key.
- To test for third normal form, remove any attributes that depend on other attributes, rather than directly on the identifier.

In this example, the Employee and Office entity contains some attributes that depend on its identifier, Employee ID. However, attributes such as Office location and Office phone depend on another attribute, Office code. They do not depend directly on the identifier, Employee ID.

Remove Office code and those attributes that depend on it. Make another entity called Office. Then, create a relationship that connects Employee with Office.

Step 4: Resolve the relationships

When you finish the normalization process, your design is almost complete. All you need to do is to generate the physical data model that corresponds to your conceptual data model. This process is also known as resolving the relationships, because a large portion of the task involves converting the relationships in the conceptual model into the corresponding tables and foreign-key relationships.

Whereas the conceptual model is largely independent of implementation details, the physical data model is tightly bound to the table structure and options available in a particular database application. In this case, that application is SQL Anywhere.
Resolving relationships that do not carry data

To implement relationships that do not carry data, you define foreign keys. A **foreign key** is a column or set of columns that contains primary key values from another table. The foreign key allows you to access data from more than one table at one time.

A database design tool such as PowerDesigner can generate the physical data model for you. However, if you are doing it yourself there are some basic rules that help you decide where to put the keys.

- **One to many**  A one-to-many relationship always becomes an entity and a foreign key relationship.

Notice that entities become tables. Identifiers in entities become (at least part of) the primary key in a table. Attributes become columns. In a one-to-many relationship, the identifier in the *one* entity will appear as a new foreign key column in the *many* table.

In this example, the Employee entity becomes an Employees table. Similarly, the Department entity becomes a Departments table. A foreign key called Department ID appears in the Employee table.

- **One to one**  In a one-to-one relationship, the foreign key can go into either table. If the relationship is mandatory on one side, but optional on the other, it should go on the optional side. In this example, put the foreign key (Vehicle ID) in the Truck table because a vehicle does not have to be a truck.

The above entity-relationship model thus resolves to the database base structure below.
• **Many to many**  In a many-to-many relationship, a new table is created with two foreign keys. This arrangement is necessary to make the database efficient.

The new Storage Location table relates the Part and Warehouse tables.

**Resolving relationships that carry data**

Some of your relationships may carry data. This situation often occurs in many-to-many relationships.

If this is the case, each entity resolves to a table. Each role becomes a foreign key that points to another table.
The Inventory entity borrows its identifiers from the Part and Warehouse tables, because it depends on both of them. Once resolved, these borrowed identifiers form the primary key of the Inventory table.

Tip
A conceptual data model simplifies the design process because it hides a lot of details. For example, a many-to-many relationship always generates an extra table and two foreign key references. In a conceptual data model, you can usually denote all of this structure with a single connection.

Step 5: Verify the design

Before you implement your design, you need to make sure that it supports your needs. Examine the activities you identified at the start of the design process and make sure you can access all of the data that the activities require.

- Can you find a path to get the information you need?
- Does the design meet your needs?
- Is all of the required data available?

If you can answer yes to all the questions above, you are ready to implement your design.

Final design

Applying steps 1 through 3 to the database for the little company produces the following entity-relationship diagram. This database is now in third normal form.
The corresponding physical data model appears below.
Designing the database table properties

The database design specifies which tables you have and what columns each table contains. This section describes how to specify each column's properties.

For each column, you must decide the column name, the data type and size, whether or not NULL values are allowed, and whether you want the database to restrict the values allowed in the column.

Choosing column names

A column name can be any set of letters, numbers, or symbols. However, you must enclose a column name in double quotes if it contains characters other than letters, numbers, or underscores, if it does not begin with a letter, or if it is the same as a keyword.

For a list of keywords, see “Reserved words” [SQL Anywhere Server - SQL Reference].

Choosing data types for columns

The following data types are available in SQL Anywhere:

- Integer data types
- Decimal data types
- Floating-point data types
- Character data types
- Binary data types
- Date/time data types
- Domains (user-defined data types)

For more information about data types, see “SQL data types” [SQL Anywhere Server - SQL Reference].

Any of the character or binary string data types such as CHAR, VARCHAR, LONG VARCHAR, NCHAR, BINARY, VARBINARY, and so on, can be used to store large objects such as images, word-processing documents, and sound files.

For more information about BLOB storage, see “Storing BLOBs in the database” on page 25.

NULL and NOT NULL

If the column value is mandatory for a row, you define the column as being NOT NULL. Otherwise, the column is allowed to contain the NULL value, which represents no value. The default in SQL Anywhere is to allow NULL values, but you should explicitly declare columns NOT NULL unless there is a good reason to allow NULL values.

The SQL Anywhere sample database has a table called Departments, which has columns named DepartmentID, DepartmentName, and DepartmentHeadID. Its definition is as follows:
If you specify NOT NULL, a column value must be supplied for every row in the table.

For more information about the NULL value, see “NULL value” [SQL Anywhere Server - SQL Reference]. For information about its use in comparisons, see “Search conditions” [SQL Anywhere Server - SQL Reference].

### Storing BLOBs in the database

A BLOB is an uninterpreted string of bytes or characters, stored as a value in a column. Common examples of a BLOB are picture or sound files. While BLOBs are typically large, you can store them in any character string or binary string data type such as CHAR, VARCHAR, NCHAR, BINARY, VARBINARY, and so on. Choose your data type and length depending on the content and length of BLOBs you expect to store.

**Note**

While a character large object is commonly called a CLOB, a binary large object is called a BLOB, and the combination of both is called a LOB, the documentation uses only the acronym BLOB.

### BLOB storage

When you create a column for storing BLOB values, you can control aspects of their storage. For example, you can specify that BLOBs up to a specified size be stored in the row (inline), while larger BLOBs are stored outside of the row in table extension pages. Additionally, you can specify that for BLOBs stored outside of the row, the first $n$ bytes of the BLOB, also referred to as the prefix, are duplicated in the row. These storage aspects are controlled by the INLINE and PREFIX settings specified in the CREATE TABLE and ALTER TABLE statements. The values you specify for these settings can have unanticipated impacts on performance or disk storage requirements.

If neither INLINE nor PREFIX is specified, or if INLINE USE DEFAULT or PREFIX USE DEFAULT is specified, default values are applied as follows:

- For character data type columns, such as CHAR, NCHAR, LONG VARCHAR, and XML, the default value of INLINE is 256, and the default value of PREFIX is 8.
- For binary data type columns, such as BINARY, LONG BINARY, VARBINARY, BIT, VARBIT, LONG VARBIT, BIT VARYING, and UUID, the default value of INLINE is 256, and the default value of PREFIX is 0.

It is strongly recommended that you not set INLINE and PREFIX values unless there are specific requirements for which the defaults are not sufficient. The default values have been chosen to balance performance and disk space requirements. For example, if you set INLINE to a large value, and all the
BLOBs are stored inline, row processing performance may degrade. If you set PREFIX too high, you increase the amount of disk space required to store BLOBs since the prefix data duplicates a portion of the BLOB.

If you do decide to set INLINE or PREFIX values, the INLINE length must not exceed the length of the column. Likewise, the PREFIX length, must not exceed the INLINE length.

The prefix data for a compressed column is stored uncompressed, so if all of the data required to satisfy a request is stored in the prefix, no decompression is necessary.

For information about the defaults for the INLINE and PREFIX clauses, see “CREATE TABLE statement” [SQL Anywhere Server - SQL Reference].

**BLOB sharing**

If a BLOB exceeds the inline size, and requires more than one database page for storage, the database server stores it so that it can be referenced by other rows in the same table, when possible. This is known as BLOB sharing. BLOB sharing is handled internally and is intended to reduce unnecessary duplication of BLOBs in the database.

BLOB sharing only occurs when you set values of one column to be equal to those of another column. For example, UPDATE t column1=column2; In this example, if column2 contains BLOBs, instead of duplicating them in column1, pointers to the values in column2 are used instead.

When a BLOB is shared, the database server keeps track of how many other references there are to the BLOB. Once the database server determines that a BLOB is no longer referenced within a table, the BLOB is removed.

If a BLOB is shared between two uncompressed columns and one of those columns is then compressed, the BLOB will no longer be shared.

**Choosing whether to compress columns**

CHAR, VARCHAR, and BINARY columns can be compressed to save disk space. For example, you can compress a column in which large BLOB files such as BMPs and TIFFs are stored. Compression is achieved using the deflate compression algorithm. This is the same algorithm used by the COMPRESS function, and is also the same algorithm used for Windows ZIP files.

Compressed columns can reside inside of encrypted tables. In this case, data is first compressed, and then encrypted.

Do not use column compression on columns containing values under 130 bytes, or values that are already in a compressed format, such as JPG files. Attempting to compress columns that contain values that are already compressed may actually increase the amount of storage required for the column.

To compress columns, use the COMPRESS clause of the CREATE TABLE and ALTER TABLE statements.

You can determine the benefits you are getting by compressing columns using the sa_column_stats system procedure.

See also

- “CREATE TABLE statement” [SQL Anywhere Server - SQL Reference]
Choosing constraints

Although the data type of a column restricts the values that are allowed in that column (for example, only numbers or only dates), you may want to further restrict the allowed values.

You can restrict the values of any column by specifying a CHECK constraint. You can use any valid condition that could appear in a WHERE clause to restrict the allowed values. Most CHECK constraints use either the BETWEEN or IN condition.

For more information about valid conditions, see “Search conditions” [SQL Anywhere Server - SQL Reference]. For more information about assigning constraints to tables and columns, see “Ensuring data integrity” on page 93.
Tutorial: Creating a SQL Anywhere database

This tutorial describes how to use Sybase Central to create a simple database, modeled on the Products, SalesOrderItems, SalesOrders, and Customers tables of the SQL Anywhere sample database.

For information about the SQL Anywhere sample database, see “SQL Anywhere sample database” [SQL Anywhere 11 - Introduction].

For information about the principles of database design, see “Designing and creating your database” on page 3.

Lesson 1: Create a database file

In this lesson, you create a database file to hold your database. The database file contains system tables and other system objects that are common to all databases; later you will add tables. Other options are available when creating a database, but the default choices are typically sufficient.

To create a new database file

1. Start Sybase Central.
2. From the Tools menu, choose SQL Anywhere 11 » Create Database.
3. Read the information on the Welcome page, and then click Next.
4. Select Create A Database On This Computer, and then click Next.
5. In the Save The Main Database File To The Following File field, type c:\temp\mysample.db.
   
   If your temporary directory is somewhere other than c:\temp, specify the appropriate path.
    
6. Click Finish.
7. Click Close.

See also

- “Tutorial: Creating a SQL Anywhere database” on page 28
- “Lesson 2: Connect to the database” on page 28
- “Lesson 3: Create a table” on page 30
- “Lesson 4: Set column properties” on page 31
- “Lesson 5: Create relationships between tables using foreign keys” on page 32

Lesson 2: Connect to the database

In this lesson, you use Sybase Central to connect to the database file you created. However, if you just finished creating the database, you are already connected to it, and you can skip directly to the next lesson, where you learn to create tables. See “Lesson 3: Create a table” on page 30.

To connect to your database

1. Start Sybase Central.
2. From the Connections menu, choose Connect With SQL Anywhere 11.
3. In the User ID field, type DBA. This is the default value for a new database.
4. In the Password field, type sql. This is the default value for a new database.
5. Select None in the profile options area.
6. Click the Database tab and in the Database File field type the full path of your database file. For example, type:

   c:\temp\mysample.db

7. Click OK.

The database starts, and information about the database and the database server it is running on appear in Sybase Central.

See also

- “Tutorial: Creating a SQL Anywhere database” on page 28
Lesson 3: Create a table

In this lesson, you create a table named Products.

For information about designing tables, see “Designing the database table properties” on page 24.

To create a table

1. In the right pane of Sybase Central, double-click Tables.
2. Right click the Tables pane and choose New » Table.
3. In the What Do You Want To Name The New Table field, type Products.
4. Click Finish.

The database server creates the table using defaults, and then displays the Columns tab in the right pane. Each row in the Columns tab defines a column in the table.

5. In the first column of your table, click PKey and type ProductID in the Name field.

A primary key can comprise several columns concatenated together. For the purposes of this tutorial, however, the primary key contains only one column, which has a value that is automatically incremented for each row added to the table. The autoincrement property ensures that values are unique—a requirement for primary keys. See “Primary keys” [SQL Anywhere 11 - Introduction].

6. In the Data Type list, select Integer.
7. Click the ellipsis (three dots) button.
8. Click the Value tab and choose Default Value » System-Defined » Autoincrement
9. Click OK.
10. From the File menu, choose New » Column.
11. Complete the following fields:
    - In the Name field, type ProductName.
    - In the Data Type list, select Char.
    - In the Size list, select 15.
12. From the File menu, choose Save.

See also

- “Tutorial: Creating a SQL Anywhere database” on page 28
- “Lesson 1: Create a database file” on page 28
- “Lesson 2: Connect to the database” on page 28
- “Lesson 4: Set column properties” on page 31
- “Lesson 5: Create relationships between tables using foreign keys” on page 32
Lesson 4: Set column properties

In this lesson, you learn how to add a NOT NULL constraint on a column.

NULL and NOT NULL

Every product in the Products table must have an associated product ID. Therefore, for each row in the Products table, the ID column must contain a value.

You can ensure that each row in the column contains a value for the ID column by defining the column as NOT NULL.

By default, columns allow NULLs, but you should explicitly declare columns NOT NULL unless there is a good reason to allow NULLs. See “NULL value” [SQL Anywhere Server - SQL Reference].

To define a column as NOT NULL

1. In the left pane of Sybase Central, double-click Tables.
2. Click Products, and then click the Columns tab in the right pane.
3. Select the ProductID column.
4. From the File menu, choose Properties.
5. Click the Constraints tab and select Values Cannot Be NULL.
6. Click OK.

This lesson and the previous lesson introduced the basic concepts you need to know to create database tables. You can practice what you have learned by adding some more tables to your database. These tables are used in the subsequent lesson in this chapter.

Add the following tables to your database:

- **Customers** Add a table named Customers, with the following columns:
  - ID An identification number for each customer. This column has integer data type, and is the primary key. Make this an autoincrement column.
  - CompanyName The company name. This column is a char data type, with a maximum length of 35 characters.

- **SalesOrders** Add a table named SalesOrders, with the following columns:
  - ID An identification number for each sales order. This column has integer data type, and is the primary key. Make this an autoincrement column.
  - OrderDate The date on which the order was placed. This column has date data type.
  - CustomerID The identification number of the customer who placed the sales order. This column has integer data type.

- **SalesOrderItems** Add a table named SalesOrderItems with the following columns:
  - ID The identification number of the sales order of which the item is a part. This column has integer data type, and should be identified as a primary key column.
Designing and creating your database

- **LineID**  An identification number for each sales order. This column has integer data type, and should be identified as a primary key column.

- **ProductID** The identification number for the product being ordered. This column has integer data type.

You have now created four tables in your database. The tables are not yet related in any way. In the next lesson, you define foreign keys to relate the tables to one another.

See also

- “Tutorial: Creating a SQL Anywhere database” on page 28
- “Lesson 1: Create a database file” on page 28
- “Lesson 2: Connect to the database” on page 28
- “Lesson 3: Create a table” on page 30
- “Lesson 5: Create relationships between tables using foreign keys” on page 32

**Lesson 5: Create relationships between tables using foreign keys**

In this lesson, you learn about creating relationships between tables using foreign keys. You use the tables you created in the previous lessons of the Creating a SQL Anywhere tutorial for this lesson.

For information about designing relationships, see “Entities and relationships” on page 5.

**To create a foreign key**

1. In the left pane of Sybase Central, double-click **Tables**.
2. In the left pane, click the **SalesOrders** table to select it.
3. In the right pane, select the **Constraints** tab.
4. Choose **File** » **New** » **Foreign Key**.
5. In the To Which Table Do You Want This Foreign Key To Refer list, select the **Products** table.
6. In the What Do You Want To Name The New Foreign Key field, type **ProductIDkey**.
7. Click **Next** and for Do You Want This Foreign Key To Reference The Primary Key Or A Unique Constraint choose **Primary Key**.
8. In the Foreign Column list, click **ProductID**.
9. Click **Finish**.

This completes this introductory section on creating relational databases.

For more information about designing databases, see “Entities and relationships” on page 5.

See also

- “Tutorial: Creating a SQL Anywhere database” on page 28
- “Lesson 1: Create a database file” on page 28
- “Lesson 2: Connect to the database” on page 28
● “Lesson 3: Create a table” on page 30
● “Lesson 4: Set column properties” on page 31
CHAPTER 2

Working with database objects

Contents

Introduction to working with database objects ............................................................. 36
Working with tables ..................................................................................................... 38
Working with temporary tables .................................................................................... 52
Working with views ...................................................................................................... 54
Working with indexes ................................................................................................... 86
Introduction to working with database objects

With the SQL Anywhere tools, you can create a database to hold your data. Once the database is created, you can begin adding database objects, such as tables or users, and you can set overall 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: you can think of the schema as an empty database.

Procedures and triggers are also database objects, but they are discussed in “Using procedures, triggers, and batches” on page 805.

If you want more conceptual information before you begin, see:

- “Designing and creating your database” on page 3
- “Ensuring data integrity” on page 93
- “Using Sybase Central” [SQL Anywhere Server - Database Administration]
- “Using Interactive SQL” [SQL Anywhere Server - Database Administration]

Setting properties for database objects

Most database objects, including the database itself, have properties that you can either view or set. Some properties that reflect settings that were chosen when you created the database are non-configurable.

The best way to view and set properties is to use the properties windows in Sybase Central.

If you are not using Sybase Central, properties can be specified when you create the object with a CREATE statement. If the object already exists, you can modify properties with an ALTER statement.

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

1. In Sybase Central, open the folder in which the object resides.
2. Select the object. The object's properties appear in the right pane of Sybase Central.
3. 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

- “Database properties” [SQL Anywhere Server - Database Administration]

Viewing system objects in a database

SQL Anywhere provides many system objects such as system tables, system views, stored procedures, and domains. These objects 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. Start the database server.
2. Connect to a database as a user with DBA authority.
4. 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.

```sql
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].

Creating tables

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

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 52.

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. 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 “Setting properties for database objects” on page 36.

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 “Altering regular views” on page 60.

- **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 “Creating materialized views” on page 73.
For an overview of how altering database objects affects view dependencies, see “View dependencies” on page 55.

**Altering 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 55.

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. Connect to the database as a user with DBA authority, or as 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 “Enabling and disabling materialized views” on page 80.

---

**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**.

It is also necessary to click **Save** or choose **File » Save**. Changes are not made to the table until then.

---

**See also**

- “Enabling and disabling materialized views” on page 80
- “Ensuring data integrity” on page 93
- “View dependencies” on page 55
Altering tables (SQL)

You can alter tables in Interactive SQL using the ALTER TABLE statement. 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 55.

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, and there are dependent materialized views, disable them using the ALTER MATERIALIZED VIEW ... DISABLE statement for 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 [compatibility]” [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]
- “Altering regular views” on page 60
- “Enabling and disabling materialized views” on page 80
- “ALTER MATERIALIZED VIEW statement” [SQL Anywhere Server - SQL Reference]
- “Ensuring data integrity” on page 93
- “View dependencies” on page 55

## Dropping tables

This section describes how to drop tables from a database. You can use either Sybase Central or Interactive SQL to perform this task. 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 “Altering regular views” on page 60.

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 “Creating materialized views” on page 73.

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 55.

### To drop a table (Sybase Central)

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:
   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

```sql
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 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:

```sql
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]

**Managing primary keys**

A **primary key** comprises a single column, or a set of columns combined together, whose values uniquely identify 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.

Managing primary keys (Sybase Central)
A primary key is a column, or set of columns, that is used to uniquely identify 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. 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.

Managing 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 a 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:

```sql
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:

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

See “ALTER TABLE statement” [SQL Anywhere Server - SQL Reference], and “Managing primary keys (Sybase Central)” on page 44.

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.

Managing 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, prior to 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. 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. 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. 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.
Managing foreign keys (SQL)

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.

A table can only have one primary key defined, but it can have as many foreign keys as necessary.

To alter the foreign key of an existing table (SQL)

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

Example 1

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 )
    );

Example 2

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 );

Example 3

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 “ALTER TABLE statement” [SQL Anywhere Server - SQL Reference].

In Sybase Central, you can also specify properties in the Create Foreign Key Wizard or on the Foreign Key Properties window. See “Managing foreign keys (Sybase Central)” on page 46.

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 as well as 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].

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 “Predicate analysis” on page 548. 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(30,22)
    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 NUMERIC(30,22). 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 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 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.

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.

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.

Global temporary tables

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 of 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.

See also

- “CREATE TABLE statement” [SQL Anywhere Server - SQL Reference]
- “DECLARE LOCAL TEMPORARY TABLE statement” [SQL Anywhere Server - SQL Reference]
Creating temporary tables

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

To create a table (Sybase Central)

1. 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 was cached in case the SELECT statement is executed again. 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. Consequently, 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.

Quick comparison between regular views, materialized views, and base tables

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

<table>
<thead>
<tr>
<th>Capability</th>
<th>Regular views</th>
<th>Materialized views</th>
<th>Base tables</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allow access permissions</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Allow SELECT</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Allow UPDATE</td>
<td>Some</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Allow INSERT</td>
<td>Some</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Allow DELETE</td>
<td>Some</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Allow dependent views</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Allow indexes</td>
<td>No</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Allow integrity constraints</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Allow keys</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

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, particularly 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 of 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:

```sql
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 “Enabling and disabling materialized views” on page 80.

2. The database server obtains exclusive schema locks on the object being altered, as well as 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 is not permitted if it is referenced by any enabled materialized views. You can add foreign keys to the table, however (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:

```sql
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:

```sql
CREATE VIEW view-name AS query;
```
Updating regular views

UPDATE, INSERT, and DELETE statements are allowed on some regular views, depending on the view's SELECT statement.

You cannot update views containing aggregate functions, such as COUNT(*). Neither can you update views containing a GROUP BY clause in the SELECT statement, or views containing a UNION statement. In all these cases, there is no way to translate the UPDATE into an action on the underlying tables.

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 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;
```

<table>
<thead>
<tr>
<th>EmployeeID</th>
<th>GivenName</th>
<th>Surname</th>
<th>DepartmentID</th>
</tr>
</thead>
<tbody>
<tr>
<td>129</td>
<td>Philip</td>
<td>Chin</td>
<td>200</td>
</tr>
<tr>
<td>195</td>
<td>Marc</td>
<td>Dill</td>
<td>200</td>
</tr>
<tr>
<td>299</td>
<td>Rollin</td>
<td>Overbey</td>
<td>200</td>
</tr>
<tr>
<td>467</td>
<td>James</td>
<td>Klobucher</td>
<td>200</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

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, "$\text{WITH CHECK OPTION violated for insert/update on base table 'Employees'}\$".

If you had not specified the $\text{WITH CHECK OPTION}$ 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 $\text{WITH CHECK OPTION}$ criteria on SalesEmployees to fail, even if View2 is defined without a $\text{WITH CHECK OPTION}$ clause.

See also
- “SELECT statement” [SQL Anywhere Server - SQL Reference]
- “Summarizing, grouping, and sorting query results” on page 349
- “Working with materialized views” on page 65

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 “Enabling and disabling regular views” on page 63.

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

## Creating 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. 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:

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

See “CREATE VIEW statement” [SQL Anywhere Server - SQL Reference].

## Altering 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 55.

**To alter 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. 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. 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.

```sql
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.

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

See “ALTER VIEW statement” [SQL Anywhere Server - SQL Reference].

Dropping 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 “Altering regular views” on page 60.

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 55.

To drop 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 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.

```sql
DROP VIEW DepartmentSize;
```

See “DROP statement” [SQL Anywhere Server - SQL Reference].
Enabling and disabling regular views

The section describes enabling and disabling regular views. For information about enabling and disabling materialized views, see “Enabling and disabling materialized views” on page 80.

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. Consequently, 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.

Sybase Central

**To enable 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 Recompile And Enable.

**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.

SQL statements

**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.
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.

Examples

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

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

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 291.

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]

Viewing system table data using regular views

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 tables 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, as well as 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. 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:

```sql
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:

```sql
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). Consequently, 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 542.

While you can create indexes on materialized views, you cannot create keys, constraints, triggers, or articles on them.
Types of materialized views (manual and immediate)

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 as soon as 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 “Changing the refresh type for a materialized view” on page 76.

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 68.

Retrieving materialized view information from the database

- **Status and property information** 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].

  See also “Materialized view statuses and properties” on page 68.

- **Database option information** You can retrieve the database options that were stored with a materialized view when it was created by querying the SYSMVOPTION system view. The following statements create a materialized view and then query the database to find out the database options used when creating the view.

```sql
CREATE MATERIALIZED VIEW EmployeeConfid15 AS
  SELECT EmployeeID, Employees.DepartmentID, SocialSecurityNumber, Salary, ManagerID,
         Departments.DepartmentName, Departments.DepartmentHeadID
  FROM Employees, Departments
  WHERE Employees.DepartmentID=Departments.DepartmentID;

SELECT option_name, option_value
FROM SYSMVOPTION JOIN SYSMVOPTIONNAME
WHERE SYSMVOPTION.view_object_id={
```

Copyright © 2008, iAnywhere Solutions, Inc. - SQL Anywhere 11.0.0
SELECT object_id FROM SYSTAB
WHERE table_name='EmployeeConfid15' )
ORDER BY option_name;

**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 “Dropping materialized views” on page 84.

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

**See also**
- “Improving performance with materialized views” on page 550
- “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]

**Deciding 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 concurrency 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 concurrency 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 concurrency 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 83.

○ **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 589, and “Improving performance with materialized views” on page 550.

  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 188, and “Choosing a diagnostic tracing level” on page 201.

See also
- “Types of materialized views (manual and immediate)” on page 66
- “Improving performance with materialized views” on page 550

### 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 “Enabling and disabling materialized
views” on page 80.

**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 “Changing the refresh type for a materialized view” on page 76, and “Additional restrictions for immediate views” on page 72.
- 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 65
Restrictions on materialized views

The following restrictions apply when creating, initializing, and refreshing materialized views, as well as during view matching, as noted below:

- 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.

- 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
  - T-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`
  - `sort_collation=Null`
  - `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`
When a view is refreshed, the connection settings for all of 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 guaranteed 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 550.

**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:

- The view must be uninitialized. See “Materialized view statuses and properties” on page 68.
- The view must have a unique index on non-nullable columns. If it does not, you must add one. See “Creating indexes” on page 89.
- 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:
  - a SUM function over a nullable expression.
  - GROUPING SETS clauses
  - CUBE clauses
  - ROLLUP clauses
  - LEFT OUTER JOIN clauses
  - RIGHT OUTER JOIN clauses
  - FULL OUTER JOIN clauses
  - DISTINCT clauses
  - row limit clauses
  - non-deterministic expressions
  - self and recursive joins
- 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. For example, \( \text{SUM(expression)} + 1 \) is not allowed in the select list.
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 step” on page 553.

- An aggregate function in the select list cannot be referenced in a complex expression.

Creating 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 71.

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. 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.
4. Initialize the materialized view so that it contains data. See “Initializing materialized views” on page 74.

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 “Initializing materialized views” on page 74.

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;

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 “Dropping materialized views” on page 84.

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 11 - Introduction]

Initializing 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 71.

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. 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.

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 “Dropping materialized views” on page 84.

See also
- “CREATE MATERIALIZED VIEW statement” [SQL Anywhere Server - SQL Reference]
- “REFRESH MATERIALIZED VIEW statement” [SQL Anywhere Server - SQL Reference]
- “Enabling and disabling materialized views” on page 80

Refreshing 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
accordingly. 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.

To refresh a manual view (Sybase Central)

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. 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 “Dropping materialized views” on page 84.

Related notes

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 83.

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.

See also

- “Changing the refresh type for a materialized view” on page 76
- “Automating tasks using schedules and events” [SQL AnywhereServer - Database Administration]
- “materialized_view_optimization option [database]” [SQL AnywhereServer - Database Administration]

Changing the refresh type for a materialized 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 72.
An immediate view can be converted to manual at any time by simply changing its refresh type.

**Changing a manual view to an immediate view**

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

For an example of how to change a manual view to an immediate view using the ALTER MATERIALIZED VIEW statement, see “ALTER MATERIALIZED VIEW statement” [SQL Anywhere Server - SQL Reference].

**Changing an immediate view to a manual view**

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. Connect to the database as the owner of the view, or as a user with DBA authority.
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 “Dropping materialized views” on page 84.

**See also**

- “ALTER MATERIALIZED VIEW statement” [SQL Anywhere Server - SQL Reference]
- “Materialized view statuses and properties” on page 68
- “Creating indexes” on page 89
- “Initializing materialized views” on page 74

**Encrypting and decrypting 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 in order 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.
Sybase Central

To encrypt a materialized view (Sybase Central)
1. Connect to the database 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 check box.
6. Click OK.

To decrypt a materialized view (Sybase Central)
1. Connect to the database 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. Clear the Materialized View Data Is Encrypted check box.
6. Click OK.

SQL statements

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 (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.

Examples

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;
```

The following statement decrypts the EmployeeConfid44 materialized view:

```
ALTER MATERIALIZED VIEW GROUPO.EmployeeConfid44 NOT 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 “Dropping materialized views” on page 84.

See also
● “Enabling table encryption” [SQL Anywhere Server - Database Administration]
● “ALTER MATERIALIZED VIEW statement” [SQL Anywhere Server - SQL Reference]
● “DB_PROPERTY function [System]” [SQL Anywhere Server - SQL Reference]

Enabling and disabling 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 in order 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. Consequently, when you re-enable it, you'll need to 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. Connect to the database as a user with DBA authority, or as owner of the materialized view.
2. In the left pane, double-click Views.
3. Right-click the materialized view and choose Disable.

To enable a materialized view (Sybase Central)
1. Connect to the database as a user with DBA authority, or as 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 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 (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.

Examples

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 GROUPO.EmployeeConfid55 DISABLE;
```

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

```
ALTER MATERIALIZED VIEW GROUPO.EmployeeConfid55 ENABLE;
REFRESH MATERIALIZED VIEW GROUPO.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 “Dropping materialized views” on page 84.

See also

- “Changing the refresh type for a materialized view” on page 76
- “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 55
- “SYSDEPENDENCY system view” [SQL Anywhere Server - SQL Reference]
Enabling and disabling 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 it is considered too stale for use. For information about what qualifies a materialized view as a candidate in the optimization process, see “Improving performance with materialized views” on page 550.

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].

Sybase Central

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

1. 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 disable a materialized view’s use in optimization (Sybase Central)

1. 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.

SQL statements

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.

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.
Examples

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

```sql
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 GROUP0.EmployeeConfid77 DISABLE USE IN OPTIMIZATION;
```

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

```sql
ALTER MATERIALIZED VIEW GROUP0.EmployeeConfid77 ENABLE USE IN OPTIMIZATION;
```

**Caution**

When you are done with this example, you should drop the materialized view you created. Otherwise, you will not be able to make schema changes to its underlying tables Employees and Departments, when trying out other examples. You cannot alter the schema of a table that has enabled, dependent materialized view. See “Dropping materialized views” on page 84.

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.

Also, if a query explicitly references a materialized view, the view is still used to process the query, regardless of whether the view is considered stale. 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. You should adjust accordingly 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 [database]” [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 589 and “Monitor query performance” on page 242.

### Hiding 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 “Dropping materialized views” on page 84.

**See also**

- “ALTER MATERIALIZED VIEW statement” [SQL Anywhere Server - SQL Reference]

### Dropping materialized views

When a materialized view is no longer needed, you can drop it.
Materialized view deletions and view dependencies

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 55.

To drop a materialized view (Sybase Central)

1. 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 statement” [SQL Anywhere Server - SQL Reference]
- “sa_dependent_views system procedure” [SQL Anywhere Server - SQL Reference]
- “View dependencies” on page 55
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.

Choosing a set of indexes

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 “Using the Index Consultant” on page 194.

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 “Using the Index Consultant” on page 194.

Use indexes for frequently-searched columns

SQL Anywhere automatically indexes primary key and foreign key columns. Thus, 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 253.

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 of 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 in the event that 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. Consequently, the database server cannot sequentially scan the table and retrieve all of the rows in 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 accordingly.

The decision to make one of the indexes on a table clustered needs to take into account the expected query workload. Consequently, some experimentation is usually required. As a general rule, however, 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 create the clustering effect**

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].
Create 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 “Creating indexes” on page 89.

Reorder 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].

Creating 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. 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 “Monitoring and improving performance” on page 185.
Validating 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. 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 (command line)
- Run the dbvalid utility.

Examples
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;

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

dbvalid -i EmployeeNames

See “VALIDATE statement” [SQL Anywhere Server - SQL Reference], and “Validation utility (dbvalid)” [SQL Anywhere Server - Database Administration].

Rebuilding 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 617.
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. 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;
```

See also

For more information about index fragmentation and skew, and how to reduce them, see “Reducing index fragmentation and skew” on page 247.

For more information about how to detect index fragmentation and skew, see “Application Profiling Wizard” on page 189, and “sa_index_density system procedure” [SQL Anywhere Server - SQL Reference].

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

Dropping 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. 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 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 in order 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 617.

- **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”.

- **ISYSIDXCOL system table**  Just as each row in the SYSIDX system view describes one index in the database, each row in the SYSPHYSIDX system view describes one column of an index described in the SYSDIX 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].
CHAPTER 3

Ensuring data integrity

Contents

Introduction to data integrity .......................................................... 94
Using column defaults ................................................................. 97
Using table and column constraints .............................................. 103
Using domains ........................................................................... 107
Enforcing entity and referential integrity ...................................... 110
Integrity rules in the system tables ................................................. 117
Introduction to 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 right into the database is the best way to make sure your data stays in good shape.

You can enforce several types of integrity constraints. 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 805.

How data can become invalid

Here are a few examples of how the data in a database may become invalid if proper checks are not made. You can prevent each of these examples from occurring using facilities described in this chapter.

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 people 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.

Integrity constraints belong in the 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). Together, checks and rules become constraints.

Building integrity constraints into the database

Constraints that are built into the database itself are more reliable than constraints that are built into client applications or that are spelled out 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 database contents 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.

Data integrity tools

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 97.

Constraints

You can apply several types of constraints to the data in individual columns or tables. For example:

● A NOT NULL constraint prevents a column from containing a NULL entry.
● 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 thus protect against user error when entering in new values.
● 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.
● Triggers can enforce more sophisticated CHECK conditions. See “Using procedures, triggers, and batches” on page 805.
As well, 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 103.

**Entity and referential integrity**

Relationships, defined by the primary keys and foreign keys, tie together the information in relational database tables. You must build these relations directly into the database design. The following integrity rules maintain the structure of the database:

- **Entity integrity**  Keeps track of the primary keys. It guarantees that every row of a given table can be uniquely identified by a primary key that guarantees IS NOT NULL.

- **Referential integrity**  Keeps track of the foreign keys that define the relationships between tables. It guarantees that all foreign key values either match a value in the corresponding primary key or contain the NULL value if they are defined to allow NULL.

For more information about enforcing referential integrity, see “Enforcing entity and referential integrity” on page 110. For more information about designing appropriate primary and foreign key relations, see “Designing and creating your database” on page 3.

**Triggers for advanced integrity rules**

You can also use triggers to maintain data integrity. 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.

For more information about triggers, see “Using procedures, triggers, and batches” on page 805.

**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.

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;
```

Working with column defaults in Sybase Central

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

1. 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 110).

You can retrieve the most recent value inserted into an autoincrement column using the @@identity global variable. For more information, 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 645.

**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 of 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 110).

While using the global autoincrement default in other cases is possible, doing so can adversely affect database performance. For example, in cases where 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. For more information, see “@@identity global variable” [SQL Anywhere Server - SQL Reference].

See

● “Using global autoincrement” [MobiLink - Server Administration]
● “Using global autoincrement default column values” [SQL Remote]
● “CREATE TABLE statement” [SQL Anywhere Server - SQL Reference]

The NEWID default

Universally Unique Identifiers (UUIDs), also known as Globally Unique Identifiers (GUIDs), can be used to uniquely identify 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]
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.

Caution
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.

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 simply be required for data to be reasonable, 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 are checked, as well as for the rest of the row.

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:

```sql
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:

```sql
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:

```sql
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:

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

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 posint INT
    CHECK ( @col > 0 );
```

Any column defined using the posint 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.

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

---

**Working with table and column constraints in Sybase Central**

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. 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.

**Managing 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 uniquely identify each row in the table. No two rows in the table can have the same values in all the named column(s). A table can have more than one UNIQUE constraint.

**To manage unique constraints (Sybase Central)**

1. 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

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:

  ```sql
  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:

  ```sql
  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:

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

Sybase Central lets you add, alter and drop both table and column CHECK constraints. For more information, see “**Working with table and column constraints in Sybase Central**” on page 105.

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. The range of permissible values is usually restricted by a check constraint. In addition, a domain can specify a default value and may or may not allow NULLs.

You can define your own domains for a number of reasons.

- A number of common errors can be prevented if inappropriate values cannot be 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.

- Domains can make it much easier to understand applications and the structure of a database.

- Domains can prove convenient. 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 “SQL data types” [SQL Anywhere Server - SQL Reference].

Creating 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. 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. 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.
Creating domains (SQL)

You can use the CREATE DOMAIN statement to create and define domains.

**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 to be used for people's names and others are to store addresses. You might then define the following domains.

```sql
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.

```sql
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.

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

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

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

**Example 3: Built-in domains**

SQL Anywhere comes with some domains pre-defined. You can use these pre-defined domains as you would a domain that you created yourself. For example, the following monetary domain has already been created for you.

```sql
CREATE DOMAIN MONEY NUMERIC(19,4) NULL;
```
Dropping 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. 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 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 structure).

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.

The application programmer should 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.
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 43.
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:

```sql
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:

```sql
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

- “Relations between tables” [SQL Anywhere 11 - 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 hence 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 of 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 hence 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 all of the above 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 [database]” [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 93.
## 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:

<table>
<thead>
<tr>
<th>System view</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYS.SYSCON-</td>
<td>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 “SYSCON-</td>
</tr>
<tr>
<td>STRAINT</td>
<td>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].</td>
</tr>
<tr>
<td>SYS.SYSCHECK</td>
<td>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].</td>
</tr>
<tr>
<td>SYS.SYSF-</td>
<td>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].</td>
</tr>
<tr>
<td>KEY</td>
<td>Each row in the SYS.SYSIDX system view defines an index in the database. See “SYSSIDX system view” [SQL Anywhere Server - SQL Reference].</td>
</tr>
<tr>
<td>SYS.SYSTRIGGER</td>
<td>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).</td>
</tr>
<tr>
<td></td>
<td>The referential_action column holds a single character indicating whether the action is cascade (C), delete (D), set null (N), or restrict (R).</td>
</tr>
<tr>
<td></td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>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].</td>
</tr>
</tbody>
</table>
CHAPTER 4

Using transactions and isolation levels

Contents

Introduction to transactions ................................................................. 120
Isolation levels and consistency .......................................................... 124
Transaction blocking and deadlock ...................................................... 138
How locking works ............................................................................ 142
Choosing isolation levels ................................................................. 156
Isolation level tutorials .................................................................... 160
Particular concurrency issues ............................................................ 178
Summary ........................................................................................ 180
Introduction to transactions

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 accomplish 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 122.

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 your data, your alterations are not made permanent right away. Instead, they are recorded in the transaction log and are made permanent when you enter the COMMIT command.
Knowing which commands or actions signify the start or end of a transaction lets you take full advantage of transactions.

Starting transactions

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.

Completing transactions

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 [compatibility]” [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 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, ODBC’s setting will override Interactive SQL’s. 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 Sybase Adaptive Server Enterprise. For more information, see “Transact-SQL Compatibility” on page 633.

For more information, see “Transact-SQL Compatibility” on page 633.

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.

Example

A database system in a department store must allow many clerks to update customer accounts concurrently. Each clerk must be able to update the status of the accounts as they assist each customer: they cannot afford to wait until no one else is using the database.

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.

**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:

<table>
<thead>
<tr>
<th>Isolation level</th>
<th>Has these characteristics...</th>
</tr>
</thead>
<tbody>
<tr>
<td>0—read uncommitted</td>
<td>● Read permitted on row with or without write lock</td>
</tr>
<tr>
<td></td>
<td>● No read locks are applied</td>
</tr>
<tr>
<td></td>
<td>● No guarantee that concurrent transaction will not modify row or roll back changes to row</td>
</tr>
<tr>
<td></td>
<td>● Corresponds to table hints NOLOCK and READUNCOMMITTED</td>
</tr>
<tr>
<td></td>
<td>● Allow dirty reads, non-repeatable reads, and phantom rows</td>
</tr>
<tr>
<td>1—read committed</td>
<td>● Read only permitted on row with no write lock</td>
</tr>
<tr>
<td></td>
<td>● Read lock acquired and held for read on current row only, but released when cursor moves off the row</td>
</tr>
<tr>
<td></td>
<td>● No guarantee that data will not change during transaction</td>
</tr>
<tr>
<td></td>
<td>● Corresponds to table hint READCOMMITTED</td>
</tr>
<tr>
<td></td>
<td>● Prevent dirty reads</td>
</tr>
<tr>
<td></td>
<td>● Allow non-repeatable reads and phantom rows</td>
</tr>
<tr>
<td>2—repeatable read</td>
<td>● Read only permitted on row with no write lock</td>
</tr>
<tr>
<td></td>
<td>● Read lock acquired as each row in the result set is read, and held until transaction ends</td>
</tr>
<tr>
<td></td>
<td>● Corresponds to table hint REPEATABLEREAD</td>
</tr>
<tr>
<td></td>
<td>● Prevent dirty reads and non-repeatable reads</td>
</tr>
<tr>
<td></td>
<td>● Allow phantom rows</td>
</tr>
<tr>
<td>3—serializable</td>
<td>● Read only permitted on rows in result without write lock</td>
</tr>
<tr>
<td></td>
<td>● Read locks acquired when cursor is opened and held until transaction ends</td>
</tr>
<tr>
<td></td>
<td>● Corresponds to table hints HOLDLOCK and SERIALIZABLE</td>
</tr>
<tr>
<td></td>
<td>● Prevent dirty reads, non-repeatable reads, and phantom rows</td>
</tr>
</tbody>
</table>
This isolation level... | Has these characteristics...
--- | ---
snapshot<sup>1</sup> |  ● 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
statement-snapshot<sup>1</sup> |  ● 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<sup>1</sup> |  ● 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

<sup>1</sup> 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 128.

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 performance is the same as isolation level 0 with respect to contention. 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 will execute completely or not at all, and that no updates will be lost.

The isolation is between transactions only: multiple cursors within the same transaction can interfere with each other.
Snapshot isolation

When users are reading and writing the same data simultaneously, blocks, and even deadlocks, can occur. 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 thus 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 useful in many cases, 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

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 [compatibility]” [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.
Remember that 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 will reduce 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.

Note that for the purposes of 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:

```sql
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:

```sql
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:

```sql
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 11 Demo;UID=DBA;PWD=sql;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 11 Demo;UID=DBA;PWD=sql;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;
   ```

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Description</th>
<th>Size</th>
<th>Color</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>300</td>
<td>Tee Shirt</td>
<td>Tank Top</td>
<td>Small</td>
<td>White</td>
<td>28</td>
</tr>
<tr>
<td>301</td>
<td>Tee Shirt</td>
<td>V-neck</td>
<td>Medium</td>
<td>Orange</td>
<td>54</td>
</tr>
<tr>
<td>302</td>
<td>Tee Shirt</td>
<td>Crew Neck</td>
<td>One size fits all</td>
<td>Black</td>
<td>75</td>
</tr>
<tr>
<td>400</td>
<td>Baseball Cap</td>
<td>Cotton Cap</td>
<td>One size fits all</td>
<td>Black</td>
<td>112</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

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;
### 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 as soon as 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

---

### SQL Examples

```sql
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 162
- “Using snapshot isolation to avoid non-repeatable reads” on page 168
- “Using snapshot isolation to avoid phantom rows” on page 172

---

### Table

| ID  | Name       | Description          | 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:
in the ISO SQL/2003 standard and are important because behavior at lower isolation levels is defined in terms of 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 160.

- **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 164.

- **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 169.

### 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.

<table>
<thead>
<tr>
<th>Isolation level</th>
<th>Dirty reads</th>
<th>Non-repeatable reads</th>
<th>Phantom rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-read uncommitted</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>readonly-statement-snapshot</td>
<td>X(^1)</td>
<td>X(^2)</td>
<td>X(^3)</td>
</tr>
<tr>
<td>1-read committed</td>
<td>X</td>
<td>X</td>
<td>X(^3)</td>
</tr>
<tr>
<td>statement-snapshot</td>
<td>X(^2)</td>
<td>X(^3)</td>
<td>X(^3)</td>
</tr>
<tr>
<td>2-repeatable read</td>
<td>X</td>
<td></td>
<td>X(^3)</td>
</tr>
<tr>
<td>3-serializable snapshot</td>
<td>X</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.
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 135.

**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 805.

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].
Setting 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 the WITH table-hint section in “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 128.

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 136.

## 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

<table>
<thead>
<tr>
<th>ValuePtr</th>
<th>Isolation level</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_TXN_READ_UNCOMMITTED</td>
<td>0</td>
</tr>
<tr>
<td>SQL_TXN_READ_COMMITTED</td>
<td>1</td>
</tr>
<tr>
<td>SQL_TXN_REPEATABLE_READ</td>
<td>2</td>
</tr>
<tr>
<td>SQL_TXN_SERIALIZABLE</td>
<td>3</td>
</tr>
<tr>
<td>SA_SQL_TXN_SNAPSHOT</td>
<td>snapshot</td>
</tr>
<tr>
<td>SA_SQL_TXN_STATEMENT_SNAPSHOT</td>
<td>statement-snapshot</td>
</tr>
<tr>
<td>SA_SQL_TXN_READONLY_STATEMENT_SNAPSHOT</td>
<td>readonly-statement-snapshot</td>
</tr>
</tbody>
</table>

### 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.

<table>
<thead>
<tr>
<th>String</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL_TXN_ISOLATION</td>
<td>108</td>
</tr>
<tr>
<td>SQL_TXN_READ_UNCOMMITTED</td>
<td>1</td>
</tr>
<tr>
<td>SQL_TXN_READ_COMMITTED</td>
<td>2</td>
</tr>
</tbody>
</table>
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:

```c
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:

```c
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 169.

**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 NON_SNAPSHOT_TRANSACTION. For example:

```sql
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:

```sql
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.

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 138.

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 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 [database]” [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. Clearly, 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.

- **All active database threads are blocked**  When a transaction becomes blocked, its database thread is not relinquished. If the server is configured with three threads 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 threads.

To eliminate a transactional deadlock, SQL Anywhere selects a connection from those involved in the deadlock, rolls back the changes for the transaction active on that connection and returns an error. SQL Anywhere selects the connection to roll back 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.

To eliminate a thread deadlock, SQL Anywhere selects the last thread to block, rolls back the changes for the transaction active on that connection and returns an error.

The number of database threads that the server uses depends on the individual database's setting.

For information about setting the number of database threads, see “Controlling threading behavior” [SQL Anywhere Server - Database Administration].

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. The database server provides more detailed information about the deadlock when used with the log_deadlocks option and when the RememberLastStatement feature is enabled.

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 265.

**To take action when a deadlock occurs**

1. Create a table to store the data returned from the sa_report_deadlocks system procedure:

   ```sql
   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:

```sql
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:

```sql
SET OPTION PUBLIC.log_deadlocks = 'On';
```

4. Enable logging of the most-recently executed statement:

```sql
CALL sa_server_option( 'RememberLastStatement', 'YES' );
```

See also
- “log_deadlocks option [database]” [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.

To use Sybase Central deadlock reporting

1. Select the database in the left pane of Sybase Central, and then choose File » Options.

The Database Options window appears.
2. Turn on the log_deadlocks option.

   The database server records information about deadlocks in an internal buffer.
   a. Select log_deadlocks in the **Options** list.
   b. Type **On** in the value field.
   c. Click **Set Temporary Now**.
   d. Click **Close**.

   For more information, see “log_deadlocks option [database]” [SQL Anywhere Server - Database Administration].

3. Click the **Deadlocks** tab in the right pane.

   A deadlock diagram appears if there are any 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

When the database server processes a transaction, it can lock one or more rows of a table. 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 which 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 noted in “Early release of read locks” on page 155.

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.

- **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 of 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

In order 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 Table 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 it is possible that the rows visible in Sybase Central, or returned by the sa_locks system procedure, 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 Table 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 uniquely identify the row in a table the lock refers to. In order 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:

```sql
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 prior to 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, it can acquire a read lock. Whether or not a read lock is acquired depends on the transaction's isolation level. Once a row has been 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 consequently 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 126.

## 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 prior to 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. Consequently, any attempt by the snapshot transaction to update that row will fail with a snapshot update conflict error.

**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.

<table>
<thead>
<tr>
<th></th>
<th>Shared</th>
<th>Intent</th>
<th>Exclusive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shared</td>
<td></td>
<td>conflict</td>
<td>conflict</td>
</tr>
<tr>
<td>Intent</td>
<td>conflict</td>
<td></td>
<td>conflict</td>
</tr>
<tr>
<td>Exclusive</td>
<td>conflict</td>
<td>conflict</td>
<td>conflict</td>
</tr>
</tbody>
</table>

**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 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. Phantoms are rows that appear, or disappear, with respect to an operation as a result of the dynamic nature of a database. 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. In the case of a sequential scan, the scan position is based on the row identifier of the current row. In the case of 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. For the purposes of this discussion with respect to locking, insertions means not only INSERT statements, but also UPDATE
statements that change the value of an indexed attribute. In such cases, the UPDATE can be considered a DELETE of the index entry followed immediately by an INSERT.

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, since it is the only isolation level that guarantees consistency with respect to 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. Consequently, 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 of 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. Hence, 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. Hence, 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.

<table>
<thead>
<tr>
<th></th>
<th>read (R)</th>
<th>intent (R)</th>
<th>write (R)</th>
<th>shared (T)</th>
<th>intent (T)</th>
<th>exclusive (T)</th>
<th>phantom (P)</th>
<th>insert (P)</th>
</tr>
</thead>
<tbody>
<tr>
<td>read (R)</td>
<td></td>
<td>conflict</td>
<td></td>
<td></td>
<td>conflict</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>intent (R)</td>
<td></td>
<td>conflict</td>
<td>conflict</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>write (R)</td>
<td>conflict</td>
<td>conflict</td>
<td>conflict</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>shared (T)</td>
<td></td>
<td></td>
<td></td>
<td>conflict</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>intent (T)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>conflict</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>exclusive (T)</td>
<td></td>
<td>conflict</td>
<td>conflict</td>
<td>conflict</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>phantom (P)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>conflict</td>
<td></td>
</tr>
<tr>
<td>insert (P)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>conflict</td>
<td>conflict</td>
</tr>
</tbody>
</table>
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 the responsibility of the programmer or 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 is free to read any row, whether or not another transaction has acquired a write lock on it. 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 as soon as 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 or not 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.

The fact that a read lock must be acquired on each row whether or not it satisfies any predicates in the SELECT statement has two important implications.

- The database server may need to place many more locks than would be necessary at isolation level 2. The number of phantom locks acquired will be 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 all transactions must see the same snapshot of the data, which can lead to increased locking overhead and reduced concurrency.
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 COMMIIts. 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. Thus, you should 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 the purposes of 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 124.
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 in some cases 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 152. 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 of 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 deleted. As rows are scanned, they are locked. The default locking behavior is described in “Isolation levels and consistency” on page 124.

   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 in some cases 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 of 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. In order 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.

**Early release of read locks**

At isolation level 3, a transaction acquires a read lock on every row it reads. Ordinarily, a transaction never releases a lock before the end of the transaction. In fact, it is essential that a transaction does not release locks early if the schedule is to be serializable.

SQL Anywhere always retains write locks until a transaction completes. This prevents another transaction from modifying that row and making it impossible to roll back the first transaction.

Read locks are released only in one circumstance: Under isolation level 1, transactions acquire a read lock on a row only when it becomes the current row of a cursor. Under isolation level 1, however, when that row is no longer current, the lock is released. This behavior is acceptable because the database server does not need to guarantee repeatable reads at isolation level 1.

For more information about isolation levels, see “Choosing isolation levels” on page 156.
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 in cases where 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 somewhat 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 126.
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 131 are typical of the types of problems that appear when the schedule is not serializable. In each case, the inconsistency appeared because the statements were interleaved in such a way as to produce a result that 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 particularly 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 particularly 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 thus 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 hence can improve the performance of your database significantly.

**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, thus shortening 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 or not 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

You should avoid running transactions at isolation level 3 whenever practical. They tend to place a large number of locks and hence impact the efficient execution of other concurrent transactions.

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.

In particular, you may find that adding an index may greatly help speed up transactions, particularly when at least one of them must execute at isolation level 3. 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 142.

For more information about performance and how SQL Anywhere plans its access of information to execute your commands, see “Monitoring and improving performance” on page 185.
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.

In this example, you will play the role of two people, 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:
   - In the ODBC Data Source Name field, choose SQL Anywhere 11 Demo.
   - Click the Advanced tab and type Sales Manager in the ConnectionName field.
   - Click OK.
3. Start a second instance of Interactive SQL.
4. In the Connect window, connect to the SQL Anywhere sample database as the Accountant:
   - In the ODBC Data Source Name field, choose SQL Anywhere 11 Demo.
   - Click the Advanced tab and type Accountant in the ConnectionName field.
   - Click OK.
5. As the Sales Manager, raise the price of all tee shirts by $0.95:
   - In the Sales Manager window, execute the following commands:

```sql
UPDATE Products
SET UnitPrice = UnitPrice + 95
WHERE Name = 'Tee Shirt';
SELECT ID, Name, UnitPrice
FROM Products;
```
The result is:

<table>
<thead>
<tr>
<th>ID</th>
<th>name</th>
<th>UnitPrice</th>
</tr>
</thead>
<tbody>
<tr>
<td>300</td>
<td>Tee Shirt</td>
<td>104.00</td>
</tr>
<tr>
<td>301</td>
<td>Tee Shirt</td>
<td>109.00</td>
</tr>
<tr>
<td>302</td>
<td>Tee Shirt</td>
<td>109.00</td>
</tr>
<tr>
<td>400</td>
<td>Baseball Cap</td>
<td>9.00</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Inventory</th>
</tr>
</thead>
<tbody>
<tr>
<td>21453.00</td>
</tr>
</tbody>
</table>

Unfortunately, this calculation is not accurate. The Sales Manager accidentally raised the price of the visor $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 124.

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';
COMMIT;
```

<table>
<thead>
<tr>
<th>ID</th>
<th>name</th>
<th>UnitPrice</th>
</tr>
</thead>
<tbody>
<tr>
<td>300</td>
<td>Tee Shirt</td>
<td>9.95</td>
</tr>
<tr>
<td>301</td>
<td>Tee Shirt</td>
<td>14.95</td>
</tr>
</tbody>
</table>
Using transactions and isolation levels

<table>
<thead>
<tr>
<th>ID</th>
<th>name</th>
<th>UnitPrice</th>
</tr>
</thead>
<tbody>
<tr>
<td>302</td>
<td>Tee Shirt</td>
<td>14.95</td>
</tr>
<tr>
<td>400</td>
<td>Baseball Cap</td>
<td>9.00</td>
</tr>
</tbody>
</table>

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.

```sql
SELECT SUM( Quantity * UnitPrice )
AS Inventory
FROM Products;
```

```
<table>
<thead>
<tr>
<th>Inventory</th>
</tr>
</thead>
<tbody>
<tr>
<td>6687.15</td>
</tr>
</tbody>
</table>
```

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.

```sql
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.

### 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:
   - In the ODBC Data Source Name field, choose SQL Anywhere 11 Demo.
   - Click the Advanced tab and type Sales Manager in the ConnectionName field.
   - Click OK to connect.

3. Execute the following statement to enable snapshot isolation for the database:

   ```sql
   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:
   - In the ODBC Data Source Name field, choose SQL Anywhere 11 Demo.
   - Click the Advanced tab and type Accountant in the ConnectionName field.
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:

<table>
<thead>
<tr>
<th>Inventory</th>
</tr>
</thead>
<tbody>
<tr>
<td>6687.15</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Inventory</th>
</tr>
</thead>
<tbody>
<tr>
<td>6538.00</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Inventory</th>
</tr>
</thead>
<tbody>
<tr>
<td>6687.15</td>
</tr>
</tbody>
</table>
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 128.

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;
```

**Tutorial: Non-repeatable reads**

The example in “Tutorial: Dirty reads” on page 160 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 will play the role of the same two people, 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.

1. Start Interactive SQL.
2. In the Connect window, connect to the SQL Anywhere sample database as the Sales Manager:
   - In the ODBC Data Source Name field, choose SQL Anywhere 11 Demo.
   - Click the Advanced tab and type Sales Manager in the ConnectionName field.
   - Click OK.
3. Start a second instance of Interactive SQL.
4. In the Connect window, connect to the SQL Anywhere sample database as the Accountant:
   - In the ODBC Data Source Name field, choose SQL Anywhere 11 Demo.
   - Click the Advanced tab and type Accountant in the ConnectionName field.
   - Click OK.
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;

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>UnitPrice</th>
</tr>
</thead>
<tbody>
<tr>
<td>300</td>
<td>Tee Shirt</td>
<td>9.00</td>
</tr>
<tr>
<td>301</td>
<td>Tee Shirt</td>
<td>14.00</td>
</tr>
<tr>
<td>302</td>
<td>Tee Shirt</td>
<td>14.00</td>
</tr>
<tr>
<td>400</td>
<td>Baseball Cap</td>
<td>9.00</td>
</tr>
<tr>
<td>401</td>
<td>Baseball Cap</td>
<td>10.00</td>
</tr>
<tr>
<td>500</td>
<td>Visor</td>
<td>7.00</td>
</tr>
<tr>
<td>501</td>
<td>Visor</td>
<td>7.00</td>
</tr>
</tbody>
</table>

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';
```

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>UnitPrice</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>Visor</td>
<td>7.00</td>
</tr>
<tr>
<td>501</td>
<td>Visor</td>
<td>5.95</td>
</tr>
</tbody>
</table>

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;
```

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>UnitPrice</th>
</tr>
</thead>
<tbody>
<tr>
<td>300</td>
<td>Tee Shirt</td>
<td>9.00</td>
</tr>
<tr>
<td>301</td>
<td>Tee Shirt</td>
<td>14.00</td>
</tr>
<tr>
<td>302</td>
<td>Tee Shirt</td>
<td>14.00</td>
</tr>
</tbody>
</table>
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;

Observe that as soon as the database server executes this statement, the Sales Manager's transaction completes.
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 doesn't bother to 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 particularly 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 138.

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:
   - In the ODBC Data Source Name field, choose SQL Anywhere 11 Demo.
   - Click the Advanced tab and type Sales Manager in the ConnectionName field.
   - Click OK.
3. Start a second instance of Interactive SQL.
4. In the Connect window, connect to the SQL Anywhere sample database as the Accountant:
   - In the ODBC Data Source Name field, choose SQL Anywhere 11 Demo.
   - Click the Advanced tab and type Accountant in the ConnectionName field.
   - Click OK.
5. Execute the following statements to enable snapshot isolation for the database and specify that the
   snapshot isolation level is used:
   ```sql
   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:
   ```sql
   SELECT ID, Name, UnitPrice
   FROM Products
   ORDER BY ID;
   ```

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>UnitPrice</th>
</tr>
</thead>
<tbody>
<tr>
<td>300</td>
<td>Tee Shirt</td>
<td>9.00</td>
</tr>
<tr>
<td>301</td>
<td>Tee Shirt</td>
<td>14.00</td>
</tr>
<tr>
<td>302</td>
<td>Tee Shirt</td>
<td>14.00</td>
</tr>
<tr>
<td>400</td>
<td>Baseball Cap</td>
<td>9.00</td>
</tr>
<tr>
<td>401</td>
<td>Baseball Cap</td>
<td>10.00</td>
</tr>
<tr>
<td>500</td>
<td>Visor</td>
<td>7.00</td>
</tr>
</tbody>
</table>
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

The following tutorial continues the same scenario. In this case, the Accountant views the Departments table while the Sales Manager creates a new department. You will observe the appearance of a phantom row.

If you have not done so, do steps 1 through 4 of the previous tutorial, “Tutorial: Non-repeatable reads” on page 164, so that you have two instances of Interactive SQL.

1. Set the isolation level to 2 in the Sales Manager window by executing the following command.

```
SET TEMPORARY OPTION isolation_level = 2;
```

2. Set the isolation level to 2 for the Accountant window by executing the following command.

```
SET TEMPORARY OPTION isolation_level = 2;
```

3. In the Accountant window, enter the following command to list all the departments.

```
SELECT * FROM Departments
ORDER BY DepartmentID;
```
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;
```

<table>
<thead>
<tr>
<th>DepartmentID</th>
<th>DepartmentName</th>
<th>DepartmentHeadID</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>R &amp; D</td>
<td>501</td>
</tr>
<tr>
<td>200</td>
<td>Sales</td>
<td>902</td>
</tr>
<tr>
<td>300</td>
<td>Finance</td>
<td>1293</td>
</tr>
<tr>
<td>400</td>
<td>Marketing</td>
<td>1576</td>
</tr>
<tr>
<td>500</td>
<td>Shipping</td>
<td>703</td>
</tr>
<tr>
<td>600</td>
<td>Foreign Sales</td>
<td>129</td>
</tr>
</tbody>
</table>

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;

<table>
<thead>
<tr>
<th>DepartmentID</th>
<th>DepartmentName</th>
<th>DepartmentHeadID</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>R &amp; D</td>
<td>501</td>
</tr>
<tr>
<td>200</td>
<td>Sales</td>
<td>902</td>
</tr>
<tr>
<td>300</td>
<td>Finance</td>
<td>1293</td>
</tr>
<tr>
<td>400</td>
<td>Marketing</td>
<td>1576</td>
</tr>
<tr>
<td>500</td>
<td>Shipping</td>
<td>703</td>
</tr>
<tr>
<td>600</td>
<td>Foreign Sales</td>
<td>129</td>
</tr>
</tbody>
</table>

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 and hence there is nothing to prevent the Sales Manager from inserting a new row.

6. 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;
```

7. 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 **Interrupt The SQL Statement** (or choose **SQL > Stop**) to interrupt this entry.

8. 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 142.

**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 169 which describe how to start two instances of Interactive SQL.

1. Enable snapshot isolation for the Accountant by executing the following command.

   ```sql
   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.

   ```sql
   SELECT * FROM Departments
   ORDER BY DepartmentID;
   ```

<table>
<thead>
<tr>
<th>DepartmentID</th>
<th>DepartmentName</th>
<th>DepartmentHeadID</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>R &amp; D</td>
<td>501</td>
</tr>
<tr>
<td>200</td>
<td>Sales</td>
<td>902</td>
</tr>
<tr>
<td>300</td>
<td>Finance</td>
<td>1293</td>
</tr>
<tr>
<td>400</td>
<td>Marketing</td>
<td>1576</td>
</tr>
<tr>
<td>500</td>
<td>Shipping</td>
<td>703</td>
</tr>
</tbody>
</table>

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.

   ```sql
   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.

```sql
SELECT * FROM Departments
ORDER BY DepartmentID;
```

<table>
<thead>
<tr>
<th>DepartmentID</th>
<th>DepartmentName</th>
<th>DepartmentHeadID</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>R &amp; D</td>
<td>501</td>
</tr>
<tr>
<td>200</td>
<td>Sales</td>
<td>902</td>
</tr>
<tr>
<td>300</td>
<td>Finance</td>
<td>1293</td>
</tr>
<tr>
<td>400</td>
<td>Marketing</td>
<td>1576</td>
</tr>
<tr>
<td>500</td>
<td>Shipping</td>
<td>703</td>
</tr>
<tr>
<td>600</td>
<td>Foreign Sales</td>
<td>129</td>
</tr>
</tbody>
</table>

4. The Accountant can execute his query again, and does not see the new row because the transaction has not ended.

```sql
SELECT *
FROM Departments
ORDER BY DepartmentID;
```

<table>
<thead>
<tr>
<th>DepartmentID</th>
<th>DepartmentName</th>
<th>DepartmentHeadID</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>R &amp; D</td>
<td>501</td>
</tr>
<tr>
<td>200</td>
<td>Sales</td>
<td>902</td>
</tr>
<tr>
<td>300</td>
<td>Finance</td>
<td>1293</td>
</tr>
<tr>
<td>400</td>
<td>Marketing</td>
<td>1576</td>
</tr>
<tr>
<td>500</td>
<td>Shipping</td>
<td>703</td>
</tr>
</tbody>
</table>

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.

```sql
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.

```sql
COMMIT;
SELECT * FROM Departments
ORDER BY DepartmentID;
```
Now the Accountant sees the Foreign Sales department, but not the Major Account Sales department.

<table>
<thead>
<tr>
<th>DepartmentID</th>
<th>DepartmentName</th>
<th>DepartmentHeadID</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>R &amp; D</td>
<td>501</td>
</tr>
<tr>
<td>200</td>
<td>Sales</td>
<td>902</td>
</tr>
<tr>
<td>300</td>
<td>Finance</td>
<td>1293</td>
</tr>
<tr>
<td>400</td>
<td>Marketing</td>
<td>1576</td>
</tr>
<tr>
<td>500</td>
<td>Shipping</td>
<td>703</td>
</tr>
<tr>
<td>600</td>
<td>Foreign Sales</td>
<td>129</td>
</tr>
</tbody>
</table>

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:

   ```sql
   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.

   ```sql
   DELETE FROM Departments
   WHERE DepartmentID = 600;
   COMMIT;
   ```

**Tutorial: Practical locking implications**

This tutorial continues the same scenario. 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 placed on an indexed scan position to prevent phantom rows. When a transaction at isolation level 3 selects rows which match a given criterion, the database server places anti-insert locks to stop other transactions from inserting rows which would also match. The number of locks placed on your behalf depends both on the search criterion and on the design of your database.

If you have not done so, follow steps 1 through 4 of the “**Tutorial: Phantom rows**” on page 169, which describe how to start two instances of Interactive SQL.

1. 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;

2. 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;
```

<table>
<thead>
<tr>
<th>EmployeeID</th>
<th>GivenName</th>
<th>Surname</th>
<th>April sales</th>
</tr>
</thead>
<tbody>
<tr>
<td>129</td>
<td>Philip</td>
<td>Chin</td>
<td>2160.00</td>
</tr>
<tr>
<td>195</td>
<td>Marc</td>
<td>Dill</td>
<td>2568.00</td>
</tr>
<tr>
<td>299</td>
<td>Rollin</td>
<td>Overbey</td>
<td>5760.00</td>
</tr>
<tr>
<td>467</td>
<td>James</td>
<td>Klobucher</td>
<td>3228.00</td>
</tr>
</tbody>
</table>

3. 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;
```

4. 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.
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!

5. At isolation level 3, the database server places anti-insert locks to ensure that no other transactions can add a row which matches the criterion of a search or select.

In the Sales Manager's window, execute the following statements to remove the new order.

```sql
DELETE
FROM SalesOrderItems
WHERE ID = 2653;
DELETE
FROM SalesOrders
WHERE ID = 2653;
COMMIT;
```

6. In the Accountant's window, execute the following two statements.

```sql
ROLLBACK;
SET TEMPORARY OPTION isolation_level = 3;
```

7. In the Accountant's window, execute same query as before.

```sql
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 his transaction.

8. Return to the Sales Manager's window. Again attempt to enter Philip Chin's missing order.

```sql
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 **Interrupt The SQL Statement** (or choose **SQL » Stop**) to interrupt this entry.

9. The Sales Manager cannot enter the order in April, but you might think that she could still enter it in May.

   Change the date of the command to May 05 and try again.

   ```sql
   INSERT INTO SalesOrders
   VALUES ( 2653, 174, '2001-05-05', 'r1', 'Central', 129);
   ```

   The Sales Manager's window stops responding again. On the toolbar, click **Interrupt The SQL Statement** (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 a large number of other 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.

10. The Sales Manager wants to add a second item to order 2651. Use the following command.

    ```sql
    INSERT INTO SalesOrderItems
    VALUES ( 2651, 2, 302, 4, '2001-05-22' );
    ```

    The Sales Manager's window stops responding. On the toolbar, click **Interrupt The SQL Statement** (or choose **SQL » Stop**) to interrupt this entry.

11. Conclude this tutorial by undoing any changes to avoid changing the SQL Anywhere sample database. Enter the following command in the Sales Manager's window.

    ```sql
    ROLLBACK;
    ```

    Enter the same command in the Accountant's window.

    ```sql
    ROLLBACK;
    ```

    You may now close both windows.
Particular concurrency issues

This section discusses the following particular concurrency issues:

● “Primary key generation” on page 178
● “Data definition statements and concurrency” on page 179

Primary key generation

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.

There are many methods for generating such numbers.

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 people 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 check that each person still has a sufficient supply of numbers.

● Create a table with the columns table name and last key value.

  One row in this table would contain the last invoice number used. Each time someone adds an invoice, establish a new connection, increment the number in the table, and commit the change immediately. The incremented number can be used for the new invoice. Other users will be able to grab invoice numbers because you updated the row with a separate transaction that only lasted an instant.

● Use a column with a default value of NEWID in conjunction with the UNIQUEIDENTIFIER binary data type to generate a universally unique identifier.

  UUID and GUID values can be used to uniquely identify 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.

  For more information about generating unique identifiers, see “The NEWID default” on page 101.

● Use a column with a default value of AUTOINCREMENT. For example:

```sql
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.

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 “Replication of data definition statements” [SQL Remote].
Transactions and locking are perhaps 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. You may end each by either rolling back any changes you have made or by committing these changes and so making them permanent.

Transactions are essential to data recovery in the event of system failure. 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 to be executed concurrently, the database server must schedule the execution of the individual statements. Concurrent transactions have the potential to introduce new, inconsistent results that could not arise were these same transactions executed sequentially.

Many types of inconsistencies are possible, but four typical types are particularly important because they are mentioned in the ISO SQL/2003 standard and the isolation levels are defined in terms of them.

- **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.
Part II. Monitoring and Improving Database Performance

This part describes how to perform database and application profiling activities, monitor and improve performance, and troubleshoot specific performance problems.
CHAPTER 5

Monitoring and improving performance

Contents

Introduction to monitoring and improving performance ............................................. 186
Application profiling ................................................................................................... 188
Advanced application profiling using diagnostic tracing ............................................ 199
Other diagnostic tools and techniques ...................................................................... 217
Monitoring database performance ............................................................................. 223
Performance improvement tips .................................................................................. 239
Introduction to monitoring and improving performance

Improving database performance involves evaluating the current performance of your database, and considering all your options before changing anything. By re-evaluating your database schema and application design using the performance features and analysis tools provided in SQL Anywhere, you can diagnose and correct performance problems and keep your database performing at its optimum level.

Ultimately, how well your database performs depends heavily on its design. One of the most basic of ways of improving performance is with good schema design. The database schema is the framework of your database, and includes definitions of such things as tables, views, triggers, and the relationships between them. Re-evaluate your database schema and make note of the areas where small changes can offer impressive gains. For more information about designing your database schema, see “Designing and creating your database” on page 3.

Once your database is in production, SQL Anywhere provides several advanced tools for detecting and diagnosing performance issues that arise. The majority of these tools rely on the diagnostic tracing infrastructure—a system of tables, files, and other components that capture and store diagnostic data. You can then use this data to perform various diagnostic and monitoring tasks such as application profiling.

There are several approaches to generating and analyzing performance data in SQL Anywhere:

- **Application profiling using 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 188.

- **Advanced application profiling using the Database Tracing Wizard**  This wizard, available from Application Profiling mode in Sybase Central, provides the ability to customize the types of performance data gathered. This allows you to focus on specific users or activities that require attention. See “Advanced application profiling using diagnostic tracing” on page 199.

- **Request trace analysis**  This feature allows you to narrow diagnostic data gathering to requests (statements) issued by specific users or connections. See “Performing request trace analysis” on page 214.

- **Index Consultant**  This feature analyzes the indexes in the database and provides improvement recommendations. This tool can be accessed either through Application Profiling mode, or as a standalone tool. See “Index Consultant” on page 194.

- **Procedure profiling**  This feature allows you to see 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. A more simple implementation is also available using system procedures. See “Procedure profiling in Application Profiling mode” on page 190.

For more information about how to perform procedure profiling tasks using commands, see “Procedure profiling using system procedures” on page 219.

- **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 589.

**Note**
In the documentation, the terms *application profiling* and *diagnostic tracing* are often used interchangeably because they are essentially the same. Diagnostic tracing is a more advanced level of application profiling.
Application profiling

Application profiling allows you to analyze tracing information to understand how applications interact with the database. With the data generated during a tracing session, you can also identify and eliminate performance problems. You can choose between two approaches for generating, and making use of, profiling information: an automated approach, using the Application Profiling Wizard, or an advanced approach using the various tools and features found in Application Profiling mode of Sybase Central.

- **Automated application profiling approach** The automated approach involves using the Application Profiling Wizard in Sybase Central to create and analyze a tracing session to look for common performance problems, and provide specific recommendations on how to improve database performance. The wizard offers you some control over the types of activities it profiles, and recommendations are automatically provided at the end of the tracing session. The Index Consultant has also been integrated into the Application Profiling Wizard and uses the data to recommend indexes, if necessary.

  The automated approach is ideal for environments where there are few connections to the database being profiled, or where sophisticated profiling is not required.

- **Advanced application profiling approach using diagnostic tracing** The advanced approach involves reviewing the data generated during a tracing session that you create either manually from the command line, or using the Database Tracing Wizard in the Sybase Central Design mode. You can configure where the tracing data is stored. You also have full control over the activities profiled, allowing you to target specific problems. For example, you can target specific statements executed by the database server, query plans used, deadlocks, connections that block each other, performance statistics, and so on.

  The advanced approach is recommended for environments where the database being profiled 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 199.

Application Profiling mode in Sybase Central allows you to explore the data gathered using either approach. For example, you can open a tracing session generated by the Database Tracing Wizard to look at a statement and then see a list of all connections that blocked the statement. Or, you can browse the tracing session data gathered by the Application Profiling Wizard to identify the most expensive statements and procedures, and gain insight into any other factors that are impacting the performance of your application.

**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.
Application Profiling Wizard

The Application Profiling Wizard in Sybase Central provides you with a simple, automated method of performing a diagnostic tracing session for the purpose of profiling applications. The wizard gathers valuable data on how your applications are interacting with the database, and then it provides you access to the data it gathered, as well as indexing recommendations, if any. See “Application Profiling Wizard” on page 189.

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 199.

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 “Creating a diagnostic tracing session” on page 210.

To use the Application Profiling Wizard (Sybase Central)

1. Connect to the database as a user with DBA authority or as a user with PROFILE authority.
2. From the Mode menu, choose 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 instructions.

   The wizard creates a local database to hold diagnostic tracing information, starts the network server, starts a tracing session, and then prompts you to run the application you would like to profile. Do not click Finish; this ends profiling, and closes the wizard.

4. Run the application you want to profile. Or, if you are profiling general database usage, allow some time for gathering data. When you have finished, 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 195.

For more information about the procedure profiling information gathered during the tracing session, see “How to read procedure profiling results” on page 193.

Tip

You can disable the Application Profiling Wizard from starting automatically when switching to Application Profiling mode. To do this, 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 11 » Preferences, select the Utilities tab, and then select the appropriate options.
Enabling 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.

To enable procedure profiling (Sybase Central)

1. Connect to the database as a user with DBA authority or as a user with PROFILE authority.
2. In the left pane, select the database.
3. From the Mode menu, choose Application Profiling.
4. Follow the Application Profiling Wizard instructions.
   
   If the Application Profiling Wizard does not appear, choose Application Profiling » Open Application Profiling Wizard.

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.
Resetting 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. Connect to the database as a user with DBA authority or as a user with PROFILE authority.
2. In the left pane, select the database.
3. From the Mode menu, choose 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.

Disabling procedure profiling

Once 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. Connect to the database as a user with DBA authority or as a user with PROFILE authority.
2. In the left pane, select the database.
3. From the **Mode** menu, choose **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. Connect to the database as a user with DBA authority or as a user with PROFILE authority.
2. In the left pane, select the database.
3. From the **Mode** menu, choose **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**

- “Enabling procedure profiling” on page 190
- “Resetting procedure profiling” on page 191
- “Analyzing procedure profiling results” on page 192

**Analyzing 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. Connect to the database as a user with DBA authority and enable procedure profiling. See “Enabling procedure profiling” on page 190.
2. In the left pane, double-click one of the following: **Triggers, System Triggers, Procedures & Functions**, or **Events**.
3. 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.

4. 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 of the objects, grouped by type, that have been executed within the database since you started procedure profiling. The information displayed includes:

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>The name of the object.</td>
</tr>
<tr>
<td>Owner</td>
<td>The owner of the object.</td>
</tr>
<tr>
<td>Table or Table Name</td>
<td>The table a trigger belongs to (this column only appears on the database Profile tab).</td>
</tr>
<tr>
<td>Event</td>
<td>The type of object, for example, a procedure.</td>
</tr>
<tr>
<td>Type</td>
<td>The type of trigger for system triggers. This can be Update or Delete.</td>
</tr>
<tr>
<td># Execs.</td>
<td>The number times each object has been called.</td>
</tr>
<tr>
<td># msec.</td>
<td>The total execution time for each object.</td>
</tr>
</tbody>
</table>

These columns, and their content, may vary depending on the type of object.

When you double-click a specific object, such as a procedure, detailed information specific to that object appears in the **Profiling Results** tab. The information displayed includes:

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execs</td>
<td>The number of times the line of code in the object was executed.</td>
</tr>
<tr>
<td>Milliseconds</td>
<td>The total amount of time that a line took to execute.</td>
</tr>
<tr>
<td>%</td>
<td>The percent of total time that a line took to execute.</td>
</tr>
<tr>
<td>Line</td>
<td>The line number within the object.</td>
</tr>
<tr>
<td>Source</td>
<td>The code that was executed.</td>
</tr>
</tbody>
</table>

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.
The selection of a proper set of indexes can make a big difference to the performance of your database. To help you in the task of selecting such a set of indexes, SQL Anywhere includes an Index Consultant. The Index Consultant provides recommendations on the ideal 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 used to analyze a database, the Index Consultant makes use of a tracing session to gather its 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, as well as single-column indexes, and also investigates the impact of clustered or uncluttered indexes.

The Index Consultant analyzes a database or single query by generating candidate indexes and exploring 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.

You must have DBA or PROFILE authority to run the Index Consultant.

Note
You can use Sybase Central to connect to a version 9 database server. However, when you do so, 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 how to locate and use Index Consultant in Sybase Central when connecting to a version 9 database server.

See also
- “Working with indexes” on page 86
- “Indexes” on page 617
- “Application profiling” on page 188
- “PROFILE authority” [SQL Anywhere Server - Database Administration]

Using the Index Consultant

The Index Consultant guides you in the proper selection of indexes. You can use the Index Consultant to obtain a set of index recommendations for an individual query, or for an entire database.

Obtaining index recommendations for a query
To obtain index recommendations for a single query, access the Index Consultant from Interactive SQL.

To obtain Index Consultant recommendations for a query (Sybase Central)
1. Connect to the database as a user with DBA authority or as a user with PROFILE authority.
2. Right-click the database and choose Open Interactive SQL.
3. In the SQL Statements pane, type the query.
Obtaining index 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 make 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. Connect to the database as the DBA, or as a user with 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.

See also

- “Working with indexes” on page 86
- “Indexes” on page 617
- “Application profiling” on page 188
- “Understanding Index Consultant recommendations” on page 195

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 indexes, as well as 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 87.

- **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.

The Index Consultant provides a set of tabs with the results of a given analysis. The results of an analysis can be saved for later review.

**Summary tab**

The **Summary** tab provides an overview of the analysis, including such information as 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. Among the information provided is the following:

- **Clustered**  Each table can have at most one clustered index. In some cases, a clustered index can provide significantly more benefit than an uncluttered index. See “Using clustered indexes” on page 87.

- **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 616.

- **Relative Benefit**  A number from one to ten, indicating the estimated overall benefit of creating the specified index. A higher number indicates a greater estimated 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 recommendations” on page 197.

- **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 541.

- **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, as well as 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 86
- “Indexes” on page 617
- “Application profiling” on page 188

Implementing Index Consultant recommendations

The Index Consultant provides a SQL script that you can run to implement its recommendations. However, before doing so, you may want to assess the recommendations in the light of your own knowledge of your database. For example, the names of the proposed indexes are generated from the name of the analysis. You may want to rename them before creating the indexes in the database.

The following is a list of questions you should consider when assessing the recommendations:

- **Do the proposed indexes match your own 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, it is possible that 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 195
- “Using SQL command files” on page 750
- “Working with indexes” on page 86
- “Indexes” on page 617
- “REORGANIZE TABLE statement” [*SQL Anywhere Server - SQL Reference*]
- “Application profiling” on page 188
Advanced application profiling using diagnostic tracing

Diagnostic tracing is an advanced method of application profiling. The diagnostic tracing data produced by the database server includes such things as the time stamps and connection ids of statements handled by the database server. For queries, it also includes isolation level, number of rows fetched, cursor type, and query execution plan. For INSERT, UPDATE, and DELETE statements, it also includes the number of rows affected. Diagnostic tracing can also record information about locking and deadlocks, and can 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, and can record all the information that request logging records, as well as detailed information to support tailored analysis. For information about request logging, see “Performing request trace analysis” on page 214.

See also

- “Application profiling” on page 188

Tracing session data

Diagnostic tracing data is gathered during a tracing session. A tracing session can be captured in one of three ways:

- using the Database Tracing Wizard in Sybase Central
- transparently, as part of the automated activities of the Application Profiling Wizard
- using the ATTACH TRACING and DETACH TRACING statements

When a tracing session is in progress, SQL Anywhere generates diagnostic information for that 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 201.
The database being profiled is either referred to as the **production database**, the source database, or simply 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, storing tracing data in a separate database is recommended since doing so avoids growing the size of the production database. This is important since database files cannot be reduced to a smaller size once 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 experiences heavy usage.

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.

Once you are done with the recommendations provided 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 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 215.

**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

The **Application Profiling Wizard** in Sybase Central uses pre-configured tracing settings, which you cannot change. However, the **Database Tracing Wizard**, used to perform diagnostic tracing, allows you to configure almost all aspects of your tracing activities. You configure diagnostic tracing settings in one of two ways:

- using the **Database Tracing Wizard** in Sybase Central. This method is recommended because it allows you to see all of the tracing settings that are in effect. See “Changing diagnostic tracing configuration settings” on page 209.

- using 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-level 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 204.

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, as well as 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 “Changing diagnostic tracing configuration settings” on page 209.

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 “Changing diagnostic tracing settings while a tracing session is in progress” on page 210.

Diagnostic tracing levels

The following is a list of diagnostic tracing levels and the types of information (diagnostic tracing types) that they include. These levels reflect the settings specified in the Database Tracing Wizard. For a description of the various diagnostic tracing types, see “Diagnostic tracing types” on page 204.

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 are gathered, as well as a sampling of executed statements (once every five seconds). For this level, the diagnostic tracing types include:
  - volatile_statistics, with sampling every 1 second
  - non_volatile_statistics, with sampling every 60 seconds

  This level has a negligible impact on performance.

- **Level 2**  This level gathers performance counters and records all executed statements, as well as a sampling of executed plans (once every five seconds). For this level, the diagnostic tracing types include:
  - volatile_statistics, with sampling every 1 second
  - non_volatile_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
  - non_volatile_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.

<table>
<thead>
<tr>
<th>Values in the scope column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATABASE</td>
<td>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.</td>
</tr>
<tr>
<td>ORIGIN</td>
<td>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.</td>
</tr>
<tr>
<td>USER</td>
<td>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.</td>
</tr>
<tr>
<td>CONNECTION_NAME, or CONNEC- TION_NUMBER</td>
<td>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.</td>
</tr>
</tbody>
</table>
Values in the scope column | Description
--- | ---
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 204
- “Diagnostic tracing conditions” on page 207

Diagnostic tracing types

Following is the list of types you can set choose from for diagnostic tracing. Diagnostic tracing types control the information for which to generate tracing data. 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 207.

The values in trace_type column reflect the settings specified in the **Database Tracing Wizard**.

<table>
<thead>
<tr>
<th>Value in the trace_type column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VOLATILE_STATISTICS</td>
<td>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 203, and “Diagnostic tracing conditions” on page 207.</td>
</tr>
<tr>
<td>Value in the trace_type column</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| NONVOLATILE_STATISTICS        | Collects a sample of database and server statistics that do not change frequently. Non-volatile statistics cannot be collected more frequently than 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 203, and “Diagnostic tracing conditions” on page 207. |
| 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 203, and “Diagnostic tracing conditions” on page 207. |
| 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 of the scopes, and can use any one of the following conditions for collection: NONE, NULL, SAMPLE_EVERY. See “Diagnostic tracing scopes” on page 203, and “Diagnostic tracing conditions” on page 207. |
| PLANS                         | Collects execution plans for queries, depending on the condition and scope.  

Scopes and conditions: This diagnostic tracing type can be used with all of 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 203, and “Diagnostic tracing conditions” on page 207. |
<table>
<thead>
<tr>
<th>Value in the trace_type column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLANS_WITH_STATISTICS</td>
<td>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 of the scopes, and accepts any one of the conditions for collection.</td>
</tr>
<tr>
<td>STATEMENTS</td>
<td>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 of the scopes, and can use any one of the conditions for collection. See “Diagnostic tracing scopes” on page 203, and “Diagnostic tracing conditions” on page 207.</td>
</tr>
<tr>
<td>STATEMENTS_WITH_VARIABLES</td>
<td>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 of the scopes, and can use any one of the conditions for collection. See “Diagnostic tracing scopes” on page 203, and “Diagnostic tracing conditions” on page 207.</td>
</tr>
<tr>
<td>OPTIMIZATION_LOGGING</td>
<td>Collects data about join strategies considered by the optimizer for execution of each query. Information about cost of execution of each strategy, as well as 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 of the scopes, and does not take a condition. See “Diagnostic tracing scopes” on page 203.</td>
</tr>
</tbody>
</table>
See also

- “Diagnostic tracing scopes” on page 203
- “Diagnostic tracing conditions” on page 207

### Diagnostic tracing conditions

Following is the list of conditions you can set for diagnostic tracing. 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.

<table>
<thead>
<tr>
<th>Value in the trace_condition column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NONE, or NULL</td>
<td>Records all of 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.</td>
</tr>
<tr>
<td>SAMPLE_EVERY</td>
<td>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.</td>
</tr>
</tbody>
</table>
### Determining 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. Connect to the database as a user with DBA authority or as a user with 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.

---

<table>
<thead>
<tr>
<th>Value in the <code>trace_condition</code> column</th>
<th>Description</th>
</tr>
</thead>
</table>
| 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 203
- “Diagnostic tracing types” on page 204
Example

The following statement shows you how to query the `sa_diagnostic_tracing_level` diagnostic table to retrieve the current diagnostic tracing settings:

```sql
SELECT * FROM sa_diagnostic_tracing_level WHERE enabled = 1;
```

The following table is an example result set from the query:

<table>
<thead>
<tr>
<th>id</th>
<th>scope</th>
<th>identifier</th>
<th>trace_type</th>
<th>trace_condition</th>
<th>value</th>
<th>enabled</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>database</td>
<td>(NULL)</td>
<td>volatile_statistics</td>
<td>sample_every</td>
<td>1,000</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>database</td>
<td>(NULL)</td>
<td>nonvolatile_statistics</td>
<td>sample_every</td>
<td>60,000</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>database</td>
<td>(NULL)</td>
<td>connection_statistics</td>
<td>(NULL)</td>
<td>60,000</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>database</td>
<td>(NULL)</td>
<td>blocking</td>
<td>(NULL)</td>
<td>(NULL)</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>database</td>
<td>(NULL)</td>
<td>deadlock</td>
<td>(NULL)</td>
<td>(NULL)</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>database</td>
<td>(NULL)</td>
<td>plans_with_statistics</td>
<td>sample_every</td>
<td>2,000</td>
<td>1</td>
</tr>
</tbody>
</table>

See also

- “sa_diagnostic_tracing_level table” [SQL Anywhere Server - SQL Reference]
- “PROFILE authority” [SQL Anywhere Server - Database Administration]

Changing 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 “Creating a diagnostic tracing session” on page 210.

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 pre-configured 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. Connect to the database as a user with DBA authority or as a user with 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:

```sql
```
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 202.

Changing diagnostic tracing settings while 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. Connect to the database as a user with DBA authority or as a user with 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.

Creating 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. Connect to the database as a user with DBA authority or as a user with 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, `ENG=Server47;DBN=TracingDB`

   **Note**
   Only DBN, DBF, ENG, DBKEY 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 188.

**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;ENG=dbsrv11;DBN=tracing;LINKS=tcpip' 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 corresponding to the diagnostic tracing levels and settings configured for the tracing session. Consequently, 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 263

## Troubleshooting performance problems

Often, performance problems are a combination of several factors. Use the application profiling feature to determine whether performance problems are the result such things as:

- long processing times by the application
- poor query plans
- contention for shared hardware resources such as CPU or disk I/O
- contention for database objects
- suboptimal database design

The first task in a performance troubleshooting scenario is to determine whether the application or the database server is the primary cause of any observed slowdown. If the time that the application client spends processing data it receives accounts for most of the total run time, the database server may be idle much of the time relative to the application. This can be detected by looking at the Details tab in the application profiling tool (choose to filter the results by a single connection). Observe whether there are large gaps of time between different requests from that connection. If so, then the primary delay is within the application client itself.
If the database server accounts for the majority of the slowdown, the next step is to identify the specific cause of the slowdown.

See also

- “Application profiling tutorials” on page 263

Detecting when hardware resources are a limiting factor

As larger and larger workloads are placed on a database system, performance is eventually limited by one or more hardware resources. Typically one of three resources is exhausted: CPU cycles, memory space, or disk I/O bandwidth. When this happens, it may be because there are inefficiencies in the application or in how the database server is performing. If no inefficiencies are detected, you may need to add additional hardware resources to allow the database server handle larger workloads, and to improve performance on the existing workload. To view a list of common inefficiencies and recommendations for solving them, see “Troubleshooting performance problems” on page 212.

Adding resources may not resolve all scalability problems. Furthermore, even when resources are added, the capability of the computer is rarely improved in a linear fashion. For example, if it appears that the database server is fully using the allotted CPUs, it indicates that more CPU resources must be assigned in order to handle a larger workload. However, doubling the number of CPUs available to the database server likely will not double the amount of work the database server can perform in the same amount of time.

You can examine several database statistics using 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 227
- “Application profiling tutorials” on page 263
- “Troubleshooting performance problems” on page 212

### 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 (procedures, triggers, and so on), 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 857.

### Performing 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**

1. Connect to the database as a user with DBA authority or as a user with 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 have a minimal impact on performance.

The easiest method for creating an external tracing database is to do so while starting a tracing session using the **Database Tracing Wizard**. The **Database Tracing Wizard** unloads schema and permission information from the production database when creating the tracing database. The tracing database that you create can be used to store data for subsequent tracing sessions as well. For information about how to create a tracing session, see “Creating a diagnostic tracing session” on page 210.

If you do not want to create a tracing session, but want only to create a tracing database, you can do so manually using the Unload utility (dbunload).

### 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:

   ```sh
dbunload -c "UID=DBA;PWD=sql;ENG=demo;DBN=demo" -an tracing.db -n -k
   ```
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.

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]
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 of 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 199.

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 server option” [SQL Anywhere Server - Database Administration]
- “-zo server option” [SQL Anywhere Server - Database Administration]
- “-zn server option” [SQL Anywhere Server - Database Administration]
- “-zs 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].

**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:

```sql
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 190.

## Enabling profiling using `sa_server_option`

This section explains how to enable procedure profiling from Interactive SQL using the `sa_server_option` system procedure.

### 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:

   ```sql
   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:

   ```sql
   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]
Resetting 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:
  
  ```sql
  CALL sa_server_option( 'ProcedureProfiling' , 'RESET' );
  ```

**See also**

- “`sa_server_option` system procedure” [SQL Anywhere Server - SQL Reference]

Disabling 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:
  
  ```sql
  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:
  
  ```sql
  CALL sa_server_option( 'ProcedureProfiling' , 'CLEAR' );
  ```

**See also**

- “`sa_server_option` system procedure” [SQL Anywhere Server - SQL Reference]
Retrieving 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 “Enabling profiling using sa_server_option” on page 219.

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:
   ```sql
   CALL sa_procedure_profile_summary;
   ```
3. Choose SQL » Execute.
   A result set with information about all of 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:
   ```sql
   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 transtest, are available in samples-dir SQLAnywhere. For more 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 transtest 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 223.

- **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 224.

- **Windows Performance Monitor** This is a monitoring tool provided by your Windows operating system. See “Monitoring statistics using Windows Performance Monitor” on page 226.

- **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 199.

For a complete listing of the SQL Anywhere statistics available for monitoring, see “Performance Monitor statistics” on page 227.

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:

```sql
SET server_name = PROPERTY( 'name' );
```

The following query returns the user ID for the current connection:

```sql
SELECT CONNECTION_PROPERTY( 'UserID' );
```

The following query returns the file name for the root file of the current database:

```sql
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:

```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 224.

Monitoring statistics using Sybase Central Performance Monitor
The Sybase Central Performance Monitor is useful for tracking detailed information 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 “Monitoring statistics using Windows Performance Monitor” on page 226.

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 227.

**Opening 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 “Adding and removing statistics” on page 225.

**To open the Performance Monitor**

1. In the left pane, select the server.
2. In the right pane, click the Performance Monitor tab.

**See also**

- “Monitoring statistics using Windows Performance Monitor” on page 226
- “Adding and removing statistics” on page 225

**Adding and removing statistics**

**To add statistics to the Sybase Central Performance Monitor**

1. In the left pane, select the server.
2. In the right pane, click the Statistics tab.
3. 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. In the left pane, select the server.
2. In the right pane, click the Statistics tab.
3. 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 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 227.

See also
● “Opening the Sybase Central Performance Monitor” on page 225
● “Monitoring statistics using Windows Performance Monitor” on page 226

Monitoring 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.

The Windows Performance Monitor is available on Windows. 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 227.

There are database server options you can specify when starting the database server that control the memory used by the Windows Performance Monitor, as well as the maximum number of connections or database that the Performance Monitor can monitor. See:

● “-ks server option” [SQL Anywhere Server - Database Administration]
● “-ksc server option” [SQL Anywhere Server - Database Administration]
● “-ksd server option” [SQL Anywhere Server - Database Administration]

The following instructions explain how to start Windows Performance Monitor on Microsoft 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 11 Connection**  This monitors performance for a single connection. A connection must currently exist to see this selection.

• **SQL Anywhere 11 Database**  This monitors performance for a single database.

• **SQL Anywhere 11 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 11 Connection or SQL Anywhere 11 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 makes a large variety of statistics available. Rates are reported in 1 second intervals. The statistics are grouped into the following areas:

- “Cache statistics” on page 227
- “Checkpoint and recovery statistics” on page 228
- “Communications statistics” on page 230
- “Disk I/O statistics” on page 231
- “Disk read statistics” on page 231
- “Disk write statistics” on page 232
- “Index statistics” on page 233
- “Memory pages statistics” on page 235
- “Request statistics” on page 236
- “Miscellaneous statistics” on page 237

### Cache statistics

These statistics describe the use of the cache.

<table>
<thead>
<tr>
<th>Statistic</th>
<th>Scope</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache Hits/sec</td>
<td>Connection and database</td>
<td>Shows the rate at which database page lookups are satisfied by finding the page in the cache.</td>
</tr>
</tbody>
</table>
### Checkpoint and recovery statistics

These statistics isolate the checkpoint and recovery actions performed when the database is in an idle state.

<table>
<thead>
<tr>
<th>Statistic</th>
<th>Scope</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Checkpoint Flushes/sec</td>
<td>Database</td>
<td>Shows the rate at which ranges of adjacent pages are written out during a checkpoint.</td>
</tr>
<tr>
<td>Checkpoint Urgency</td>
<td>Database</td>
<td>Shows the checkpoint urgency, expressed as a percentage.</td>
</tr>
<tr>
<td>Statistic</td>
<td>Scope</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------</td>
<td>----------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Checkpoints/sec</td>
<td>Database</td>
<td>Shows the rate at which checkpoints are performed.</td>
</tr>
<tr>
<td>ChkptLog: Bitmap size</td>
<td>Database</td>
<td>Shows the size of the checkpoint log bitmap.</td>
</tr>
<tr>
<td>ChkptLog: Commit to disk/sec</td>
<td>Database</td>
<td>Shows the rate at which checkpoint log commit_to_disk operations are being performed.</td>
</tr>
<tr>
<td>ChkptLog: Log size</td>
<td>Database</td>
<td>Shows the size of the checkpoint log in pages.</td>
</tr>
<tr>
<td>ChkptLog: Page images saved/sec</td>
<td>Database</td>
<td>Shows the rate at which pages are being saved in the checkpoint log prior to modification.</td>
</tr>
<tr>
<td>ChkptLog: Pages in use</td>
<td>Database</td>
<td>Shows the number of pages in the checkpoint log which are currently in use.</td>
</tr>
<tr>
<td>ChkptLog: Relocate pages/sec</td>
<td>Database</td>
<td>Shows the rate at which pages in the checkpoint log are being relocated.</td>
</tr>
<tr>
<td>ChkptLog: Save preimage/sec</td>
<td>Database</td>
<td>Shows the rate at which new database page preimages are being added to the checkpoint log.</td>
</tr>
<tr>
<td>ChkptLog: Write pages/sec</td>
<td>Database</td>
<td>Shows the rate at which pages are being written to the checkpoint log.</td>
</tr>
<tr>
<td>ChkptLog: Writes/sec</td>
<td>Database</td>
<td>Shows the rate at which disk writes are being performed in the checkpoint log. One write can include multiple pages.</td>
</tr>
<tr>
<td>ChkptLog: Writes to bitmap/sec</td>
<td>Database</td>
<td>Shows the rate at which disk writes are being performed in the checkpoint log for bitmap pages.</td>
</tr>
<tr>
<td>Idle Actives/sec</td>
<td>Database</td>
<td>Shows the rate at which the database server’s idle thread becomes active to do idle writes, idle checkpoints, and so on.</td>
</tr>
<tr>
<td>Idle Checkpoint Time</td>
<td>Database</td>
<td>Shows the total time spent doing idle checkpoints, in seconds.</td>
</tr>
<tr>
<td>Idle Checkpoints/sec</td>
<td>Database</td>
<td>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.</td>
</tr>
<tr>
<td>Idle Writes/sec</td>
<td>Database</td>
<td>Shows the rate at which disk writes are issued by the database server's idle thread.</td>
</tr>
</tbody>
</table>


### Communications statistics

These statistics describe client/server communications activity.

<table>
<thead>
<tr>
<th>Statistic</th>
<th>Scope</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comm: Bytes Received/sec</td>
<td>Connection and server</td>
<td>Shows the rate at which network data (in bytes) are received.</td>
</tr>
<tr>
<td>Comm: Bytes Received Uncompressed/sec</td>
<td>Connection and server</td>
<td>Shows the rate at which bytes would have been received if compression was disabled.</td>
</tr>
<tr>
<td>Comm: Bytes Sent/sec</td>
<td>Connection and server</td>
<td>Shows the rate at which bytes are transmitted over the network.</td>
</tr>
<tr>
<td>Comm: Bytes Sent Uncompressed/sec</td>
<td>Connection and server</td>
<td>Shows the rate at which bytes would have been sent if compression was disabled.</td>
</tr>
<tr>
<td>Comm: Free Buffers</td>
<td>Server</td>
<td>Shows the number of free network buffers.</td>
</tr>
<tr>
<td>Comm: Multi-packets Received/sec</td>
<td>Server</td>
<td>Shows the rate at which multi-packet deliveries are received.</td>
</tr>
<tr>
<td>Comm: Multi-packets Sent/sec</td>
<td>Server</td>
<td>Shows the rate at which multi-packet deliveries are transmitted.</td>
</tr>
<tr>
<td>Comm: Packets Received/sec</td>
<td>Connection and server</td>
<td>Shows the rate at which network packets are received.</td>
</tr>
<tr>
<td>Comm: Packets Received Uncompressed/sec</td>
<td>Connection and server</td>
<td>Shows the rate at which network packets would have been received if compression was disabled.</td>
</tr>
<tr>
<td>Comm: Packets Sent/sec</td>
<td>Connection and server</td>
<td>Shows the rate at which network packets are transmitted.</td>
</tr>
</tbody>
</table>
**Statistic** | **Scope** | **Description**
---|---|---
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.

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.

<table>
<thead>
<tr>
<th>Statistic</th>
<th>Scope</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disk: Active I/Os</td>
<td>Database</td>
<td>Shows the current number of file I/Os issued by the database server which have not yet completed.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Statistic</th>
<th>Scope</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disk: Maximum Active I/Os</td>
<td>Database</td>
<td>Shows the maximum value &quot;Disk: Active I/Os&quot; has reached.</td>
</tr>
</tbody>
</table>

**Disk read statistics**
These statistics describe the amount and type of activity devoted to reading information from disk.
### Disk Reads Statistics

<table>
<thead>
<tr>
<th>Statistic</th>
<th>Scope</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disk Reads: Total Pages/sec</td>
<td>Connection and database</td>
<td>Shows the rate at which pages are read from a file.</td>
</tr>
<tr>
<td>Disk Reads: Active</td>
<td>Database</td>
<td>Shows the current number of file reads issued by the database server which haven't yet completed.</td>
</tr>
<tr>
<td>Disk Reads: Index interior/sec</td>
<td>Connection and database</td>
<td>Shows the rate at which index internal-node pages are being read from disk.</td>
</tr>
<tr>
<td>Disk Reads: Index leaf/sec</td>
<td>Connection and database</td>
<td>Shows the rate at which index leaf pages are being read from disk.</td>
</tr>
<tr>
<td>Disk Reads: Table/sec</td>
<td>Connection and database</td>
<td>Shows the rate at which table pages are being read from disk.</td>
</tr>
<tr>
<td>Disk Reads: Maximum Active</td>
<td>Database</td>
<td>Shows the maximum value &quot;Disk Reads: Active&quot; has reached.</td>
</tr>
<tr>
<td>Disk Reads: Work Table</td>
<td>Connection and database</td>
<td>Shows the rate at which work table pages are being read from disk.</td>
</tr>
</tbody>
</table>

### Disk Write Statistics

These statistics describe the amount and type of activity devoted to writing information to disk.

<table>
<thead>
<tr>
<th>Statistic</th>
<th>Scope</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disk Writes: Active</td>
<td>Database</td>
<td>Shows the current number of file writes issued by the database server that aren't yet completed.</td>
</tr>
<tr>
<td>Disk Writes: Maximum Active</td>
<td>Database</td>
<td>Shows the maximum value &quot;Disk Writes: Active&quot; has reached.</td>
</tr>
<tr>
<td>Disk Writes: Commit Files/sec</td>
<td>Database</td>
<td>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.</td>
</tr>
<tr>
<td>Disk Writes: Database Extends/sec</td>
<td>Database</td>
<td>Shows the rate at which the database file is extended, in pages/sec.</td>
</tr>
<tr>
<td>Disk Writes: Temp Extends/sec</td>
<td>Database</td>
<td>Shows the rate at which temporary files are extended, in pages/sec.</td>
</tr>
</tbody>
</table>
### Disk Writes: Pages/sec
Shows the rate at which modified pages are being written to disk.

### Disk Writes: Transaction Log/sec
Shows the rate at which pages are written to the transaction log.

### Translog Group Commits/sec
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.

#### Index: Adds/sec
Shows the rate at which entries are added to indexes.

#### Index: Lookups/sec
Shows the rate at which entries are looked up in indexes.

#### Index: Full Compares/sec
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.

#### Cache: Multi-Page Allocations
Shows the number of multi-page allocations.

#### Cache: Panics
Shows the number of times the cache manager has failed to find a page to allocate.

#### Cache: Scavenge Visited
Shows the number of pages visited while scavenging for a page to allocate.

#### Cache: Scavenges
Shows the number of times the cache manager has scavenged for a page to allocate.
<table>
<thead>
<tr>
<th>Statistic</th>
<th>Scope</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache Pages: Allocated Structures</td>
<td>Server</td>
<td>Shows the number of cache pages that have been allocated for database server data structures.</td>
</tr>
<tr>
<td>Cache Pages: File</td>
<td>Server</td>
<td>Shows the number of cache pages used to hold data from database files.</td>
</tr>
<tr>
<td>Cache Pages: File Dirty</td>
<td>Server</td>
<td>Shows the number of cache pages that are dirty (needing a write).</td>
</tr>
<tr>
<td>Cache Pages: Free</td>
<td>Server</td>
<td>Shows the number of cache pages not being used.</td>
</tr>
<tr>
<td>Cache Pages: Pinned</td>
<td>Server</td>
<td>Shows the number of pages currently unavailable for reuse.</td>
</tr>
<tr>
<td>Cache Replacements: Total Pages/sec</td>
<td>Server</td>
<td>Shows the rate at which database pages are being purged from the cache to make room for another page that is needed.</td>
</tr>
<tr>
<td>Heaps: Carver</td>
<td>Connection and server</td>
<td>Shows the number of heaps used for short-term purposes such as query optimization.</td>
</tr>
<tr>
<td>Heaps: Query Processing</td>
<td>Connection and server</td>
<td>Shows the number of heaps used for query processing (hash and sort operations).</td>
</tr>
<tr>
<td>Heaps: Relocatable</td>
<td>Connection and server</td>
<td>Shows the number of relocatable heaps.</td>
</tr>
<tr>
<td>Heaps: Relocatable Locked</td>
<td>Connection and server</td>
<td>Shows the number of relocatable heaps currently locked in the cache.</td>
</tr>
<tr>
<td>Map physical memory/sec</td>
<td>Server</td>
<td>Shows the rate at which database page address space windows are being mapped to physical memory in the cache using Address Windowing Extensions.</td>
</tr>
<tr>
<td>Mem Pages: Carver</td>
<td>Connection and server</td>
<td>Shows the number of heap pages used for short-term purposes such as query optimization.</td>
</tr>
<tr>
<td>Statistic</td>
<td>Scope</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>-----------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Mem Pages: Pinned Cursor</td>
<td>Server</td>
<td>Shows the number of pages used to keep cursor heaps pinned in memory.</td>
</tr>
<tr>
<td>Mem Pages: Query Processing</td>
<td>Connection and server</td>
<td>Shows the number of cache pages used for query processing (hash and sort operations).</td>
</tr>
<tr>
<td>Query Memory: Current Active</td>
<td>Connection and server</td>
<td>Shows the current number of requests actively using query memory.</td>
</tr>
<tr>
<td>Query Memory: Estimated Active</td>
<td>Server</td>
<td>Shows the database server's estimate of the steady state average of the number of requests actively using query memory.</td>
</tr>
<tr>
<td>Query Memory: Extra Available</td>
<td>Server</td>
<td>Shows the amount of memory available to grant beyond the base memory-intensive grant.</td>
</tr>
<tr>
<td>Query Memory: Number of Grant Fails</td>
<td>Connection and server</td>
<td>Shows the total number of times any request waited for query memory and failed to get it.</td>
</tr>
<tr>
<td>Query Memory: Number of Grant Requests</td>
<td>Connection and server</td>
<td>Shows the total number of times any request attempted to acquire query memory.</td>
</tr>
<tr>
<td>Query Memory: Number of Grant Waits</td>
<td>Connection and server</td>
<td>Shows the total number of times any request waited for memory.</td>
</tr>
<tr>
<td>Query Memory: Pages Granted</td>
<td>Connection and server</td>
<td>Shows the number of pages currently granted to requests.</td>
</tr>
<tr>
<td>Query Memory: Requests Waiting</td>
<td>Connection and server</td>
<td>Shows the current number of requests waiting for query memory.</td>
</tr>
</tbody>
</table>

**Memory pages statistics**

These statistics describe the amount and purpose of memory used by the database server.

<table>
<thead>
<tr>
<th>Statistic</th>
<th>Scope</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mem Pages: Lock Table</td>
<td>Database</td>
<td>Shows the number of pages used to store lock information.</td>
</tr>
<tr>
<td>Mem Pages: Locked Heap</td>
<td>Server</td>
<td>Shows the number of heap pages locked in the cache.</td>
</tr>
</tbody>
</table>
### Statistic

<table>
<thead>
<tr>
<th>Statistic</th>
<th>Scope</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mem Pages: Main Heap</td>
<td>Server</td>
<td>Shows the number of pages used for global database server data structures.</td>
</tr>
<tr>
<td>Mem Pages: Map Pages</td>
<td>Database</td>
<td>Shows the number of map pages used for accessing the lock table, frequency table, and table layout.</td>
</tr>
<tr>
<td>Mem Pages: Procedure Definitions</td>
<td>Database</td>
<td>Shows the number of relocatable heap pages used for procedures.</td>
</tr>
<tr>
<td>Mem Pages: Relocatable</td>
<td>Database</td>
<td>Shows the number of pages used for relocatable heaps (cursors, statements, procedures, triggers, views, and so on).</td>
</tr>
<tr>
<td>Mem Pages: Relocations/sec</td>
<td>Database</td>
<td>Shows the rate at which relocatable heap pages are read from the temporary file.</td>
</tr>
<tr>
<td>Mem Pages: Rollback Log</td>
<td>Connection and database</td>
<td>Shows the number of pages in the rollback log.</td>
</tr>
<tr>
<td>Mem Pages: Trigger Definitions</td>
<td>Database</td>
<td>Shows the number of relocatable heap pages used for triggers.</td>
</tr>
<tr>
<td>Mem Pages: View Definitions</td>
<td>Database</td>
<td>Shows the number of relocatable heap pages used for views.</td>
</tr>
</tbody>
</table>

### Request statistics

These statistics describe the database server activity devoted to responding to requests from client applications.

<table>
<thead>
<tr>
<th>Statistic</th>
<th>Scope</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cursors</td>
<td>Connection</td>
<td>Shows the number of declared cursors currently maintained by the database server.</td>
</tr>
<tr>
<td>Cursors Open</td>
<td>Connection</td>
<td>Shows the number of open cursors currently maintained by the database server.</td>
</tr>
<tr>
<td>Lock Count</td>
<td>Connection and database</td>
<td>Shows the number of locks.</td>
</tr>
<tr>
<td>Requests/sec</td>
<td>Server</td>
<td>Shows the rate at which the database server is entered to allow it to handle a new request or continue processing an existing request.</td>
</tr>
<tr>
<td>Statistic</td>
<td>Scope</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------</td>
<td>----------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Requests: Active</td>
<td>Server</td>
<td>Shows the number of database server threads that are currently handling a request.</td>
</tr>
<tr>
<td>Tasks: Exchange</td>
<td>Server</td>
<td>Shows the number of database server threads that are currently being used for parallel execution of a query.</td>
</tr>
<tr>
<td>Requests: Unscheduled</td>
<td>Server</td>
<td>Shows the number of requests that are currently queued up waiting for an available database server thread.</td>
</tr>
<tr>
<td>Snapshot Count</td>
<td>Connection and database</td>
<td>Shows the number of active snapshots.</td>
</tr>
<tr>
<td>Statement Cache Hits</td>
<td>Connection and server</td>
<td>Shows the rate at which statement prepares cached by the client are being re-used by the database server.</td>
</tr>
<tr>
<td>Statement Cache Misses</td>
<td>Connection and server</td>
<td>Shows the rate at which statement prepares cached by the client need to be prepared again by the database server.</td>
</tr>
<tr>
<td>Statement Prepares</td>
<td>Connection</td>
<td>Shows the rate at which statement prepares are being handled by the database server.</td>
</tr>
<tr>
<td>Statements</td>
<td>Connection</td>
<td>Shows the number of prepared statements currently maintained by the database server.</td>
</tr>
<tr>
<td>Transaction Commits</td>
<td>Connection</td>
<td>Shows the rate at which Commit requests are handled.</td>
</tr>
<tr>
<td>Transaction Rollbacks</td>
<td>Connection</td>
<td>Shows the rate at which Rollback requests are handled.</td>
</tr>
</tbody>
</table>

**Miscellaneous statistics**

<table>
<thead>
<tr>
<th>Statistic</th>
<th>Scope</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avail IO</td>
<td>Server</td>
<td>Shows the current number of available I/O control blocks.</td>
</tr>
<tr>
<td>Connection Count</td>
<td>Database</td>
<td>Shows the number of connections to this database.</td>
</tr>
<tr>
<td>Main Heap Bytes</td>
<td>Server</td>
<td>Shows the number of bytes used for global database server data structures.</td>
</tr>
<tr>
<td>Statistic</td>
<td>Scope</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Query: Plan cache pages</td>
<td>Connection and database</td>
<td>Shows the number of pages used to cache execution plans.</td>
</tr>
<tr>
<td>Query: Low memory strategies</td>
<td>Connection and database</td>
<td>Shows the number of times the database server changed its execution plan during execution because of low memory conditions.</td>
</tr>
<tr>
<td>Query: Rows materialized/sec</td>
<td>Connection and database</td>
<td>Shows the rate at which rows are written to work tables during query processing.</td>
</tr>
<tr>
<td>Requests: GET DATA/sec</td>
<td>Connection and database</td>
<td>Shows the rate at which a connection is issuing GET DATA requests.</td>
</tr>
<tr>
<td>Temporary Table Pages</td>
<td>Connection and database</td>
<td>Shows the number of pages in the temporary file used for temporary tables.</td>
</tr>
<tr>
<td>Version Store Pages</td>
<td>Database</td>
<td>Shows the number of pages of the temporary file currently being used for the row version store when snapshot isolation is enabled.</td>
</tr>
</tbody>
</table>
Performance improvement tips

SQL Anywhere provides excellent performance automatically. However, the following tips help you achieve the most from the product.

Acquire adequate hardware

When running on a PC, the following are the minimum CPU, memory, and disk requirements:

- A minimum of 4 MB of memory. If you are using the administration tools, such as Sybase Central and Interactive SQL, a minimum of 32 MB of RAM is required.
- Enough disk space to hold your database and log files.

If your server meets only the minimum hardware requirements, performance may suffer; you might want to upgrade your hardware. In general, evaluate the hardware configuration to see if it is adequate for the kind of work load being placed on the database server.

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.

For more information, see “-fc server option” [SQL Anywhere Server - Database Administration].

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 a number of 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-level 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), then SQL Anywhere optimizes a query so as 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 [database]” [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 clause” [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 103.

**Increase the cache size**

SQL Anywhere stores recently used pages in a cache. Should a request need to access the page more than once, or should another connection require the same page, it may find it already in memory and hence avoid having to read information from disk. This is especially an issue for encrypted databases, which require a larger cache than unencrypted.

If your cache is too small, SQL Anywhere cannot keep pages in memory long enough to reap these benefits.

On Unix and Windows, the database server dynamically changes cache size as needed. However, the cache is still limited by the amount of memory that is physically available, and by the amount used by other applications.
Tip
Increasing the cache size can often improve performance dramatically, since retrieving information from memory is many times faster than reading it from disk. You may find it worthwhile to add more RAM to allow a larger cache.

See “Use the cache to improve performance” on page 254.

Minimize cascading referential actions

Cascading referential actions are costly in terms of performance because they cause updates to multiple tables for every transaction. 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 “Introduction to data integrity” on page 94.

Monitor query performance

SQL Anywhere includes a number of 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

tranetest
Function Measures the load that can be handled by a given database server configuration given a database design and a set of transactions.
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 the people in your company work at a number of 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 “Database design concepts” on page 5.

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 has to skip over any 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
- “Database design concepts” on page 5
- “Managing primary keys” on page 43

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)
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, as well as any dbspaces, appropriate to your needs.

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 “Changing 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 a sufficient number of 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
[database]” [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 server option” [SQL Anywhere Server -
Database Administration].

See also

● “Use work tables in query processing (use All-rows optimization goal)” on page 260
● “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,
thus filling space and improving performance, much like defragmenting your disk drive. It also gives you
the opportunity to change certain settings. See “Rebuilding databases” on page 736.

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 a number of 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.

Reducing file fragmentation

Performance can suffer if your database file is excessively fragmented. This is disk fragmentation and it becomes more important as your database increases in size.

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-level 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.

Reducing table fragmentation

Table fragmentation occurs when rows are not stored contiguously, or when rows are split between multiple pages. Performance decreases because these rows require additional page accesses. Table fragmentation is distinct from file fragmentation.

The effect that fragmentation has on performance varies from one situation to the next. 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. At the other end of the scale, a fragmented table may cause much more I/O to be done and can result in a significant performance hit 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 reduces fragmentation, doing so too frequently or not frequently enough, can also impact performance. Experiment using the tools and methods described in the section below 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.

Determine the degree of table fragmentation

Use the sa_table_fragmentation system procedure to obtain information about the degree of fragmentation of your database tables. Running this system procedure just once is not helpful in determining whether to
defragment to improve performance. Instead, rebuild your database and run the procedure to establish baseline results. Then, continue to run it periodically over an extended length of time, looking for correlation between the change in its output to changes in performance measures. In this way you can determine the rate at which tables become fragmented to the degree that performance is impacted, and thus determine the optimal frequency at which to defragment tables.

You must have DBA authority to run this procedure. The following statement calls the sa_table_fragmentation system procedure:

```
CALL sa_table_fragmentation( [ 'table-name' [, 'owner-name' ] ] );
```

Methods to reduce 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.

- **Reorganize tables** You can defragment specific tables using the REORGANIZE TABLE statement. Reorganizing tables does not disrupt database access.

- **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 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]
- “Rebuild utility (rebuild)” [SQL Anywhere Server - Database Administration]

Reducing 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 189.
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>
<thead>
<tr>
<th>TableName</th>
<th>TableId</th>
<th>IndexName</th>
<th>IndexID</th>
<th>IndexType</th>
<th>LeafPages</th>
<th>Density</th>
<th>Skew</th>
</tr>
</thead>
<tbody>
<tr>
<td>Customers</td>
<td>686</td>
<td>CustKey</td>
<td>0</td>
<td>PKEY</td>
<td>1</td>
<td>0.645992</td>
<td>1.002772</td>
</tr>
<tr>
<td>Customers</td>
<td>686</td>
<td>IX_cust_name</td>
<td>1</td>
<td>NUI</td>
<td>1</td>
<td>0.789795</td>
<td>1.432239</td>
</tr>
</tbody>
</table>

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].

See also

- “sa_index_density system procedure” [SQL Anywhere Server - SQL Reference]
- “Working with indexes” on page 86

### 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 43
- “Use keys to improve query performance” on page 254

### 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 “Database design concepts” on page 5.
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 connection parameter [LCLOSE]” [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 814.

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 818.

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 362, or “The GROUP BY clause: Organizing query results into groups” on page 355.

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.

The cost of using autocommit mode is particularly high if you are not using a transaction log file. 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 whichever page size you choose.

While smaller pages hold less information and may force less efficient use of space, particularly if you insert rows that are slightly more than half a page in size, small page sizes allow SQL Anywhere to run with fewer
resources because it can store more pages in a cache of the same size. Small pages are particularly useful if your database must run on small computers with limited memory. They can also help in situations when you use your database primarily to retrieve small pieces of information from random locations.

By contrast, a larger page size help SQL Anywhere read databases more efficiently. Large page sizes also 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 particularly 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 702.

**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 [database]” [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 of the database files within the cache.
You can choose between two different in-memory modes. In never-write mode, committed transactions are not be written to the database file on disk. 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 server option” [SQL Anywhere Server - Database Administration].

Separately licensed component required
In-memory mode requires a separate license. See “Separately licensed components” [SQL Anywhere 11 - 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 people, but you only know either their first or their 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 86.

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 194.

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 617, and “Using clustered indexes” on page 87.
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 “Relations between tables” [SQL Anywhere 11 - 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.

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( 'CacheSize' );
```

See also

- “-c server option” [SQL Anywhere Server - Database Administration]
- “-ca server option” [SQL Anywhere Server - Database Administration]
- “-ch server option” [SQL Anywhere Server - Database Administration]
- “-cl server option” [SQL Anywhere Server - Database Administration]
Limiting the memory used by the cache

The initial, minimum, and maximum cache sizes are all controllable from the database server command line.

- **Initial cache size**  You can change the initial cache size by specifying the database server -c option. The default value is as follows:
  - **Windows Mobile**  The formula is as follows:
    \[
    \text{max}( 600 \text{ KB}, \min( \text{dbsize}, \text{physical-memory} ) )
    \]
    
    The \textit{dbsize} is the total size of the database file or files started, and \textit{physical-memory} is 25\% of the physical memory on the computer.
  - **Windows**  The formula is as follows:
    \[
    \text{max}( 2 \text{ MB}, \min( \text{dbsize}, \text{physical-memory} ) )
    \]
    
    The \textit{dbsize} is the total size of the database file or files started, and \textit{physical-memory} is 25\% of the physical memory on the computer.
  - **Unix**  At least 8 MB.

  For information about Unix initial cache size, see “Dynamic cache sizing on Unix” on page 257.

- **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:
  - **32-bit Unix platforms**, it is the lesser of 90\% of total physical memory or 1,834,880 KB.
  - **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.

You can also disable dynamic cache sizing by using the -ca 0 server option.

The following server properties return information about the database server cache:

- **MinCacheSize**  Returns the minimum allowed cache size, in kilobytes.
- **MaxCacheSize**  Returns the maximum allowed cache size, in kilobytes.
- **CurrentCacheSize**  Returns the current 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 “Server-level properties” [SQL Anywhere Server - Database Administration].

See also

- “-c server option” [SQL Anywhere Server - Database Administration]
- “-ca server option” [SQL Anywhere Server - Database Administration]
- “-ch server option” [SQL Anywhere Server - Database Administration]
- “-cl server option” [SQL Anywhere Server - Database Administration]

Dynamic cache sizing

SQL Anywhere provides automatic resizing of the database cache. The capabilities are different on different operating systems. On Windows and Unix operating systems, the cache grows and shrinks. Details are provided in the following sections.

Full dynamic cache sizing helps to ensure that the performance of your database server is not impacted by allocating inadequate memory. 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, so that the database server does not unduly impact other applications on the system. The effectiveness of dynamic cache sizing is limited, of course, by the physical memory available on your system.

Generally, dynamic cache sizing assesses cache requirements at the rate of approximately once per minute. However, after a new database is started or when a file grows significantly, statistics are sampled and the cache may be resized every five seconds for thirty seconds. After the initial thirty second period, the sampling rate drops back down to once per minute. Significant growth of a file is defined as a 1/8 growth since the database started or since the last growth that triggered an increase in the sampling rate. 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.

Dynamic cache sizing removes the need for explicit configuration of database cache in many situations, making SQL Anywhere even easier to use.

When an Address Windowing Extensions (AWE) cache is used, dynamic cache sizing is disabled. You cannot use an AWE cache on Windows Mobile.

For more information about AWE caches, see “--cw server option” [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 server option” [SQL Anywhere Server - Database Administration].

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 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 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 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

**Note**
The Windows Performance Monitor is available on Windows.

For more information about these properties, see “Server-level properties” [SQL Anywhere Server - Database Administration].

For information about monitoring performance, see “Monitoring database performance” on page 223.

**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.

For more information about the database server options used for cache warming, see “-cc server
option” [SQL Anywhere Server - Database Administration], “-cr server option” [SQL Anywhere Server -
Database Administration], and “-cv server option” [SQL Anywhere Server - Database Administration].

Use the compression features

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.

See also

● “Adjusting communication compression settings to improve performance” [SQL Anywhere Server -
Database Administration]
● “Compress connection parameter [COMP]” [SQL Anywhere Server - Database Administration]
● “CompressionThreshold connection parameter [COMPTH]” [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 SQL Anywhere determines that the cost of using one is less than 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 in some cases 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 using 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 [database]” [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.
CHAPTER 6

Application profiling tutorials

Contents

Setting up for the application profiling tutorials .......................................................... 264
Tutorial: Diagnosing deadlocks ................................................................................. 265
Tutorial: Diagnosing slow statements ........................................................................ 270
Tutorial: Diagnosing index fragmentation ................................................................. 275
Tutorial: Diagnosing table fragmentation ................................................................. 279
Tutorial: Baselining with procedure profiling ......................................................... 284
Setting up for the 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 following tutorials use the test database app_profiling.db which you create, and not the sample database (demo.db). Do not use the sample database to complete the tutorials.

You must have PROFILE authority to perform application profiling. For the application profiling tutorials, connect as a user with DBA authority.

You need to perform the following procedure before trying the application profiling tutorials. The procedure creates a test database app_profiling.db using data from the sample database. The test database is used in all application profiling tutorials.

To create a testing database

1. Create the directory C:\AppProfilingTutorial.
2. Open a command prompt and type the following command to create the test database app_profiling.db. The samples-dir is the location of your Samples directory:

   dbunload -c "UID=DBA;PWD=sql;DBF=samples-dir\demo.db" -an C:\AppProfilingTutorial\app_profiling.db

   For example, on a computer with Microsoft Windows XP and SQL Anywhere installed in the default locations, the command would be:

   dbunload -c "UID=DBA;PWD=sql;DBF=C:\Documents and Settings\All Users\Documents\SQL Anywhere 11\Samples\demo.db" -an C:\AppProfilingTutorial\app_profiling.db

   For more information about the location of samples-dir, see “Samples directory” [SQL Anywhere Server - Database Administration].

Tip
In the application profiling tutorials, tracing information is stored in the same database that you are profiling (app_profiling.db). However, if you profile a database that experiences heavy loads, you should consider storing tracing data in a separate database to avoid impacting performance on the production database.

See also
- “Application Profiling Wizard” on page 189
- “Advanced application profiling using diagnostic tracing” on page 199
- “PROFILE authority” [SQL Anywhere Server - Database Administration]
Tutorial: Diagnosing deadlocks

Use 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.

**Note**

This tutorial assumes you have followed the setup steps for the application profiling tutorials. If you have not, see “Setting up for the application profiling tutorials” on page 264.

**Introduction**

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.

**Preparation**

**Tip**

You can copy and paste the SQL statements in this tutorial into Interactive SQL.

**To create a deadlock**

1. Start Sybase Central and connect to the test database *app_profiling.db* with the user ID DBA and the password sql.
   
   If you have not created the test database, see “Setting up for the application profiling tutorials” on page 264.
   
   If you are unfamiliar with starting Sybase Central and connecting to a database, see “Connecting to a database on your own computer from Sybase Central or Interactive SQL” [SQL Anywhere Server - Database Administration].

2. 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.

3. In Interactive SQL, run the following SQL statements:

   a. Create two tables

      ```sql
      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. Insert values into each table
INSERT INTO "deadlock1"("val") VALUES('x');
INSERT INTO "deadlock2"("val") VALUES('x');

c. Create two procedures which you later use 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. Commit the changes you made to the database

COMMIT;

4. Exit Interactive SQL.

Profiling the problem
The Database Tracing Wizard can be used to create a diagnostic tracing session. The tracing session captures deadlock data.

To use the Database Tracing Wizard to create a diagnostic tracing session

1. In Sybase Central, choose Mode » Application Profiling.
   
   If the Application Profiling Wizard appears, click Cancel.

2. Start the Database Tracing Wizard.
   a. In the left pane click app_profiling - DBA, and choose File » Tracing.
      
      The Database Tracing Wizard appears.
   b. On the Welcome page, click Next.
   c. On the Tracing Detail Level page, select High Detail (Recommended For Short-Term, Intensive Monitoring) and 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.
   h. Click Finish.

3. Create the deadlock.
   a. In the left pane of Sybase Central, select the app_profiling - DBA database and choose File » Open Interactive SQL.
      
      Interactive SQL starts and connects to the app_profiling - DBA database.
   b. Repeat the previous step to open a second Interactive SQL window.
c. In one Interactive SQL window, run the following SQL statement:

```sql
CALL "DBA"."proc_deadlock1"();
```

d. In the second Interactive SQL window, run the following SQL statement within 20 seconds:

```sql
CALL "DBA"."proc_deadlock2"();
```

After a few moments, the ISQL Error window appears indicating that a deadlock has been detected. This is because `proc_deadlock1` requires access to the deadlock2 table, which is locked by `proc_deadlock2`. And `proc_deadlock2` requires access to the deadlock1 table, which is locked by `proc_deadlock1`.

e. Click OK.

4. Close both Interactive SQL windows.

5. To stop the tracing session, in Sybase Central select the `app_profiling - DBA` database and choose **File** » **Tracing** » **Stop Tracing With Save**.

### Analyzing the 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 view a graphical representation of the deadlock

1. Open the analysis file created during the tracing session.
   
   a. In Sybase Central, choose **Application Profiling** » **Open Analysis File Or Connect To A Tracing Database**.
      
      The **Open Analysis Or Connect To Tracing Database** window appears.
   
   b. Select **In A Tracing Database**.
   
   c. Click **Open**.
   
   d. Click the **Identification** tab, and type `DBA` in the **User ID** field and `sql` in the **Password** field.
   
   e. Click the **Database** tab, and in the **Database File** field, browse to and select `app_profiling - DBA`.
   
   f. Click **OK**.

2. View the graphical representation of the deadlock.
   
   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.

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.
View deadlock data

You may want to view more deadlock data, such as how often they occur, and how long they last. Use the Connection Blocks tab to view a list of all deadlocks recorded during the database tracing session.

To view deadlock data

1. In the Application Profiling Details pane, click the Database Tracing Data tab.
2. Click the Connection Blocks tab, just above the Database Tracing Data tab.

The Connection Blocks tab appears, displaying block time, unblock time, and duration of each blocked connection.

See also

- “Transaction blocking and deadlock” on page 138
- “Choosing a diagnostic tracing level” on page 201
- “Deadlock” on page 138
- “Advanced application profiling using diagnostic tracing” on page 199
Tutorial: Diagnosing slow statements

Use 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.

Introduction

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, a small cache size, and so on. A statement may also run slowly because it is not well formed, or does 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.

Note
This tutorial assumes you have followed the setup steps for the application profiling tutorials. If you have not, see “Setting up for the application profiling tutorials” on page 264.

Profiling the problem

The Database Tracing Wizard is used to create a diagnostic tracing session. The tracing session captures processing statement data which includes duration times.

Tip
You can copy and paste the SQL statements in this tutorial into Interactive SQL.

To use the Database Tracing Wizard to create a tracing session

1. Start Sybase Central and connect to the test database app_profiling.db with the user ID DBA and the password sql.
   
   If you have not created the test database, see “Setting up for the application profiling tutorials” on page 264.

   If you are unfamiliar with starting Sybase Central and connecting to a database, see “Connecting to a database on your own computer from Sybase Central or Interactive SQL” [SQL Anywhere Server - Database Administration].

2. Start the Database Tracing Wizard.
   
   a. In Sybase Central, choose Mode » Application Profiling. If the Application Profiling Wizard appears, click Cancel.

   b. Choose File » Configure And Start Tracing.
      
      The Database Tracing Wizard appears.

   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**.

3. In Sybase Central, in the left pane, select the **app_profiling - DBA** database, and then choose **File » Open Interactive SQL**.

Interactive SQL starts and connects to the **app_profiling - DBA** database.

4. In Interactive SQL, run the following SQL statement.

```sql
SELECT SalesOrderItems.ID, LineID, ProductID, SalesOrderItems.Quantity, ShipDate
FROM SalesOrderItems, SalesOrders
WHERE SalesOrders.CustomerID = 105 AND SalesOrderItems.ID=SalesOrders.ID;
```

5. In Interactive SQL, run the following SQL statement. This query returns the same results as the previous query, but uses an uncorrelated subquery.

```sql
SELECT *
FROM SalesOrderItems
WHERE SalesOrderItems.ID IN (SELECT SalesOrders.ID
                                FROM SalesOrders
                                WHERE SalesOrders.CustomerID = 105 );
```

6. Exit Interactive SQL.

7. In Sybase Central, select the database and then choose **File » Tracing » Stop Tracing With Save** to stop the tracing session.

**Analyzing the data**

You can identify which statements the database server spends the most time processing by using the **Summary** and **Detail** tabs, located in **Application Profiling** pane in Sybase Central.

**To view the Summary and Details tabs**

1. Open the analysis file.
   a. In Sybase Central, choose **Mode » Application Profiling**. If the **Application Profiling Wizard** appears, click **Cancel**.
   b. Choose **Application Profiling » Open Analysis File Connect To A Tracing Database**.
      The **Open Analysis Or Connect To Tracing Database** window appears.
   c. Select **In A Tracing Database**, and then click **Open**.
   d. Click the **Identification** tab, and type **DBA** in the **User ID** field, and **sql** in the **Password** field.
   e. Click the **Database** tab, and in the **Database File** field, browse to and select **app_profiling - DBA**.
   f. Click **OK**.
If the Application Profiling Details pane does not appear at the bottom of the window, choose View » Application Profiling Details.

2. Examine statement execution times of statements that were processed during the tracing session.
   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.
   b. Data for the session appears.

   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.

   Use the Total Time and Maximum Time columns to examine the execution times for the two queries you executed earlier in this tutorial. The first query shows a total time of 20 milliseconds for execution. The second query shows a faster execution time (16 milliseconds), indicating that the second query, which uses a uncorrelated subquery, may be a more efficient syntax to use.
3. To view additional information about any SQL statement in the Summary tab, right-click the statement and choose Show The Detailed SQL Statements For The Selected Summary SQL Statement.

- To view information about the connection that executed the statement, right-click the statement and choose View Connection Details For The Selected Statement.

- To view the execution plan used for the statement, right-click the statement in the Details tab, 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. Note that 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.

To view the execution plan, click the Query Information tab.
See also

For information about the **Database Tracing Wizard**, see “Advanced application profiling using diagnostic tracing” on page 199.

For more information on the items shown in the execution plan, see “Reading execution plans” on page 589.

For information about correlated and uncorrelated subqueries, see “Using subqueries” on page 485.

For information about using the **Summary** and **Details** tabs, see “Performing request trace analysis” on page 214.

See also:

- “Querying data” on page 291
- “Joins: Retrieving data from several tables” on page 373
- “Using subqueries” on page 485
Tutorial: Diagnosing index fragmentation

Use this tutorial to learn how to use the Application Profiling Wizard to determine if your database has unacceptable levels of index fragmentation.

Introduction

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.

A symptom of index fragmentation is when commonly executed queries start to take longer to perform, particularly on tables where large blocks of rows are continuously being inserted, updated, and deleted.

Note

This tutorial assumes you have followed the setup steps for the application profiling tutorials. If you have not, see “Setting up for the application profiling tutorials” on page 264.

Preparation

Tip

You can copy and paste the SQL statements in this tutorial into Interactive SQL.

To set up index fragmentation

1. Start Sybase Central and connect to the test database app_profiling.db with the user ID DBA and the password sql.

   If you have not created the test database, see “Setting up for the application profiling tutorials” on page 264.

   If you are unfamiliar with starting Sybase Central and connecting to a database, see “Connecting to a database on your own computer from Sybase Central or Interactive SQL” [SQL Anywhere Server - Database Administration].

2. In the left pane, select the app_profiling - DBA database, and then choose File » Open Interactive SQL.

   Interactive SQL starts and connects to the app_profiling - DBA database.

3. In Interactive SQL, run the following SQL statements to introduce index fragmentation. These statements can take a few minutes to complete.

   ```sql
   CREATE TABLE fragment ( id INT );
   CREATE INDEX idx_fragment ON fragment ( id );
   INSERT INTO fragment SELECT * FROM sa_rowgenerator ( 0, 100000 );
   DELETE FROM fragment WHERE MOD ( id, 2 ) = 0;
   INSERT INTO fragment SELECT * FROM sa_rowgenerator ( 0, 100000 );
   ```
4. Exit Interactive SQL.

Analyzing the data

Use this procedure to identify index fragmentation and to learn where to look for index fragmentation warnings. It is recommended that you periodically check for fragmentation warnings on your production database.

Note
The statements you executed in the previous procedure introduced index fragmentation. However, on some systems there might not be enough fragmentation to result in the warnings and recommendations described in this procedure.

To use the Application Profiling Wizard to identify index fragmentation

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 Overall Database Performance Based On The Database Schema, and then click Next.

4. On the Analysis File page, in the Save The Analysis To The Following File field, type C:\AppProfilingTutorial.

5. Click Finish.

A list of recommendations appear in the Application Profiling Details pane.
6. To view more detail, double-click the **Fragmented Indexes**. A **Recommendation** window appears containing a SQL statement you can run to resolve the index fragmentation.

**More information**

To periodically check the density of indexes for a table, run the `sa_index_density` system procedure. 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.
In Interactive SQL, run the following SQL statement to view the index fragmentation introduced to the fragment table during this tutorial:

```sql
CALL sa_index_density( 'fragment' );
```

<table>
<thead>
<tr>
<th>TableName</th>
<th>TableId</th>
<th>IndexName</th>
<th>IndexId</th>
<th>IndexType</th>
<th>LeafPages</th>
<th>Density</th>
</tr>
</thead>
<tbody>
<tr>
<td>fragment</td>
<td>736</td>
<td>idx_fragment</td>
<td>1</td>
<td>NUI</td>
<td>1,177</td>
<td>0.597509</td>
</tr>
</tbody>
</table>

Your results might be different, but the **Density** column value should be approximately 0.6.

In Interactive SQL, run the following SQL statement to improve the density of the index:

```sql
ALTER INDEX idx_fragment ON fragment REBUILD;
```

**See also**
- “sa_index_density system procedure” [*SQL Anywhere Server - SQL Reference*]
- “REORGANIZE TABLE statement” [*SQL Anywhere Server - SQL Reference*]
- “Rebuilding indexes” on page 90
- “Reducing index fragmentation and skew” on page 247
- “Application profiling” on page 188
Tutorial: Diagnosing table fragmentation

Use this tutorial to learn how to use the Application Profiling Wizard to determine if your database has unacceptable levels of table fragmentation.

Introduction

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.

Note

This tutorial assumes you have followed the setup steps for the application profiling tutorials. If you have not, see “Setting up for the application profiling tutorials” on page 264.

Preparation

Tip

You can copy and paste the SQL statements in this tutorial into Interactive SQL.

To set up table fragmentation

1. Start Sybase Central and connect to the test database app_profiling.db with the user ID DBA and the password sql.

   If you have not created the test database, see “Setting up for the application profiling tutorials” on page 264.

   If you are unfamiliar with starting Sybase Central and connecting to a database, see “Connecting to a database on your own computer from Sybase Central or Interactive SQL” [SQL Anywhere Server - Database Administration].

2. In the left pane, select the app_profiling - DBA database, and then choose File » Open Interactive SQL.

   Interactive SQL starts and connects to the app_profiling - DBA database.

3. In Interactive SQL, run the following SQL statements to introduce table fragmentation:

   a. Create the table:

      ```sql
      CREATE TABLE "DBA"."tablefrag" {
        "id" UNSIGNED BIGINT NOT NULL DEFAULT AUTOINCREMENT,
        "val1" LONG VARCHAR NULL,
        "val2" LONG VARCHAR NULL,
        "val3" LONG VARCHAR NULL,
        "val4" LONG VARCHAR NULL,
        "val5" LONG VARCHAR NULL,
        "val6" LONG VARCHAR NULL,
        "val7" LONG VARCHAR NULL,
        "val8" LONG VARCHAR NULL,
        "val9" LONG VARCHAR NULL,
      }
      ```
b. Create a procedure to insert values into the table:

```sql
CREATE PROCEDURE "DBA"."proc_tablefrag"( )
BEGIN
  DECLARE i INTEGER;
  SET i = 0;
  WHILE i < 1000
    LOOP
      INSERT INTO "DBA"."tablefrag" ( "val1" )
        VALUES('a');
      SET i = i + 1;
      END LOOP;
  END;
END;
```

c. Insert values:

```sql
CALL proc_tablefrag( );
```
d. Update the values in the table:

```sql
UPDATE "DBA"."tablefrag"
SET "val1" = 'abcdefghijklmnopqrstuvwxyz0123456789',
"val2" = 'abcdefghijklmnopqrstuvwxyz0123456789',
"val3" = 'abcdefghijklmnopqrstuvwxyz0123456789',
"val4" = 'abcdefghijklmnopqrstuvwxyz0123456789',
"val5" = 'abcdefghijklmnopqrstuvwxyz0123456789',
"val6" = 'abcdefghijklmnopqrstuvwxyz0123456789',
"val7" = 'abcdefghijklmnopqrstuvwxyz0123456789',
"val8" = 'abcdefghijklmnopqrstuvwxyz0123456789',
"val9" = 'abcdefghijklmnopqrstuvwxyz0123456789',
"val10" = 'abcdefghijklmnopqrstuvwxyz0123456789';
```
e. Commit the changes you made to the database:

```sql
COMMIT;
```

4. Exit Interactive SQL.

**Analyzing the data**

Use this procedure to identify table fragmentation and how to locate table fragmentation warnings. It is recommended that you periodically check for fragmentation warnings on your production database.

**Note**
The statements you executed in the previous procedure introduced table fragmentation. However, on some systems there might not be enough table fragmentation to result in the warnings and recommendations described in this procedure.

**To use the Application Profiling Wizard to identify table fragmentation**

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 **Profiling Options** page, select **Overall Database Performance Based On The Database Schema**.
3. On the **Analysis File** page, save the analysis file in the appropriate directory. For example, *C: \AppProfilingTutorial*.

4. Click **Finish**.

   A list of recommendations appear in the **Application Profiling Details** pane.
5. To view more detail, double-click **Fragmented Tables**. A **Recommendation** window appears containing a SQL statement you can run to resolve the table fragmentation.
More information

To check for table fragmentation (for example, CALL sa_table_fragmentation( 'tablefrag' );) run the sa_table_fragmentation system procedure. If the number of segments per row is greater than 1.1, then table fragmentation is present. Higher degrees of fragmentation may negatively impact performance. See “sa_table_fragmentation system procedure” [SQL Anywhere Server - SQL Reference].

The table you created in this tutorial should have a fragmentation value of approximately 1.9.

In Interactive SQL, run the following SQL statement to reduce table fragmentation:

```
REORGANIZE TABLE tablefrag;
```

See “REORGANIZE TABLE statement” [SQL Anywhere Server - SQL Reference].

See also

- “Reducing table fragmentation” on page 246
- “Application profiling” on page 188
Tutorial: Baselining with procedure profiling

Use 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.

Introduction

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 the procedure and run it after each change you make. This allows you to compare the new results to the baseline.

Note
This tutorial assumes you have followed the setup steps for the application profiling tutorials. If you have not, see “Setting up for the application profiling tutorials” on page 264.

Preparation

Tip
You can copy and paste the SQL statements in this tutorial into Interactive SQL.

To create a baseline procedure

1. Start Sybase Central and connect to the test database app_profiling - DBA with the user ID DBA and the password sql.

   If you have not created the test database, see “Setting up for the application profiling tutorials” on page 264.

   If you are unfamiliar with starting Sybase Central and connecting to a database, see “Connecting to a database on your own computer from Sybase Central or Interactive SQL” [SQL Anywhere Server - Database Administration].

2. In the left pane, select the app_profiling - DBA database, and then choose File » Open Interactive SQL.

   Interactive SQL starts and connects to the app_profiling - DBA database.

3. In Interactive SQL, run the following SQL statements:

   a. Create a table:

      ```sql
      CREATE TABLE table1 ( 
      Count INT );
      ```

   b. Create a baseline procedure:

      ```sql
      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;

4. Close Interactive SQL.

Profiling the problem

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.


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.

      The App_Profiling - DBA Database Properties window appears.

   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.

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:

   ```sql
   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.

**Analyzing the data**

**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 a execution time in green font. The INSERT statement in the updated procedure has a slower time than the INSERT statement in the baseline procedure.

**See also**
- “Analyzing procedure profiling results” on page 192
- “Procedure profiling in Application Profiling mode” on page 190
Part III. Querying and Modifying Data

This part describes how to query and modify data, including how to use joins. It includes several chapters on queries, from simple to complex, as well as material on inserting, deleting, and updating data. This part also includes an in-depth look at how to create analytical queries that return multidimensional results.
CHAPTER 7

Querying data

Contents

Query overview .................................................................................................................. 292
The select list: Specifying columns .............................................................................. 295
The FROM clause: Specifying tables ............................................................................ 303
The WHERE clause: Specifying rows ........................................................................... 305
The ORDER BY clause: Ordering results ..................................................................... 316
Summarizing data ......................................................................................................... 319
Full text search ........................................................................................................... 323
Query overview

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 chapter 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 349.

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.

```sql
SELECT GivenName, Surname
FROM Contacts
WHERE State = 'CA';
```

It is equally valid, though not as readable, to enter the statement as follows:

```sql
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 database” [SQL Anywhere Server - Database Administration]
- “Initialization utility (dbinit)” [SQL Anywhere Server - Database Administration]
- “Case sensitivity” on page 643

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.

```sql
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

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 349.

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 stands for 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:
You get exactly the same results by listing all the column names in the table in order after the SELECT keyword:

```sql
SELECT DepartmentID, DepartmentName, DepartmentHeadID
FROM Departments;
```

Like a column name, "*" can be qualified with a table name, as in the following query:

```sql
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:

```sql
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:

```sql
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:

```sql
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;
```

<table>
<thead>
<tr>
<th>DepartmentID</th>
<th>DepartmentName</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>R &amp; D</td>
</tr>
<tr>
<td>200</td>
<td>Sales</td>
</tr>
<tr>
<td>300</td>
<td>Finance</td>
</tr>
<tr>
<td>400</td>
<td>Marketing</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

```
SELECT DepartmentName, DepartmentID
FROM Departments;
```

<table>
<thead>
<tr>
<th>DepartmentName</th>
<th>DepartmentID</th>
</tr>
</thead>
<tbody>
<tr>
<td>R &amp; D</td>
<td>100</td>
</tr>
<tr>
<td>Sales</td>
<td>200</td>
</tr>
<tr>
<td>Finance</td>
<td>300</td>
</tr>
<tr>
<td>Marketing</td>
<td>400</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

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 373.
Renaming columns in query results

Query results consist of a set of columns. By default, the heading for each column is the expression supplied in the select list.

When query results are displayed, each column’s default heading is the name given to it when it was created. You can specify a different column heading, or alias, as follows:

```sql
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:

```sql
SELECT DepartmentName AS Department,
      DepartmentID AS "Identifying Number"
FROM Departments;
```

<table>
<thead>
<tr>
<th>Department</th>
<th>Identifying Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>R &amp; D</td>
<td>100</td>
</tr>
<tr>
<td>Sales</td>
<td>200</td>
</tr>
<tr>
<td>Finance</td>
<td>300</td>
</tr>
<tr>
<td>Marketing</td>
<td>400</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Using spaces and keywords in alias

The Identifying Number alias for DepartmentID is enclosed in double quotes because it is an identifier. You also use double quotes if you want to use keywords in aliases. For example, the following query is invalid without the quotation marks:

```sql
SELECT DepartmentName AS Department,
      DepartmentID AS "integer"
FROM Departments;
```

If you want to ensure compatibility with Adaptive Server Enterprise, you should use quoted aliases of 30 bytes or less.

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:

```sql
SELECT 'The department''s name is' AS "Prefix",
      DepartmentName AS Department
FROM Departments;
```
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;
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Quantity</th>
<th>UnitPrice</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tee Shirt</td>
<td>28</td>
<td>9</td>
</tr>
<tr>
<td>Tee Shirt</td>
<td>54</td>
<td>14</td>
</tr>
<tr>
<td>Tee Shirt</td>
<td>75</td>
<td>14</td>
</tr>
<tr>
<td>Baseball Cap</td>
<td>112</td>
<td>9</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

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;
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Sell before reorder</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tee Shirt</td>
<td>18</td>
</tr>
<tr>
<td>Tee Shirt</td>
<td>44</td>
</tr>
</tbody>
</table>
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;
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Inventory value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tee Shirt</td>
<td>252.00</td>
</tr>
<tr>
<td>Tee Shirt</td>
<td>756.00</td>
</tr>
<tr>
<td>Tee Shirt</td>
<td>1050.00</td>
</tr>
<tr>
<td>Baseball Cap</td>
<td>1008.00</td>
</tr>
</tbody>
</table>

**Arithmetic operator precedence**

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;
```

See also “Operator precedence” [SQL Anywhere Server - SQL Reference].

**String operations**

You can concatenate strings using a string concatenation operator. You can use either || (SQL/2003 compliant) 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;

<table>
<thead>
<tr>
<th>EmployeeID</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>102</td>
<td>Fran Whitney</td>
</tr>
<tr>
<td>105</td>
<td>Matthew Cobb</td>
</tr>
<tr>
<td>129</td>
<td>Philip Chin</td>
</tr>
<tr>
<td>148</td>
<td>Julie Jordan</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

**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.

- **Other operators are available**  
  The multiplication operator can be used to combine columns. You can use other operators, including the standard arithmetic operators as well as logical operators and string operators.

For example, the following query lists the full names of all customers:

```sql
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:

```sql
SELECT ID, UCASE( Name )
FROM Products;
```

<table>
<thead>
<tr>
<th>ID</th>
<th>UCASE(Products.name)</th>
</tr>
</thead>
<tbody>
<tr>
<td>300</td>
<td>TEE SHIRT</td>
</tr>
<tr>
<td>301</td>
<td>TEE SHIRT</td>
</tr>
<tr>
<td>302</td>
<td>TEE SHIRT</td>
</tr>
</tbody>
</table>
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 373.

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 of the query in which they are defined.

The following query uses a derived table (my_drv_tbl) 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_drv_tbl.max_sal AS Salary,
       my_drv_tbl.DepartmentID
FROM Employees e,
       SELECT select-list
       FROM owner.table-name;
```
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:

```sql
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.

```sql
SELECT *
FROM ShowCustomerProducts( 149 );
```
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** (\(=, \langle, \rangle\), and so on) For example, you can list all employees earning more than $50,000:

  ```sql
  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:

  ```sql
  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:

  ```sql
  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):

  ```sql
  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:

  ```sql
  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.

  ```sql
  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 to be ignored or not for the purposes of comparison.

  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:

```sql
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 (!>).

```sql
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 are searched for as well as the values they bracket.

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;
```

<table>
<thead>
<tr>
<th>Name</th>
<th>UnitPrice</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tee Shirt</td>
<td>14</td>
</tr>
<tr>
<td>Tee Shirt</td>
<td>14</td>
</tr>
<tr>
<td>Baseball Cap</td>
<td>10</td>
</tr>
<tr>
<td>Shorts</td>
<td>15</td>
</tr>
</tbody>
</table>

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;
```

<table>
<thead>
<tr>
<th>Name</th>
<th>UnitPrice</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tee Shirt</td>
<td>9</td>
</tr>
<tr>
<td>Baseball Cap</td>
<td>9</td>
</tr>
<tr>
<td>Visor</td>
<td>7</td>
</tr>
<tr>
<td>Visor</td>
<td>7</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
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:

```sql
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:

```sql
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:

<table>
<thead>
<tr>
<th>Symbols</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>%</td>
<td>Matches any string of 0 or more characters</td>
</tr>
<tr>
<td>_</td>
<td>Matches any one character</td>
</tr>
<tr>
<td>[specifier]</td>
<td>The specifier in the brackets may take the following forms:</td>
</tr>
<tr>
<td>● <strong>Range</strong></td>
<td>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</td>
</tr>
<tr>
<td>● <strong>Set</strong></td>
<td>A set can be comprised of any discrete set of values, in any order. For example, [a2bR].</td>
</tr>
</tbody>
</table>

Note that the range [a-f], and the sets [abcdef] and [fcbdae] return the same set of values.
Symbols | Meaning
--- | ---
[^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%';
```

<table>
<thead>
<tr>
<th>Match expression</th>
<th>Description</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>'Mc%'</td>
<td>Search for every name that begins with the letters Mc</td>
<td>McEvoy</td>
</tr>
<tr>
<td>'%er'</td>
<td>Search for every name that ends with er</td>
<td>Brier, Miller, Weaver, Rayner</td>
</tr>
<tr>
<td>'%en%'</td>
<td>Search for every name containing the letters en.</td>
<td>Pettengill, Lencki, Cohen</td>
</tr>
<tr>
<td>'_ish'</td>
<td>Search for every four-letter name ending in ish.</td>
<td>Fish</td>
</tr>
<tr>
<td>'Br[iy][ae]r'</td>
<td>Search for Brier, Bryer, Briar, or Bryar.</td>
<td>Brier</td>
</tr>
<tr>
<td>'[M-Z]owell'</td>
<td>Search for all names ending with owell that begin with a single letter in the range M to Z.</td>
<td>Powell</td>
</tr>
<tr>
<td>'M[^c]%'</td>
<td>Search for all names beginning with M' that do not have c as the second letter</td>
<td>Moore, Mulley, Miller, Masalsky</td>
</tr>
</tbody>
</table>

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 and time fields as well as on character data. When you use LIKE with date and time values, the dates are converted to the standard DATETIME format, and then to VARCHAR.

One feature of using LIKE when searching for DATETIME values is that, since date and time entries may contain a variety of date parts, an equality test has to be written carefully to succeed.

For example, if you insert the value 9:20 and the current date into a column named arrival_time, the following clause fails to find the value, because the entry holds the date as well as the time:

```
WHERE arrival_time = '9:20'
```

However, the clause below would find the 9:20 value:

```
WHERE arrival_time LIKE '%09:20%'
```

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 [compatibility]” [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;
```

<table>
<thead>
<tr>
<th>DepartmentID</th>
<th>DepartmentName</th>
<th>DepartmentHeadID</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>R &amp; D</td>
<td>501</td>
</tr>
<tr>
<td>200</td>
<td>Sales</td>
<td>904</td>
</tr>
<tr>
<td>300</td>
<td>Finance</td>
<td>1293</td>
</tr>
<tr>
<td>400</td>
<td>Marketing</td>
<td>1576</td>
</tr>
<tr>
<td>500</td>
<td>Shipping</td>
<td>703</td>
</tr>
<tr>
<td>201</td>
<td>Eastern Sales</td>
<td>(null)</td>
</tr>
<tr>
<td>202</td>
<td>Western Sales</td>
<td>(null)</td>
</tr>
</tbody>
</table>

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.

  ```sql
  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:

  ```sql
  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 of 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.

```sql
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.

```sql
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:

```sql
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:

```sql
SELECT Surname, BirthDate
FROM Employees
WHERE BirthDate < 'March 13, 1964'
ORDER BY BirthDate DESC;
```

<table>
<thead>
<tr>
<th>Surname</th>
<th>BirthDate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ahmed</td>
<td>1963-12-12</td>
</tr>
<tr>
<td>Dill</td>
<td>1963-07-19</td>
</tr>
<tr>
<td>Rebeiro</td>
<td>1963-04-12</td>
</tr>
<tr>
<td>Garcia</td>
<td>1963-01-23</td>
</tr>
</tbody>
</table>
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 [compatibility]" [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:

  ```sql
  SELECT Surname, GivenName
  FROM Employees
  WHERE SOUNDEX( Surname ) = SOUNDEX( 'Brown' );
  ```

<table>
<thead>
<tr>
<th>Surname</th>
<th>GivenName</th>
</tr>
</thead>
<tbody>
<tr>
<td>Braun</td>
<td>Jane</td>
</tr>
</tbody>
</table>

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:

  ```sql
  SELECT ID, Name, Description
  FROM Products
  ORDER BY Name;
  ```

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>400</td>
<td>Baseball Cap</td>
<td>Cotton Cap</td>
</tr>
<tr>
<td>401</td>
<td>Baseball Cap</td>
<td>Wool cap</td>
</tr>
<tr>
<td>700</td>
<td>Shorts</td>
<td>Cotton Shorts</td>
</tr>
<tr>
<td>600</td>
<td>Sweatshirt</td>
<td>Hooded Sweatshirt</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

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:

  ```sql
  SELECT ID, Quantity
  FROM Products
  ORDER BY Quantity DESC;
  ```
• **You can order by several columns**  The following query sorts first by size (alphabetically), and then by name:

```sql
SELECT ID, Name, Size
FROM Products
ORDER BY Size, Name;
```

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>600</td>
<td>Sweatshirt</td>
<td>Large</td>
</tr>
<tr>
<td>601</td>
<td>Sweatshirt</td>
<td>Large</td>
</tr>
<tr>
<td>700</td>
<td>Shorts</td>
<td>Medium</td>
</tr>
<tr>
<td>301</td>
<td>Tee Shirt</td>
<td>Medium</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

• **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

```sql
SELECT ID, Name, Size
FROM Products
ORDER BY UnitPrice;
```

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>Visor</td>
<td>One size fits all</td>
</tr>
<tr>
<td>501</td>
<td>Visor</td>
<td>One size fits all</td>
</tr>
<tr>
<td>300</td>
<td>Tee Shirt</td>
<td>Small</td>
</tr>
<tr>
<td>400</td>
<td>Baseball Cap</td>
<td>One size fits all</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

...
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 253.
Summarizing data

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.

A first look at aggregate functions

Aggregate functions return a single value for a set of rows. If there is no GROUP BY clause, an aggregate function returns a single value for all the rows that satisfy other aspects of the query.

List the number of employees in the company

- In Interactive SQL, execute the following query:

  ```sql
  SELECT COUNT( * )
  FROM Employees;
  ```

<table>
<thead>
<tr>
<th>COUNT(*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>75</td>
</tr>
</tbody>
</table>

  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:

  ```sql
  SELECT COUNT( * ), MIN( BirthDate ), MAX( BirthDate )
  FROM Employees;
  ```

<table>
<thead>
<tr>
<th>COUNT(*)</th>
<th>MIN(Employees.BirthDate)</th>
<th>MAX(Employees.BirthDate)</th>
</tr>
</thead>
<tbody>
<tr>
<td>75</td>
<td>1936-01-02</td>
<td>1973-01-18</td>
</tr>
</tbody>
</table>

  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].
### 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:

  ```sql
  SELECT SalesRepresentative, COUNT( * )
  FROM SalesOrders
  GROUP BY SalesRepresentative
  ORDER BY SalesRepresentative;
  ```

<table>
<thead>
<tr>
<th>SalesRepresentative</th>
<th>count(*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>129</td>
<td>57</td>
</tr>
<tr>
<td>195</td>
<td>50</td>
</tr>
<tr>
<td>299</td>
<td>114</td>
</tr>
<tr>
<td>467</td>
<td>56</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

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. Thus, 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 355.

#### 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.

```sql
SELECT SalesRepresentative, Surname, COUNT( * )
FROM SalesOrders KEY JOIN Employees
GROUP BY SalesRepresentative;
```

The error *Function or column reference to 'Surname' in the select list must also appear in a GROUP BY* is reported because SQL Anywhere cannot be sure that each of the result rows for an employee with a given ID all 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;

<table>
<thead>
<tr>
<th>SalesRepresentative</th>
<th>orders</th>
</tr>
</thead>
<tbody>
<tr>
<td>299</td>
<td>114</td>
</tr>
<tr>
<td>129</td>
<td>57</td>
</tr>
<tr>
<td>1142</td>
<td>57</td>
</tr>
<tr>
<td>467</td>
<td>56</td>
</tr>
</tbody>
</table>

See also “The HAVING clause: selecting groups of data” on page 360.

**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.

```sql
SELECT SalesRepresentative, COUNT( * )
FROM SalesOrders
WHERE SalesRepresentative > 1000
GROUP BY SalesRepresentative
HAVING count( * ) > 55
ORDER BY SalesRepresentative;
```

The following statement produces the same results.

```sql
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

Full text search can quickly find all instances of a term (word) in a database without having to scan table rows and without having to know which column a term is stored in. Full text search works by using text indexes, which you must create first. A text index stores complete positional information for every instance of every term in every indexed column. Using a text index to find rows that contain a term can be faster than scanning every row in the table for the same reasons that it can be faster to use a regular index to find rows containing a given value. See “Text indexes” on page 332.

Full text search differs from searching using predicates such as LIKE, REGEXP, and SIMILAR TO, because the matching is term-based and not pattern-based.

Like regular indexes, text indexes consume disk space. Therefore, you should be careful not to create more text indexes than are required to support your queries.

Text indexes require a refresh strategy that reflects the acceptable amount of data staleness. Frequently refreshing many text indexes, especially large ones, can impact performance. See “Creating text indexes” on page 333.

Text indexes are created using settings stored in a text configuration object. Text configuration objects allow you to build text indexes that fit the type of text searching you anticipate. See “Text configuration objects” on page 339.

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].

String comparisons in full text search use all of 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 “SQL Anywhere international features” [SQL Anywhere Server - Database Administration].

Performing a full text query

You can perform a full text query by using the CONTAINS clause in the FROM clause of a SELECT statement, or by using the CONTAINS search condition in a WHERE clause. Both return the same rows. However, the CONTAINS procedure also provides scores for the matching rows. Columns specified in a CONTAINS clause must be part of a text index. See “Text indexes” on page 332.

For example, the following two statements query the Description column in the Products table, and return the rows where the value in the Description column contains the term cotton. The second statement also returns scores for the matching rows.

```
SELECT *
FROM Products
WHERE CONTAINS ( Description, 'cotton' );

SELECT *
FROM Products CONTAINS ( Description, 'cotton' );
```
See “FROM clause” [SQL Anywhere Server - SQL Reference], and “CONTAINS search condition” [SQL Anywhere Server - SQL Reference].

Reference links in the documentation

<table>
<thead>
<tr>
<th>Topic</th>
<th>See</th>
</tr>
</thead>
</table>
| How to use full text search in a query | - For more information about how to use full text search as part of a search condition, see “CONTAINS search condition” [SQL Anywhere Server - SQL Reference].  
- To return scores for matching columns and rows as part of a full text search, see the CONTAINS clause in “FROM clause” [SQL Anywhere Server - SQL Reference]. |
| Managing text configuration objects | - “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] |
| Managing text indexes | - “CREATE TEXT INDEX statement” [SQL Anywhere Server - SQL Reference]  
- “ALTER TEXT INDEX statement” [SQL Anywhere Server - SQL Reference]  
- “DROP TEXT INDEX statement” [SQL Anywhere Server - SQL Reference]  
- “REFRESH TEXT INDEX statement” [SQL Anywhere Server - SQL Reference]  
- “TRUNCATE TEXT INDEX statement” [SQL Anywhere Server - SQL Reference] |
| Related system procedures | - To return all terms in a CHAR string, including their position in the string, see “sa_char_terms system procedure” [SQL Anywhere Server - SQL Reference].  
- To return all terms in an NCHAR string, including their position in the string, see “sa_nchar_terms system procedure” [SQL Anywhere Server - SQL Reference].  
- To refresh all text indexes that are not up to date, see “sa_refresh_text_indexes system procedure” [SQL Anywhere Server - SQL Reference].  
- To review statistical information for all text indexes, see “sa_text_index_stats system procedure” [SQL Anywhere Server - SQL Reference].  
- To view a list of all terms that are present in a text index, see “sa_text_index_vocab system procedure” [SQL Anywhere Server - SQL Reference]. |
| Related system views | - To view all text configuration objects, see “SYSTEXTCONFIG system view” [SQL Anywhere Server - SQL Reference].  
- To view all text indexes, see “SYSTEXTIDX system view” [SQL Anywhere Server - SQL Reference]. |
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...). Columns specified in a CONTAINS clause must be part of a text index. See “Text indexes” on page 332.

You cannot search for the words and characters that are considered keywords and special characters of the CONTAINS clause grammar. For example, you cannot search for the word "and", or the character "|". For the list of keywords and special characters for the CONTAINS clause, see the Remarks section of “CONTAINS search condition” [SQL Anywhere Server - SQL Reference].

Term and phrase searching

Term searching

When performing a full text search for a term, the database server uses a text index to find rows that contain the term. Columns specified in a CONTAINS clause must be part of a text index. See “Text indexes” on page 332.

The following statement queries the Description column in the Products table and returns the rows where the value in the Description column contains the term cotton.

```
SELECT Name, Description
FROM Products
WHERE CONTAINS ( Description, 'cotton' );
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseball Cap</td>
<td>Cotton Cap</td>
</tr>
<tr>
<td>Shorts</td>
<td>Cotton Shorts</td>
</tr>
</tbody>
</table>

The following example queries the Products table and returns a single value for each row indicating whether the value in the Description column contains the term cotton.

```
SELECT IF CONTAINS ( Description, 'cotton' )
THEN 1
ELSE 0
ENDIF AS Results
FROM Products;
```

<table>
<thead>
<tr>
<th>Results</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
The next example queries the Employees table for employees who have the term **drive** in their street address, and then scores the result of each match.

```sql
SELECT *
FROM Employees CONTAINS ( Employees.Street, 'Drive' );
```

<table>
<thead>
<tr>
<th>EmployeeID</th>
<th>Street</th>
<th>score</th>
</tr>
</thead>
<tbody>
<tr>
<td>207</td>
<td>127 Hawthorne Drive</td>
<td>2.189679853731752</td>
</tr>
<tr>
<td>316</td>
<td>1423 Cricklewood Drive</td>
<td>2.0529345967217356</td>
</tr>
<tr>
<td>501</td>
<td>291 Riverdale Drive</td>
<td>2.189679853731752</td>
</tr>
<tr>
<td>529</td>
<td>1294 Minuteman Drive</td>
<td>2.142117944178755</td>
</tr>
<tr>
<td>586</td>
<td>577 Heather Hill Drive</td>
<td>2.0529345967217356</td>
</tr>
<tr>
<td>862</td>
<td>405 Belleview Drive</td>
<td>2.189679853731752</td>
</tr>
<tr>
<td>1013</td>
<td>589 West Drive</td>
<td>2.463126516439687</td>
</tr>
<tr>
<td>1483</td>
<td>951 Vista Drive</td>
<td>2.4031066471715454</td>
</tr>
</tbody>
</table>

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 331.
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 specified sequence of terms. For example, the following statement searches the Remarks column of a fictitious table AccountStatus for the phrase **has been notified**.

```sql
SELECT *
FROM AccountStatus CONTAINS ( Remarks, '"has been notified"' );
```

You cannot directly search for a term that is a keyword in the CONTAINS grammar, such as AND or FUZZY. However, you can search for such a term inside a phrase, and a single-term phrase is allowed. This provides a way to search for a single keyword interpreted as a term. For example, the statement below is acceptable even though NOT is a CONTAINS keyword:

```sql
SELECT *
FROM AccountStatus CONTAINS ( Remarks, '"not"' );
```

For a list of the CONTAINS keywords, see “CONTAINS search condition” [SQL Anywhere Server - SQL Reference].

Prefix searching

When performing a full text search for a prefix, the database server uses a text index to find all rows that contain terms matching the prefix. Columns specified in a CONTAINS clause must be part of a text index. See “Text indexes” on page 332.

For example, suppose you want to search the Employees table for employees who live on a street that starts with **haw**. You can query the Employees table as follows:

```sql
SELECT Surname, Street, City
FROM Employees
WHERE CONTAINS ( Street, 'haw*' );
```

<table>
<thead>
<tr>
<th>Surname</th>
<th>Street</th>
<th>City</th>
</tr>
</thead>
<tbody>
<tr>
<td>Francis</td>
<td>127 Hawthorne Drive</td>
<td>Scarborough</td>
</tr>
</tbody>
</table>

An employee with the street name Haw, Hawthorpe, or Hawk would also be returned. Employees with the street names Shaw or Chawdrie, are not returned.

Individual terms in a phrase can be prefixes. For example, the phrase `"12* haw*"` is acceptable.

Proximity searching

You can use the NEAR keyword to find terms that appear in close proximity to each other. For example, `term1 NEAR term2` finds instances of `term1` that appear close to `term2`. Unless otherwise specified, the database server searches for instances that appear within ten terms of each other.

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`. 
You can also use a tilde character (~) instead of the NEAR keyword. For example, 'term1 ~ term2'. However, you cannot specify a distance when using the tilde form, so the database server always searches within ten terms.

To use the CONTAINS clause, you must first create a text index. See “Text indexes” on page 332.

Example

Supposed you have a table named Applicants with a column called Schools that contains applicants' educational history. You want to search for applicants who have attended a university in Virginia. Assuming you have a text index on the Schools column, you could search as follows, specifying that the search terms should be within five terms of each other:

```
SELECT * FROM Applications CONTAINS ( Schools, 'university NEAR[5] virginia' );
```

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.

Using the AND operator in full text searches

The AND operator matches a row if it contains all of the specified search terms. You can also use an ampersand (&) for the AND operator; the two are equivalent. If multiple 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 the rows in the Products table that contain the terms cotton and cap in the Description column.

```
SELECT * 
FROM Products 
WHERE CONTAINS ( Products.Description, 'cotton AND cap' );
SELECT * 
FROM Products 
WHERE CONTAINS ( Products.Description, 'cotton & cap' );
SELECT * 
FROM Products 
WHERE CONTAINS ( Products.Description, 'cotton cap' );
```

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. 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 the rows in the Products table that contain either, or both, cotton or cap in the Description column.

```
SELECT * 
FROM Products 
WHERE CONTAINS ( Products.Description, 'cotton OR cap' );
```
SELECT *
FROM Products
WHERE CONTAINS ( Products.Description, 'cotton | cap' );

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, either statement below returns the rows in the Products table that contain cotton, but that do not contain cap, in the Description column.

SELECT *
FROM Products
WHERE CONTAINS ( Products.Description, 'cotton AND NOT cap' );
SELECT *
FROM Products
WHERE CONTAINS ( Products.Description, 'cotton - cap' );

Grouping terms and phrases
Terms and expressions can be grouped with parentheses. For example, the following statement searches the Description column for items that have cotton and cap, or that have visor.

SELECT * FROM Products
WHERE CONTAINS( Product.Description, ' ( cotton AND cap ) OR visor ' );

You can also search across multiple columns in a single query, as shown in the following two SELECT statements:

SELECT *
FROM Products
WHERE CONTAINS( Products.Description, 'cap' )
OR CONTAINS( Products.Name, 'visor' );

SELECT *
FROM Products
CONTAINS ( Products.Description, Products.Name, 'visor | cap' );

In the latter syntax, scores are assigned to the results. See “Scoring full text search results” on page 331.

Also, note that the two queries have different semantics. In the first query, a row matches if the Description column contains cap or the Name column contains visor. In the second query, a row matches if either column contains either visor or cap.

Using the FUZZY operator in full text searches
You can use the FUZZY operator followed by a string in double quotes to find an approximate match for the string. This is only allowed for text indexes configured to use the NGRAM term breaker.
The FUZZY operator is normally only useful for a full text query in the FROM clause that returns a score. This is because many strings may be approximate matches, but usually only the matches with the highest scores are close enough to be meaningful matches.

Using the FUZZY operator is equivalent to breaking the string manually into n-grams 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 specifying '500 OR mai OR ain OR str OR tre OR ree OR eet'.

**Searching for unindexed terms**

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 any of the following conditions are true:

- the term is included in the stoplist
- the term is shorter than the minimum term length
- the term is longer than the maximum term length

If you search for a term that is not in the text index, the term is dropped before the query is processed. The exact way that it is dropped depends on where the term is specified. For example, suppose in the following that 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, it is not dropped and, consequently, no rows are returned.

- If the term appears in a phrase, the phrase is allowed to match with any term at the unindexed 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**

- "Altering text configuration objects" on page 341

**Using a text index to query a view**

When querying a view, you can specify a text index on the underlying base table to improve response time. 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.

### Scoring full text search results

If you use a CONTAINS query in the FROM clause, then an additional score column is available for use in the SELECT list, ORDER BY clause, or other parts of the query. By default, the score column has the correlation name, contains. You can optionally specify a different correlation name following the closing bracket of the CONTAINS query. A higher score corresponds to a better match. Therefore, it is often useful to order by descending score to get the best matches first.

**Example 1: Searching one column**

For example, create a text index on the Products.Description column as follows:

```
CREATE TEXT INDEX scoringExample
    ON Products ( Description );
```

Query the text index as follows. Note that the results of the CONTAINS clause can optionally be followed by a correlation name used to reference the score columns from elsewhere in the SELECT statement.

```
SELECT Products.Description, ct.*
FROM Products CONTAINS ( Products.Description, 'cap OR cotton' ) ct
ORDER BY ct.score DESC;
```

<table>
<thead>
<tr>
<th>Description</th>
<th>score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cotton Cap</td>
<td>3.3664304042764206</td>
</tr>
<tr>
<td>Wool cap</td>
<td>1.8224517537856726</td>
</tr>
<tr>
<td>Cotton Shorts</td>
<td>1.5101504250714481</td>
</tr>
</tbody>
</table>

Note that **Cotton Cap** has a high score in the results because it contains both search terms.

**Example 2: Searching multiple columns**

When you perform a full text search across multiple columns, the result includes a score for the row. For example, create a text index on Products.Description and Products.Name columns as follows:

```
CREATE TEXT INDEX scoringExampleMult
    ON Products ( Description, Name );
```
Query the Description and Name columns for the terms cap or visor.

```
SELECT Products.Description, Products.Name, ct.score
FROM Products CONTAINS ( Products.Description,
Products.Name, 'cap OR visor' ) ct
ORDER BY ct.score DESC;
```

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cloth Visor</td>
<td>Visor</td>
<td>3.5635154905713042</td>
</tr>
<tr>
<td>Plastic Visor</td>
<td>Visor</td>
<td>3.4507856451176244</td>
</tr>
<tr>
<td>Wool cap</td>
<td>Baseball Cap</td>
<td>3.2340501745357333</td>
</tr>
<tr>
<td>Cotton Cap</td>
<td>Baseball Cap</td>
<td>3.090467108972918</td>
</tr>
</tbody>
</table>

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.

**Text indexes**

Before you can perform a full text search, you must create a **text index** on the column(s) you will search. A text index stores complete positional information for every instance of every term in every indexed column. When you perform a full text search, a text index is used to find matching rows. For this reason, 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 how often you want the index to be refreshed. A more recently refreshed text index returns more accurate results. However, refreshing takes time and can impede performance. As well, updates to a table can become expensive if a text index is configured to refresh each time the underlying data changes. See “Creating text indexes” on page 333.

You can specify which **text configuration object** to use when creating a 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 the default configuration object. See “Text configuration objects” on page 339.

You cannot create a text index on a materialized view, a non-materialized view, or a temporary table.

**Accessing text index information in the database**

To view information about text indexes in the database, including the text configuration objects they refer to and statistics for each index, execute the following statement:

```
CALL sa_text_index_stats( );
```

See “sa_text_index_stats system procedure” [SQL Anywhere Server - SQL Reference].
SQL statements for managing text indexes

The following SQL statements are provided to help you manage text indexes:

<table>
<thead>
<tr>
<th>Statement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;CREATE TEXT INDEX statement&quot; [SQL Anywhere Server - SQL Reference]</td>
<td>Use this statement to create a text index.</td>
</tr>
<tr>
<td>&quot;ALTER TEXT INDEX statement&quot; [SQL Anywhere Server - SQL Reference]</td>
<td>Use this statement to alter a text index.</td>
</tr>
<tr>
<td>&quot;DROP TEXT INDEX statement&quot; [SQL Anywhere Server - SQL Reference]</td>
<td>Use this statement to drop a text index.</td>
</tr>
<tr>
<td>&quot;REFRESH TEXT INDEX statement&quot; [SQL Anywhere Server - SQL Reference]</td>
<td>Use this statement to refresh a text index, or to repopulate a text index that you truncated.</td>
</tr>
<tr>
<td>&quot;TRUNCATE TEXT INDEX statement&quot; [SQL Anywhere Server - SQL Reference]</td>
<td>Use this statement to truncate the data in a text index without having to recreate the text index. This is useful if you want to alter the underlying text configuration object without having to recreate dependent text indexes.</td>
</tr>
</tbody>
</table>

Creating text indexes

When you create a text index, you can specify the text configuration object to use. Text configuration objects contain settings that affect how strings are broken into terms for indexing. If you do not specify a text configuration object, a default text configuration object is used. For more information about text configuration objects, including the defaults, see “Text configuration objects” on page 339.

When you create a text index, you must also choose a refresh type. Refreshing text indexes can impact performance, as well as the availability of the underlying tables. Choosing a refresh type involves weighing the tradeoff between performance and data staleness.
Refresh types for text indexes

There are three refresh types supported for text indexes: immediate, automatic, and manual.

- **Immediate refresh**  The text index is automatically refreshed when data in the underlying table changes. Use the IMMEDIATE REFRESH clause of the CREATE TEXT INDEX statement to specify this refresh type. This is the default behavior if the refresh type is not specified at text index creation time.

  If you update a column that requires a change to an immediate refresh text index, then the index is changed during the update statement, which may make it significantly more expensive. Therefore, IMMEDIATE REFRESH is only recommended if the data must be up-to-date at all times, or the indexed columns are relatively short.

  An IMMEDIATE REFRESH text index is populated at creation time, and an exclusive lock is held on the table during this initial refresh.

  IMMEDIATE REFRESH indexes support all isolation levels.

  If you alter a column that a text index defined as IMMEDIATE REFRESH is built on, then the text index is immediately updated.

- **Automatic refresh**  The text index is automatically refreshed using a time interval that you specify. Use the AUTO REFRESH clause of the CREATE TEXT INDEX statement or ALTER TEXT INDEX statement to specify this refresh type.

  If some data staleness is acceptable, you can define the text index as either MANUAL REFRESH, or as AUTO REFRESH and set the interval at which to refresh the text index. A query on a stale index returns matching data as of the last refresh. Rows that have been inserted, deleted, or updated since the last refresh are not returned by a query.

  You can manually refresh an AUTO REFRESH text index at any time using the REFRESH TEXT INDEX statement. See “REFRESH TEXT INDEX statement” [SQL Anywhere Server - SQL Reference].

  Refreshes for AUTO REFRESH text indexes always scan the table using isolation level 0. See “isolation_level option [compatibility]” [SQL Anywhere Server - Database Administration].

  Text indexes defined as AUTO REFRESH may be refreshed more often than the interval specified. The database server executes an internal event every minute to check for required refreshes. It refreshes an index if either the time since the last refresh is larger than the refresh interval, or 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). See “sa_text_index_stats system procedure” [SQL Anywhere Server - SQL Reference].

  An AUTO REFRESH index is not automatically refreshed (that is, it contains no data) at creation time, and is not available for use until after the first refresh. The first refresh occurs when an internal refresh event runs, usually within the first minute after the text index is created.

  Changes to indexed values are never reflected in a text index, or in queries that use the text index, until after a refresh.

  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**  The text index is refreshed only when you execute a REFRESH TEXT INDEX statement. Use the MANUAL REFRESH clause of the CREATE TEXT INDEX statement to specify this refresh type.

MANUAL REFRESH is recommended if data in the underlying table is rarely changed, if a greater degree of data staleness is acceptable, or if you want to tie refreshing to an event or condition. If you define a text index as MANUAL REFRESH, you can design your own refresh strategy. For example, you can use a procedure like the following which refreshes all MANUAL REFRESH text indexes using a refresh interval that is passed as an argument, and rules that are similar to those used for AUTO REFRESH text indexes.

```sql
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;
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 is not refreshed (populated with data) at creation time, and is not usable until it is refreshed.

You refresh a MANUAL REFRESH text index using the REFRESH TEXT INDEX statement. See “REFRESH TEXT INDEX statement” [SQL Anywhere Server - SQL Reference].

Changes to indexed values are never reflected in a text index, or in queries that use the text index, until after a refresh.

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].

---

**Create a text index**

**To create a text index (Sybase Central)**

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. In the left pane, click Tables.
3. Double-click the table on which you want to create a text index.
4. Click the Text Indexes tab.
5. Choose File » New » Text Index.
   The Create Index Wizard appears.
6. Follow the instructions in the wizard.
   The new text index appears on the Text Indexes tab. It also appears in the Text Indexes folder.

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.
   For example, the following statement creates a text index called txt_index_manual on the Description column of the Products table in the demo database:
   
   ```sql
   CREATE TEXT INDEX txt_index_manual ON Products ( Description )
   MANUAL REFRESH;
   ```
   
   Since the text index is defined as MANUAL REFRESH, you need to refresh it using the REFRESH TEXT INDEX statement.

See also
- “Text indexes” on page 332
- “Full text search” on page 323
- “CREATE 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]
- “DROP TEXT INDEX statement” [SQL Anywhere Server - SQL Reference]
- “TRUNCATE TEXT INDEX statement” [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]

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. 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.
The **Refresh Data** window appears.

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 `txt_index_manual` on the `Description` column of the `Products` table in the demo database:

   ```sql
   REFRESH TEXT INDEX txt_index_manual ON Products;
   ```

**See also**

- “Text indexes” on page 332
- “Full text search” on page 323
- “CREATE 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]
- “DROP TEXT INDEX statement” [SQL Anywhere Server - SQL Reference]
- “TRUNCATE TEXT INDEX statement” [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]

**Altering text indexes**

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 remember to truncate the text index prior to, and refresh the text index after, altering the text configuration object.

  See:

  - “Altering text configuration objects” on page 341
  - “TRUNCATE TEXT INDEX statement” [SQL Anywhere Server - SQL Reference]
Note
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.

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. 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.
   
   For example, the following statement changes the txt_index_manual text index to refresh every 6 hours.

   ```sql
   ALTER TEXT INDEX txt_index_manual ON Products AUTO REFRESH EVERY 6 HOURS;
   ```

To rename a text index (Sybase Central)

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. 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.
   
   For example, the following statement renames the txt_index_manual text index to new_txt_index.
ALTER TEXT INDEX txt_index_manual ON Products
  RENAME AS new_txt_index;

See also
- “Text indexes” on page 332
- “Full text search” on page 323
- “CREATE 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]
- “DROP TEXT INDEX statement” [SQL Anywhere Server - SQL Reference]
- “TRUNCATE TEXT INDEX statement” [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]

Text configuration objects

A text configuration object is a set of configuration settings that control characteristics of text index data such as what terms to ignore, and the minimum and maximum length of terms to include in the index. Each text configuration object is stored as a row in the ISYSTEXTCONFIG system table.

When you create a text index, you can specify the text configuration object to use for building and refreshing the text index. If you do not specify a text configuration object when creating a text index, the database server chooses the applicable default text configuration object to use.

Two default text configuration objects, default_char and default_nchar, are automatically created in the database the first time a text index or text configuration object is created. default_nchar is used for creating text indexes when at least one column in the text index is an NCHAR column. Otherwise, default_char is the default.

When the database server builds a text index, non-text columns are automatically converted to strings in the CHAR database character set if default_char, or any CHAR text configuration object, is specified. Similarly, non-text columns are automatically converted to strings in the NCHAR database character set if default_nchar, or any NCHAR text configuration object, is specified.

Accessing text configuration object information in the database

To see a list of all configuration objects and their settings, execute the following statement:

```
SELECT * FROM SYSTEXTCONFIG;
```

See “SYSTEXTCONFIG system view” [SQL Anywhere Server - SQL Reference].

To see a list of text indexes and the text configuration objects they refer to, execute the following statement:

```
CALL sa_text_index_stats( );
```

See “sa_text_index_stats system procedure” [SQL Anywhere Server - SQL Reference].
### SQL statements for managing text configuration objects

Use the following statements to manage text configuration objects:

<table>
<thead>
<tr>
<th>Statement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>“CREATE TEXT CONFIGURATION statement”</strong> <a href="#">SQL Anywhere Server - SQL Reference</a></td>
<td>Use this statement to create a text configuration object. You create text configuration objects by copying an existing one, and then altering the settings as desired.</td>
</tr>
<tr>
<td><strong>“ALTER TEXT CONFIGURATION statement”</strong> <a href="#">SQL Anywhere Server - SQL Reference</a></td>
<td>Use this statement to alter the settings for a text configuration object. For example, use this statement to specify the terms to ignore when building a text index.</td>
</tr>
<tr>
<td><strong>“DROP TEXT CONFIGURATION statement”</strong> <a href="#">SQL Anywhere Server - SQL Reference</a></td>
<td>Use this statement to drop a text configuration object.</td>
</tr>
</tbody>
</table>
Creating text configuration objects

You create a text configuration object using another text configuration object as a template. Then, you alter the configuration settings as needed. Since you change settings after creating the text configuration object, the description of the settings you can change is included in the section on altering text configuration objects. See “Altering text configuration objects” on page 341.

Create a text configuration object

To create a text configuration object (Sybase Central)

1. 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 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:

```sql
CREATE TEXT CONFIGURATION myTxtConfig FROM default_char;
```

See also
- “Text configuration objects” on page 339
- “Full text search” on page 323
- “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]
- “CREATE 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]
- “DROP TEXT INDEX statement” [SQL Anywhere Server - SQL Reference]
- “TRUNCATE TEXT INDEX statement” [SQL Anywhere Server - SQL Reference]

Altering text configuration objects

Each text configuration object contains the following settings. You can alter these in Sybase Central, or by executing the ALTER TEXT CONFIGURATION statement.
### Setting | Description
--- | ---
**Stoplist** | The stoplist setting specifies the terms to ignore when building a text index. If a stoplist term is specified in a full text query, then the query returns an error.

A newly created text configuration object has the same stoplist as the text configuration object that was used as the template (including if the stoplist is empty).

To alter the stoplist, use the ALTER STOPLIST clause of the ALTER TEXT CONFIGURATION statement. For example:

```sql
ALTER TEXT CONFIGURATION config-name
STOPLIST string-expression;
```

To drop a stoplist, use the DROP STOPLIST clause of the ALTER TEXT CONFIGURATION statement. For example:

```sql
ALTER TEXT CONFIGURATION config-name
DROP STOPLIST;
```

Dropping a stoplist removes all terms from the stoplist, leaving it empty.

By default the stoplist is empty for both the default_char and the default_nchar configuration.

The Samples directory contains sample SQL code that loads stoplists for several languages. For the location of the Samples directory, see “Samples directory” [SQL Anywhere Server - Database Administration].

**Minimum term length** | The minimum term length setting specifies the minimum length, in characters, of terms allowed in the text index. Terms that are shorter than this setting are ignored when building or refreshing the text index.

If you perform a full text search for a term that is shorter than the minimum term length, the term is not found because it does not exist in the text index.

To set the minimum term length, use the MINIMUM TERM LENGTH clause of the ALTER TEXT CONFIGURATION statement. For example:

```sql
ALTER TEXT CONFIGURATION config-name
MINIMUM TERM LENGTH 4;
```

The value of this option must be greater than 0. If you set it higher than the maximum term length, the maximum term length is automatically adjusted to be equal to minimum term length. The default for this setting in the default_char and default_nchar text configuration objects is 1.
<table>
<thead>
<tr>
<th>Setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum term length</td>
<td>The maximum term length setting specifies the maximum length, in characters, of terms allowed in the text index. Terms that are longer than this setting are ignored when building or refreshing the text index.</td>
</tr>
<tr>
<td></td>
<td>If you perform a full text search for a term that is longer than the maximum term length, the term is not found because it does not exist in the text index.</td>
</tr>
<tr>
<td></td>
<td>To set the maximum term length, use the MAXIMUM TERM LENGTH clause of the ALTER TEXT CONFIGURATION statement. For example:</td>
</tr>
<tr>
<td></td>
<td>ALTER TEXT CONFIGURATION config-name MAXIMUM TERM LENGTH 20;</td>
</tr>
<tr>
<td></td>
<td>The value of this option must be less than or equal to 60. If you set it lower than the minimum term length, the minimum term length is automatically adjusted to be equal to maximum term length. The default for this setting in the default_char and default_nchar text configuration objects is 20.</td>
</tr>
</tbody>
</table>
The term breaker setting specifies the name of the algorithm to use for separating column values into terms. The choices are GENERIC or NGRAM.

The GENERIC algorithm treats as a term any string of one or more alphanumerics, separated by non-alphanumerics.

The NGRAM algorithm breaks the strings into **n-grams**. An n-gram is an n-character substring of a larger string. N-grams are useful for approximate matching or for documents that do not use a whitespace to separate terms. The NGRAM algorithm still pays attention to whether characters are alphanumerics. However, it breaks terms by counting characters rather than by using non-alphanumerics as separators. For example, an n-gram where n is 3 is any sequence of 3 adjacent alphanumeric characters. The NGRAM algorithm also returns overlaps. For example, the string abc def separated into 3-grams gives the terms abc, bcd, and def.

Use the NGRAM term breaker for any text where words aren't separated by spaces, or when fuzzy searching is required. The FUZZY clause of the CONTAINS search condition is only supported if the NGRAM term breaker is in use. See “Using the FUZZY operator in full text searches” on page 329.

The MAXIMUM TERM LENGTH setting determines the length of the n-grams in characters. An appropriate choice of length depends on the language. Typical values are 4 or 5 characters for English, and 2 or 3 characters for Chinese. The MINIMUM TERM LENGTH setting is not meaningful for the NGRAM term breaker and is ignored.

To set the term breaker algorithm, use the TERM BREAKER clause of the ALTER TEXT CONFIGURATION statement. For example:

```
ALTER TEXT CONFIGURATION config-name
TERM BREAKER GENERIC;
```

The default for this setting in the default_char and default_nchar text configuration objects is GENERIC.

<table>
<thead>
<tr>
<th>Setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Term breaker</td>
<td>The term breaker setting specifies the name of the algorithm to use for separating column values into terms. The choices are GENERIC or NGRAM.</td>
</tr>
</tbody>
</table>

**See also**

- “Text indexes” on page 332
- “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]

**Alter a text configuration object**

**To alter a text configuration object (Sybase Central)**

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

**To alter a text index (SQL)**

1. Connect to the database as a user with DBA authority.
2. Execute an ALTER TEXT CONFIGURATION statement. For example, the following statement alters the minimum term length for the myTxtConfig text configuration object:

```sql
ALTER TEXT CONFIGURATION myTxtConfig
MINIMUM TERM LENGTH 4;
```

See “Altering text configuration objects” on page 341, and “ALTER TEXT CONFIGURATION statement” [SQL Anywhere Server - SQL Reference].

**Database options and text indexes**

When a text configuration object is created, the current settings for the following database options are stored for the object: date_format, time_format, timestamp_format.

These settings affect string conversions when creating and refreshing the text indexes that refer to the text configuration object.

Since the settings for these options are stored with the text configuration object, changing these database options does not affect string conversions when creating or refreshing text indexes. However, this means that if you want to change how string conversions are handled for a text index, you must:

1. Drop the text index.
2. Optionally drop the text configuration object.
3. Change the database options.
4. Create a new text configuration object.
5. Recreate the text index using the new text configuration object.

The conversion_error option must be set to ON when creating or refreshing a text index.
End-to-end example of full text searching statements

The following procedure allows you to test the statements that support the full text searching feature. The examples are simple and are designed for you to use with the demo database.

To test the full text searching statements

1. Create the text configuration object.

   The following example creates a text configuration object called myTxtConfig. Remember that you must include the FROM clause to specify the text configuration object to use as a template.

   ```sql
   CREATE TEXT CONFIGURATION myTxtConfig FROM default_char;
   ```

2. Customize the text configuration object.

   Once you create the text configuration object, you want to add a stoplist (terms to ignore when indexing) and set the maximum term length to 30. You must do this in separate ALTER TEXT CONFIGURATION statements, as follows:

   ```sql
   ALTER TEXT CONFIGURATION myTxtConfig
   STOPLIST 'because about therefore only';
   ALTER TEXT CONFIGURATION myTxtConfig
   MINIMUM TERM LENGTH 4;
   ALTER TEXT CONFIGURATION myTxtConfig
   MAXIMUM TERM LENGTH 30;
   ```

   **Note**
   The example statements above that alter the term lengths are for demonstration purposes only. Changing the default term length settings should only be done if the data in the columns used in the index justify it. That is, change the defaults only when they exclude terms that you need to search on, or when they include terms that you never want to search on.

3. Create the text index.

   Now you want to create a text index that references the myTxtConfig text configuration object. For the purposes of this example, you can build it on the Description column of the MarketingInformation table in the demo database. The data in the table is not expected to change often, so you decide to set up a refresh interval of 24 hours.

   ```sql
   CREATE TEXT INDEX myTxtIndex ON MarketingInformation ( Description )
   CONFIGURATION myTxtConfig
   AUTO REFRESH EVERY 24 HOURS;
   ```

4. Refresh the text index.
The following statement refreshes the text index you created:

```
REFRESH TEXT INDEX myTxtIndex ON MarketingInformation;
```

5. Test the text index.
   a. Query 1:

   ```sql
   SELECT ID, Description
   FROM MarketingInformation
   WHERE CONTAINS( Description, 'cotton | cap' );
   ```

<table>
<thead>
<tr>
<th>ID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>906</td>
<td>Lightweight 100% organically grown cotton construction. Shields against sun and precipitation. Metallic ions in the fibers inhibit bacterial growth, and help neutralize odor.</td>
</tr>
<tr>
<td>908</td>
<td>Lightweight 100% organically grown cotton hooded sweatshirt with taped neck seams. Comes pre-washed for softness and to lessen shrinkage.</td>
</tr>
<tr>
<td>909</td>
<td>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.</td>
</tr>
<tr>
<td>910</td>
<td>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.</td>
</tr>
</tbody>
</table>

   Note that the search is a fuzzy search so the term cap returns a row with the word cap in the Description column.

   b. Query 2:

   ```sql
   SELECT ID, Description
   FROM MarketingInformation
   WHERE CONTAINS( Description, 'cotton' )
   AND NOT CONTAINS( Description, 'visor' );
   ```
<table>
<thead>
<tr>
<th>ID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>908</td>
<td>Lightweight 100% organically grown cotton hooded sweatshirt with taped neck seams. Comes pre-washed for softness and to lessen shrinkage.</td>
</tr>
<tr>
<td>909</td>
<td>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.</td>
</tr>
<tr>
<td>910</td>
<td>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.</td>
</tr>
</tbody>
</table>

C. Query 3:

```sql
SELECT IF CONTAINS ( Description, 'cotton' )
    THEN 1
    ELSE 0
ENDIF AS Results
FROM MarketingInformation;
```

<table>
<thead>
<tr>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>
CHAPTER 8

Summarizing, grouping, and sorting query results

Contents

Summarizing query results using aggregate functions ...................................................... 350
The GROUP BY clause: Organizing query results into groups ............................................. 355
The HAVING clause: selecting groups of data ...................................................................... 360
The ORDER BY clause: sorting query results ..................................................................... 362
Performing set operations on query results with UNION, INTERSECT, and EXCEPT ........................................................ 365
Standards and compatibility ............................................................................................... 371
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.

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 360.

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/2003 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 \( \textit{SELECT MAX(S.y + R.y) FROM S} \) contains both a local column reference (S.y) and an outer column reference (R.y), which is now illegal. It can be rewritten as \( \textit{SELECT MAX(S.y) + R.y FROM S} \). 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 the following results in Adaptive Server Anywhere version 7.

```
SELECT Name,
       ( SELECT SUM( p.Quantity )
           FROM SalesOrderItems )
FROM Products p;
```

<table>
<thead>
<tr>
<th>Name</th>
<th>SUM(p.Quantity)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tee shirt</td>
<td>30,716</td>
</tr>
<tr>
<td>Tee shirt</td>
<td>59,238</td>
</tr>
</tbody>
</table>

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:

```sql
SELECT COUNT( * ), AVG( UnitPrice )
  FROM Products
  WHERE UnitPrice > 10;
```

<table>
<thead>
<tr>
<th>COUNT( * )</th>
<th>AVG(Products.UnitPrice)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>18.2</td>
</tr>
</tbody>
</table>

### 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:

```sql
SELECT COUNT( DISTINCT City )
  FROM Contacts;
```

<table>
<thead>
<tr>
<th>COUNT( DISTINCT Contacts.City)</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
</tr>
</tbody>
</table>

You can use more than one aggregate function with DISTINCT in a query. Each DISTINCT is evaluated independently. For example:

```sql
SELECT COUNT( DISTINCT GivenName ) "first names",
       COUNT( DISTINCT Surname ) "last names"
  FROM Contacts;
```

<table>
<thead>
<tr>
<th>first names</th>
<th>last names</th>
</tr>
</thead>
<tbody>
<tr>
<td>48</td>
<td>60</td>
</tr>
</tbody>
</table>

### Aggregate functions and NULL

Any NULLS in the column on which the aggregate function is operating are ignored for the purposes of 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:

```sql
SELECT COUNT( DISTINCT Name )
  FROM Products
  WHERE UnitPrice > 50;
```
<table>
<thead>
<tr>
<th>COUNT(DISTINCT Name)</th>
<th>0</th>
</tr>
</thead>
</table>

SELECT AVG(UnitPrice) 
FROM Products 
WHERE UnitPrice > 50;

<table>
<thead>
<tr>
<th>AVG(Products.UnitPrice)</th>
<th>(NULL)</th>
</tr>
</thead>
</table>
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**

A GROUP BY clause must always appear before a HAVING clause. If a WHERE clause and a GROUP BY clause are present, the WHERE clause must appear before the GROUP BY clause.

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” on page 443.

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;
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>AVG(Products.UnitPrice)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tee Shirt</td>
<td>Small</td>
<td>9</td>
</tr>
<tr>
<td>Tee Shirt</td>
<td>Medium</td>
<td>14</td>
</tr>
<tr>
<td>Tee Shirt</td>
<td>One size fits all</td>
<td>14</td>
</tr>
<tr>
<td>Baseball Cap</td>
<td>One size fits all</td>
<td>9.5</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

**Columns in GROUP BY that are not in the select list**

A Sybase extension to the SQL/92 standard that is supported by both Adaptive Server Enterprise and SQL Anywhere is to allow expressions to the GROUP BY clause that are not in the select list. For example, the following query lists the number of contacts in each city:

```
SELECT State, COUNT( ID )
FROM Contacts
GROUP BY State, City;
```
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:

```sql
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:

```sql
SELECT Name, SUM( Quantity )
FROM Products
WHERE Name LIKE '%shirt%'
GROUP BY Name
HAVING SUM( Quantity ) > 100;
```

<table>
<thead>
<tr>
<th>Name</th>
<th>SUM(Products.Quantity)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tee Shirt</td>
<td>157</td>
</tr>
</tbody>
</table>

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:

```sql
SELECT Name, AVG( UnitPrice ) AS Price
FROM Products
GROUP BY Name;
```
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;
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tee Shirt</td>
<td>12.333333</td>
</tr>
<tr>
<td>Baseball Cap</td>
<td>9.5</td>
</tr>
<tr>
<td>Visor</td>
<td>7</td>
</tr>
<tr>
<td>Sweatshirt</td>
<td>24</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

AVG(Products.UnitPrice)

13.3
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;
```

<table>
<thead>
<tr>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tee Shirt</td>
</tr>
<tr>
<td>Baseball Cap</td>
</tr>
<tr>
<td>Visor</td>
</tr>
<tr>
<td>Sweatshirt</td>
</tr>
</tbody>
</table>

For information about when you can use aggregate functions in HAVING clauses, see “Where you can use aggregate functions” on page 351.

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%';
```
More than one condition in HAVING

More than one 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.

```sql
SELECT Name
FROM Products
GROUP BY Name
HAVING COUNT( * ) > 1
AND MAX( UnitPrice ) > 10;
```

<table>
<thead>
<tr>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseball Cap</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tee Shirt</td>
</tr>
<tr>
<td>Sweatshirt</td>
</tr>
</tbody>
</table>
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;
```

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>400</td>
<td>Baseball Cap</td>
</tr>
<tr>
<td>401</td>
<td>Baseball Cap</td>
</tr>
<tr>
<td>700</td>
<td>Shorts</td>
</tr>
<tr>
<td>600</td>
<td>Sweatshirt</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

**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;
```

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>600</td>
<td>Sweatshirt</td>
<td>39</td>
</tr>
<tr>
<td>601</td>
<td>Sweatshirt</td>
<td>32</td>
</tr>
<tr>
<td>302</td>
<td>Tee Shirt</td>
<td>75</td>
</tr>
<tr>
<td>301</td>
<td>Tee Shirt</td>
<td>54</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
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 5 *
FROM Employees
ORDER BY Surname;
```
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:

```sql
SELECT Name, AVG( UnitPrice )
FROM Products
GROUP BY Name
ORDER BY AVG( UnitPrice );
```

<table>
<thead>
<tr>
<th>Name</th>
<th>AVG(Products.UnitPrice)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Visor</td>
<td>7</td>
</tr>
<tr>
<td>Baseball Cap</td>
<td>9.5</td>
</tr>
<tr>
<td>Tee Shirt</td>
<td>12.3333333333</td>
</tr>
<tr>
<td>Shorts</td>
<td>15</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
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 368
● “EXCEPT clause” [SQL Anywhere Server - SQL Reference]
● “INTERSECT clause” [SQL Anywhere Server - SQL Reference]
● “UNION clause” [SQL Anywhere Server - SQL Reference]

Combining sets with the UNION statement

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:

\[
\begin{align*}
& x \text{ UNION ALL } ( y \text{ UNION } z ) \\
& (x \text{ UNION ALL } y) \text{ UNION } z
\end{align*}
\]

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 statement 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 statement 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 statement, both EXCEPT and INTERSECT take the ALL modifier, which prevents the elimination of duplicate rows from the result set.

See also
- “EXCEPT clause” [SQL Anywhere Server - SQL Reference]
- “INTERSECT clause” [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 statement.

```
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
```
**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 particularly useful consequence 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:

```sql
    col1 INT,
    col2 CHAR(1)
```

The tables and data are set up as follows:

```sql
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.**

<table>
<thead>
<tr>
<th>col1</th>
<th>col2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
</tr>
<tr>
<td>2</td>
<td>b</td>
</tr>
<tr>
<td>3</td>
<td>(NULL)</td>
</tr>
<tr>
<td>3</td>
<td>(NULL)</td>
</tr>
<tr>
<td>4</td>
<td>(NULL)</td>
</tr>
<tr>
<td>4</td>
<td>(NULL)</td>
</tr>
</tbody>
</table>

- **Table T2**

Copyright © 2008, iAnywhere Solutions, Inc. - SQL Anywhere 11.0.0
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;
```

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;
```

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);
```

Copyright © 2008, iAnywhere Solutions, Inc. - SQL Anywhere 11.0.0
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;
```

<table>
<thead>
<tr>
<th>col1</th>
<th>col2</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>b</td>
</tr>
<tr>
<td>3</td>
<td>(NULL)</td>
</tr>
<tr>
<td>4</td>
<td>(NULL)</td>
</tr>
<tr>
<td>4</td>
<td>(NULL)</td>
</tr>
</tbody>
</table>

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;
```

<table>
<thead>
<tr>
<th>col1</th>
<th>col2</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>b</td>
</tr>
<tr>
<td>4</td>
<td>(NULL)</td>
</tr>
</tbody>
</table>
Standards and compatibility

This section describes standards and compatibility aspects of the SQL Anywhere GROUP BY clause.

GROUP BY and the SQL/2003 standard

The SQL/2003 standard for GROUP BY requires the following:

- A column used in an expression of the SELECT clause must be in the GROUP BY clause. Otherwise, the expression using that column is an aggregate function.
- A GROUP BY expression can only contain column names from the select list, but not those used only as arguments for vector aggregates.

A standard GROUP BY with vector aggregate functions produces one row with one value per group.

SQL Anywhere support an extension that allows aggregate functions in the HAVING clause, even if they do not exist in the select list or GROUP BY clause.
CHAPTER 9

Joins: Retrieving data from several tables

Contents

Introduction ................................................................................................................ 374
Sample database schema ......................................................................................... 375
How joins work ...................................................................................................... 376
Explicit join conditions (the ON clause) ................................................................. 381
Cross joins ............................................................................................................. 385
Inner and outer joins ............................................................................................ 386
Specialized joins .................................................................................................... 393
Natural joins .......................................................................................................... 402
Key joins ................................................................................................................ 406
Introduction

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 407.
How joins work

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 the various tables. For example, you may want to answer the question "Who manages the Sales department?" To find the name of this person, you must identify the correct person using information from the Departments table, then look up that person's name in the Employees table.

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. All of the concepts are explored in greater detail in later sections.

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. In the case of a key join, the generated join condition is based on the foreign key relationships between the tables. In the case of a natural join, 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 381.

### 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?

```sql
SELECT DISTINCT Name, Size, SalesOrderItems.Quantity
FROM Products JOIN SalesOrderItems
ON Products.ID = SalesOrderItems.ProductID
AND Products.Quantity = SalesOrderItems.Quantity;
```

<table>
<thead>
<tr>
<th>name</th>
<th>Size</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseball Cap</td>
<td>One size fits all</td>
<td>12</td>
</tr>
<tr>
<td>Visor</td>
<td>One size fits all</td>
<td>36</td>
</tr>
</tbody>
</table>

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 386.

**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:

```sql
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 386.

For more information about how SQL Anywhere performs a key join of table expressions, see “Key joins of table expressions” on page 409.

For more information about how SQL Anywhere performs a natural join of table expressions, see “Natural joins of table expressions” on page 404.

### 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 and INSERT statements, as well as in 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 version 7. For databases created in version 7 or later, the default is Cursors. See “ansi_update_constraints option [compatibility]” [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 390
- “Duplicate correlation names in joins (star joins)” on page 394
- “Key joins” on page 406
- “Natural joins” on page 402

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.

```sql
SELECT *
FROM SalesOrders JOIN Customers
    ON SalesOrders.CustomerID = Customers.ID
JOIN SalesOrderItems
    ON SalesOrderItems.ID = SalesOrders.ID;
```

Tables that can be referenced

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:

```sql
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:

```sql
FROM (A KEY JOIN B), (C JOIN D ON A.x = C.x)
```

For more information about commas, see “Commas” on page 385.

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.

```sql
SELECT Employees.Surname, SalesOrders.ID, SalesOrders.OrderDate
FROM SalesOrders
JOIN Employees
    ON SalesOrders.SalesRepresentative = Employees.EmployeeID;
```
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:

```sql
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.

```sql
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.

```sql
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 406.

**Types of explicit join conditions**

Most join conditions are based on equality, and so are called **equijoins**. For example,

```sql
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?

```sql
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 in the case of 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 388.
• 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 “When key join is the default” on page 406.

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. Hence, 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. Thus, the keywords INNER, LEFT OUTER and RIGHT OUTER are not applicable to cross joins.

For example, the following statement joins two tables.

```sql
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 \(n \times m\) 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:

```sql
SELECT *
FROM A CROSS JOIN B CROSS JOIN C
WHERE A.x = B.y;
```

and

```sql
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 in the case of 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 409.

In the syntax of star joins, commas have a special use. For more information, see “Duplicate correlation names in joins (star joins)” on page 394.
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;
```

<table>
<thead>
<tr>
<th>GivenName</th>
<th>Surname</th>
<th>OrderDate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardy</td>
<td>Mums</td>
<td>2000-01-02</td>
</tr>
<tr>
<td>Aram</td>
<td>Najarian</td>
<td>2000-01-03</td>
</tr>
<tr>
<td>Tommie</td>
<td>Wooten</td>
<td>2000-01-03</td>
</tr>
<tr>
<td>Alfredo</td>
<td>Margolis</td>
<td>2000-01-06</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

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.

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.

For information about creating outer joins with Transact-SQL syntax, see “Transact-SQL outer joins (=* or =*)” on page 390.

**Example**

For example, the following statement includes all customers, whether or not they have placed an order. If a particular customer has placed no orders, each column in the result that corresponds to order information contains the NULL value.

```sql
SELECT Surname, OrderDate, City
FROM Customers LEFT OUTER JOIN SalesOrders
  ON Customers.ID = SalesOrders.CustomerID
WHERE Customers.State = 'NY'
ORDER BY OrderDate;
```

<table>
<thead>
<tr>
<th>Surname</th>
<th>OrderDate</th>
<th>City</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thompson</td>
<td>(NULL)</td>
<td>Bancroft</td>
</tr>
<tr>
<td>Reiser</td>
<td>2000-01-22</td>
<td>Rockwood</td>
</tr>
<tr>
<td>Clarke</td>
<td>2000-01-27</td>
<td>Rockwood</td>
</tr>
<tr>
<td>Mentary</td>
<td>2000-01-30</td>
<td>Rockland</td>
</tr>
</tbody>
</table>

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 of 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. In most cases, if you place restrictions on the null-supplying table in a WHERE clause, the join is 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 prior to 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. Hence, 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

```sql
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:

```sql
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 410.

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.

```sql
SELECT * 
FROM (A LEFT OUTER JOIN B ON B.x = C.x) JOIN C;
```

and

```sql
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

```sql
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;

<table>
<thead>
<tr>
<th>EmployeeID</th>
<th>Region</th>
<th>DepartmentName</th>
</tr>
</thead>
<tbody>
<tr>
<td>243</td>
<td>(NULL)</td>
<td>R &amp; D</td>
</tr>
<tr>
<td>316</td>
<td>(NULL)</td>
<td>R &amp; D</td>
</tr>
<tr>
<td>529</td>
<td>(NULL)</td>
<td>R &amp; D</td>
</tr>
<tr>
<td>902</td>
<td>Eastern</td>
<td>Sales</td>
</tr>
</tbody>
</table>

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 prior to 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 [compatibility]” [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, which is available at http://www.sybase.com/detail?id=1017447.

In the Transact-SQL dialect, you can 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 prior to 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 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 *
```
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.

```sql
SELECT a.GivenName, a.Surname,
       b.GivenName, b.Surname
FROM Employees AS a CROSS JOIN Employees AS b;
```

<table>
<thead>
<tr>
<th>GivenName</th>
<th>Surname</th>
<th>GivenName</th>
<th>Surname</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fran</td>
<td>Whitney</td>
<td>Fran</td>
<td>Whitney</td>
</tr>
<tr>
<td>Fran</td>
<td>Whitney</td>
<td>Matthew</td>
<td>Cobb</td>
</tr>
<tr>
<td>Fran</td>
<td>Whitney</td>
<td>Philip</td>
<td>Chin</td>
</tr>
<tr>
<td>Fran</td>
<td>Whitney</td>
<td>Julie</td>
<td>Jordan</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

```sql
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.
If the order of the names is not important, you can produce a list of the \((75 \times 74)/2 = 2775\) unique pairs.

```sql
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.

```sql
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.

<table>
<thead>
<tr>
<th>GivenName</th>
<th>Surname</th>
<th>GivenName</th>
<th>Surname</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alex</td>
<td>Ahmed</td>
<td>Scott</td>
<td>Evans</td>
</tr>
<tr>
<td>Joseph</td>
<td>Barker</td>
<td>Jose</td>
<td>Martinez</td>
</tr>
<tr>
<td>Irene</td>
<td>Barletta</td>
<td>Scott</td>
<td>Evans</td>
</tr>
<tr>
<td>Jeannette</td>
<td>Bertrand</td>
<td>Jose</td>
<td>Martinez</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

**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 much 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.

```
  C
 /|
B  A
 |
  D
```

**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 [database]” [SQL Anywhere Server - Database Administration].
Example 1

Create a list of the names of the customers who have 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;
```

<table>
<thead>
<tr>
<th>GivenName</th>
<th>Surname</th>
<th>OrderDate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tommie</td>
<td>Wooten</td>
<td>2000-01-03</td>
</tr>
<tr>
<td>Michael</td>
<td>Agliori</td>
<td>2000-01-08</td>
</tr>
<tr>
<td>Salton</td>
<td>Pepper</td>
<td>2000-01-17</td>
</tr>
<tr>
<td>Tommie</td>
<td>Wooten</td>
<td>2000-01-23</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 (see “Sample database schema” on page 375), you will notice 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.
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 393.

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;
```

<table>
<thead>
<tr>
<th>GivenName</th>
<th>Name</th>
<th>SUM(SalesOrderItems.Quantity)</th>
<th>GivenName</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sheng</td>
<td>Baseball Cap</td>
<td>240</td>
<td>Moira</td>
</tr>
<tr>
<td>Laura</td>
<td>Tee Shirt</td>
<td>192</td>
<td>Moira</td>
</tr>
<tr>
<td>Moe</td>
<td>Tee Shirt</td>
<td>192</td>
<td>Moira</td>
</tr>
<tr>
<td>Leilani</td>
<td>Sweatshirt</td>
<td>132</td>
<td>Moira</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

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 414.

For an explanation of natural joins of derived tables, see “Natural joins of views and derived tables” on page 405.

For an explanation of outer joins of derived tables, see “Outer joins of views and derived tables” on page 389.

## 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.

```sql
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.
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;
```

<table>
<thead>
<tr>
<th>DepartmentName</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>R &amp; D</td>
<td>Kim Lull</td>
</tr>
<tr>
<td>R &amp; D</td>
<td>David Scott</td>
</tr>
<tr>
<td>R &amp; D</td>
<td>John Sheffield</td>
</tr>
<tr>
<td>Sales</td>
<td>Moira Kelly</td>
</tr>
<tr>
<td>Finance</td>
<td>Mary Anne Shea</td>
</tr>
<tr>
<td>Marketing</td>
<td>NULL</td>
</tr>
<tr>
<td>Shipping</td>
<td>NULL</td>
</tr>
</tbody>
</table>

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)
              AS HS( Name );
```

<table>
<thead>
<tr>
<th>DepartmentName</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>R &amp; D</td>
<td>Kim Lull</td>
</tr>
<tr>
<td>R &amp; D</td>
<td>David Scott</td>
</tr>
<tr>
<td>R &amp; D</td>
<td>John Sheffield</td>
</tr>
<tr>
<td>Sales</td>
<td>Moira Kelly</td>
</tr>
<tr>
<td>Finance</td>
<td>Mary Anne Shea</td>
</tr>
</tbody>
</table>

**See also**

- “FROM clause” [SQL Anywhere Server - SQL Reference]
● “Cross joins” on page 385
● “Inner and outer joins” on page 386
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;
```

```
<table>
<thead>
<tr>
<th>GivenName</th>
<th>Surname</th>
<th>DepartmentName</th>
</tr>
</thead>
<tbody>
<tr>
<td>Janet</td>
<td>Bigelow</td>
<td>Finance</td>
</tr>
<tr>
<td>Kristen</td>
<td>Coe</td>
<td>Finance</td>
</tr>
<tr>
<td>James</td>
<td>Coleman</td>
<td>Finance</td>
</tr>
<tr>
<td>Jo Ann</td>
<td>Davidson</td>
<td>Finance</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
```

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;

<table>
<thead>
<tr>
<th>Surname</th>
<th>DepartmentName</th>
</tr>
</thead>
<tbody>
<tr>
<td>Whitney</td>
<td>R &amp; D</td>
</tr>
<tr>
<td>Cobb</td>
<td>R &amp; D</td>
</tr>
<tr>
<td>Breault</td>
<td>R &amp; D</td>
</tr>
<tr>
<td>Shishov</td>
<td>R &amp; D</td>
</tr>
<tr>
<td>Driscoll</td>
<td>R &amp; D</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

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 of 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 happens to be 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 416.

**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 375 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 SQL_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:
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 410
- “Key joins of table expression lists” on page 411

### 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 407.

### 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. Hence, 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 407.

**Example**

The following query returns the names of all salespeople who have sold at least one order to a specific region.

```sql
SELECT Employees.Surname, Departments.DepartmentName
FROM ( Employees JOIN Departments
    ON ( Employees.DepartmentID = Departments.DepartmentID ) )
JOIN SalesOrders
    ON ( Employees.EmployeeID = SalesOrders.SalesRepresentative );
```
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
       AS FK_DepartmentID_DepartmentID )
       KEY JOIN Employees;
```

---

**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 410.

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 411.

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 410.

**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.

```sql
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 426, and “RecursiveTable algorithm (RT)” on page 582.

### Example 1

For example, in the following statement, View1 is a view.

```sql
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.)

```sql
SELECT *
FROM C CROSS JOIN D;
```
or

\[
\text{SELECT *}
\text{FROM C, D;}
\]

or

\[
\text{SELECT *}
\text{FROM C JOIN D ON (C.x = D.y);}
\]

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 409, 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:

\[
\text{SELECT *}
\text{FROM View1 KEY JOIN B;}
\]

\[
\text{SELECT *}
\text{FROM View1 JOIN B ON B.y = View1.z;}
\]

See “Key joins when there are multiple foreign key relationships” on page 407.

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 SQL_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 SQL_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 SQL_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 SQL_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 \((A_1, A_2, \ldots) \text{ KEY JOIN } (B_1, B_2, \ldots)\) where \(A_1, B_1,\) and so on are table expressions that do not contain commas.

1. For each pair of table expressions \(A_i\) and \(B_j\), find a unique generated join condition for the table expression \((A_i \text{ KEY JOIN } B_j)\) 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 \((A_1, A_2, \ldots) \text{ KEY JOIN } (B_1, B_2, \ldots)\) where \(A_1, B_1,\) and so on are table expressions that may contain commas.

1. For each pair of table expressions \(A_i\) and \(B_j\), find a unique generated join condition for the table expression \((A_i \text{ KEY JOIN } B_j)\) 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.
CHAPTER 10

Common table expressions

Contents

Introduction to common table expressions ................................................................. 420
Typical applications of common table expressions .................................................. 423
Recursive common table expressions ...................................................................... 426
Parts explosion problems ........................................................................................ 428
Data type declarations in recursive common table expressions .............................. 431
Least distance problem ............................................................................................. 432
Using multiple recursive common table expressions .............................................. 435
Introduction to 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.

Recursive common table expressions permit you to query tables that represent hierarchical information, such as reporting relationships within a company. They can also be used to solve parts explosion problems and least distance problems. See “Recursive common table expressions” on page 426.

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.

\[
\begin{align*}
&\text{SELECT DepartmentID, COUNT( } * \text{ ) AS n} \\
&\text{FROM Employees} \\
&\text{GROUP BY DepartmentID;}
\end{align*}
\]

This query can be used to extract the department with the most employees as follows:

\[
\begin{align*}
&\text{SELECT DepartmentID, n} \\
&\text{FROM ( SELECT DepartmentID, COUNT( } * \text{ ) AS n} \\
&\text{ FROM Employees GROUP BY DepartmentID ) AS a} \\
&\text{WHERE a.n =} \\
&\text{ ( SELECT MAX( n )} \\
&\text{ FROM ( SELECT DepartmentID, COUNT( } * \text{ ) AS n} \\
&\text{ FROM Employees GROUP BY DepartmentID ) AS b );}
\end{align*}
\]

While this statement provides the correct result, it has some disadvantages. The first disadvantage is that the repeated subquery makes this statement clumsy. 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.

\[
\begin{align*}
&\text{CREATE VIEW CountEmployees( DepartmentID, n ) AS} \\
&\text{ SELECT DepartmentID, COUNT( } * \text{ ) AS n} \\
&\text{ FROM Employees GROUP BY DepartmentID;}
\end{align*}
\]

\[
\begin{align*}
&\text{SELECT DepartmentID, n} \\
&\text{FROM CountEmployees} \\
&\text{WHERE n = ( SELECT MAX( n )} \\
&\text{ FROM CountEmployees );}
\end{align*}
\]

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, in cases where the view is used only once within a particular SELECT statement, the preferred method is to instead use a common table expression.
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.

```sql
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.

```sql
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.

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.

```sql
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;
```

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.

```sql
WITH
    CountEmployees( DepartmentID, n ) AS
    ( SELECT DepartmentID, COUNT( * ) AS n
      FROM Employees GROUP BY DepartmentID ),
    DeptPayroll( DepartmentID, amt ) AS
    ( SELECT DepartmentID, SUM( Salary ) AS amt
      FROM Employees GROUP BY DepartmentID )
SELECT count.DepartmentID, count.n, pay.amt
FROM CountEmployees AS count JOIN DeptPayroll AS pay
ON count.DepartmentID = pay.DepartmentID
```
WHERE count.n = ( SELECT MAX( n ) FROM CountEmployees )
  OR pay.amt = ( SELECT MIN( amt ) FROM DeptPayroll );

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.

```sql
WITH DeptPayroll( DepartmentID, amt ) AS
  ( SELECT DepartmentID, SUM( Salary ) AS amt
    FROM Employees GROUP BY DepartmentID )
SELECT DepartmentID, amt
FROM DeptPayroll
WHERE amt = ( SELECT MAX( amt )
             FROM DeptPayroll );
```

- **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.

```sql
CREATE VIEW LargestDept ( DepartmentID, Size, pay ) AS
  WITH CountEmployees( DepartmentID, n ) AS
    ( SELECT DepartmentID, COUNT( * ) AS n
      FROM Employees GROUP BY DepartmentID ),
  DeptPayroll( DepartmentID, amt ) AS
    ( SELECT DepartmentID, SUM( Salary ) AS amt
      FROM Employees GROUP BY DepartmentID )
SELECT count.DepartmentID, count.n, pay.amt
FROM CountEmployees count JOIN DeptPayroll pay
ON count.DepartmentID = pay.DepartmentID
WHERE count.n = ( SELECT MAX( n ) FROM CountEmployees )
  OR pay.amt = ( SELECT MAX( amt ) FROM DeptPayroll );
```

- **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.

```sql
CREATE TABLE LargestPayrolls ( DepartmentID INTEGER, Payroll NUMERIC, 
                              CurrentDate DATE );
INSERT INTO LargestPayrolls( DepartmentID, Payroll, CurrentDate )
  WITH DeptPayroll( DepartmentID, amt ) AS
    ( SELECT DepartmentID, SUM( Salary ) AS amt
      FROM Employees
      GROUP BY DepartmentID )
SELECT DepartmentID, amt, CURRENT TIMESTAMP
FROM DeptPayroll
WHERE amt = ( SELECT MAX( amt )
              FROM DeptPayroll );
```
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.

```sql
WITH DeptPayroll( DepartmentID, amt ) AS
  ( SELECT DepartmentID, SUM( Salary ) AS amt
    FROM Employees GROUP BY DepartmentID )
SELECT DepartmentID, amt
FROM DeptPayroll
WHERE amt = ( SELECT MAX( amt )
  FROM DeptPayroll )
```

For more information about aggregate functions, see “Window aggregate functions” on page 456.

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 thus permitted.

The following statement selects the gross sales of the various sales representatives in the SQL Anywhere sample database.

```sql
SELECT GivenName || ' ' || Surname AS sales_rep_name,
       SalesRepresentative AS sales_rep_id,
```
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 demonstrates how to call the above procedure.

```
CALL sales_rep_total( 129, 2000 );
```

Views that store values

Sometimes 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.

```sql
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 demonstrates how to call the above procedure.

```sql
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.

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.

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.

Recursion provides a much 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 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 prior to 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.

For more information about common table expressions, see “Introduction to common table expressions” on page 420.

Selecting hierarchical data

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, GivenName, Surname, 0
      FROM Employees AS e
      )
    UNION ALL
    ( SELECT EmployeeID, ManagerID, GivenName, Surname, mgmt_level + 1
      FROM Employees AS e, manager
      WHERE ManagerID = EmployeeID
    )
  )
SELECT givenName, surname, mgmt_level
FROM manager
ORDER BY mgmt_level;
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 in the event that 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. Thus, 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.

<table>
<thead>
<tr>
<th>component</th>
<th>subcomponent</th>
<th>quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>bookcase</td>
<td>back</td>
<td>1</td>
</tr>
<tr>
<td>bookcase</td>
<td>side</td>
<td>2</td>
</tr>
<tr>
<td>bookcase</td>
<td>shelf</td>
<td>3</td>
</tr>
<tr>
<td>bookcase</td>
<td>foot</td>
<td>4</td>
</tr>
<tr>
<td>bookcase</td>
<td>screw</td>
<td>4</td>
</tr>
<tr>
<td>back</td>
<td>backboard</td>
<td>1</td>
</tr>
<tr>
<td>back</td>
<td>screw</td>
<td>8</td>
</tr>
</tbody>
</table>
### Parts explosion problems

<table>
<thead>
<tr>
<th>component</th>
<th>subcomponent</th>
<th>quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>side</td>
<td>plank</td>
<td>1</td>
</tr>
<tr>
<td>shelf</td>
<td>plank</td>
<td>1</td>
</tr>
<tr>
<td>shelf</td>
<td>screw</td>
<td>4</td>
</tr>
</tbody>
</table>

The following statements create the bookcase table and insert the data shown in the above table.

```sql
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;
```

After you have created the bookcase table, you can recreate the table of its parts, shown above, with the following query.

```sql
SELECT * FROM bookcase
ORDER BY component, subcomponent;
```

With this table constructed, you can generate a list of the primitive parts and the quantity of each required to construct the bookcase.

```sql
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.

<table>
<thead>
<tr>
<th>subcomponent</th>
<th>quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>backboard</td>
<td>1</td>
</tr>
<tr>
<td>foot</td>
<td>4</td>
</tr>
<tr>
<td>subcomponent</td>
<td>quantity</td>
</tr>
<tr>
<td>--------------</td>
<td>----------</td>
</tr>
<tr>
<td>plank</td>
<td>5</td>
</tr>
<tr>
<td>screw</td>
<td>24</td>
</tr>
</tbody>
</table>

Alternatively, you can rewrite this query to perform an additional level of recursion, thus avoiding the need for the subquery in the main SELECT statement. The results of the following query are identical to those of the previous query.

```sql
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;
```
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 thus inherits the data types of the selected columns. See “Parts explosion problems” on page 428.

If this query is rewritten as follows, explicit casts are required.

```sql
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:

```sql
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 happen to be 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.

```sql
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, adding the length of the new segments so as 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.

It is particularly important to guarantee that recursive queries finish properly when using them to explore
cyclic graphs. 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 < ( 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.
<table>
<thead>
<tr>
<th>route</th>
<th>distance</th>
<th>segments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kitchener, Barrie, Huntsville, Pembroke</td>
<td>525</td>
<td>3</td>
</tr>
<tr>
<td>Kitchener, Toronto, Belleville, Pembroke</td>
<td>525</td>
<td>3</td>
</tr>
<tr>
<td>Kitchener, Toronto, Barrie, Huntsville, Pembroke</td>
<td>565</td>
<td>4</td>
</tr>
<tr>
<td>Kitchener, Barrie, Huntsville, North Bay, Pembroke</td>
<td>630</td>
<td>4</td>
</tr>
<tr>
<td>Kitchener, Barrie, Toronto, Belleville, Pembroke</td>
<td>665</td>
<td>4</td>
</tr>
<tr>
<td>Kitchener, Toronto, Barrie, Huntsville, North Bay, Pembroke</td>
<td>670</td>
<td>5</td>
</tr>
<tr>
<td>Kitchener, Toronto, Belleville, Ottawa, Pembroke</td>
<td>675</td>
<td>4</td>
</tr>
</tbody>
</table>
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.

```sql
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, non-recursive
  ( SELECT MIN(distance)                 -- common table expression
    FROM trip                            -- common table
    WHERE destination = 'Pembroke' )
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.

```sql
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, v.origin, t.distance + v.distance, segments + 1
      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
      AND v.origin <> 'Pembroke'
      AND segments < ( SELECT count(*)/2 FROM travel ) )
  SELECT route, distance, segments FROM trip
  WHERE destination = final
  ORDER BY distance, segments, route;
END;
```
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 demonstrates how to call the above procedure.

CALL best_routes ( 'Pembroke', 'Kitchener' );
CHAPTER 11

OLAP support

Contents

Overview of OLAP functionality ................................................................. 438
GROUP BY clause extensions ...................................................................... 440
Window functions ..................................................................................... 450
Overview of OLAP functionality

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 all of 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 440.

  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 450.

- **Window aggregate functions** Almost all of the 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 456.

- **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 474.

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 [database]” [SQL Anywhere Server - Database Administration]
- “ClusteredHashGroupBy algorithm (GrByHClust)” on page 579
- “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:

```sql
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:

```sql
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:

<table>
<thead>
<tr>
<th>City</th>
<th>State</th>
<th>CompanyName</th>
<th>Cnt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(NULL)</td>
<td>(NULL)</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>(NULL)</td>
<td>Cooper Inc.</td>
<td>1</td>
</tr>
</tbody>
</table>
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 of 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 447.

Example

The following example shows how you can tailor the results that are returned from a query using GROUPING SETS, as well as use an ORDER BY clause to better organize the results. The query returns the total number of orders by Quarter in each Year, as well as a total for each Year. Ordering by Year and then Quarter makes the results easier to understand:

```sql
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:
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 447.

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:

```sql
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:
### 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 in the case of 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:

```sql
SELECT * FROM t GROUP BY GROUPING SETS ( X, Y );
SELECT * FROM t GROUP BY GROUPING SETS( ( X ), ( Y ) );
```

### Using ROLLUP and CUBE

Using GROUPING SETS is useful when you want to concatenate a number of disparate 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 pre-defined 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 447.

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 \((X_1,X_2, \ldots , X_n)\) then the ROLLUP clause generates \( n + 1 \) grouping sets as:

\[
\{(\emptyset), (X_1), (X_1,X_2), (X_1,X_2,X_3), \ldots , (X_1,X_2,X_3, \ldots , X_n)\}
\]

Example

The following query summarizes the sales orders by year and quarter, and returns the result set shown in the table below:

```sql
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:

<table>
<thead>
<tr>
<th>Quarter</th>
<th>Year</th>
<th>Orders</th>
<th>GQ</th>
<th>GY</th>
</tr>
</thead>
<tbody>
<tr>
<td>(NULL)</td>
<td>(NULL)</td>
<td>648</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>(NULL)</td>
<td>2000</td>
<td>380</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2000</td>
<td>87</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2000</td>
<td>77</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>2000</td>
<td>91</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>2000</td>
<td>125</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>2001</td>
<td>268</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>2001</td>
<td>139</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>(NULL)</td>
<td>2001</td>
<td>268</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>2001</td>
<td>139</td>
<td>0</td>
</tr>
</tbody>
</table>
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 T-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:

\[
\{((), (X_1), (X_1, X_2), (X_1, X_2, X_3), \ldots, (X_1, X_2, X_3, \ldots, X_n),
(X_2), (X_2, X_3), (X_2, X_3, X_4), \ldots, (X_2, X_3, X_4, \ldots, X_n), \ldots, (X_n)\}
\]
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:

<table>
<thead>
<tr>
<th>Quarter</th>
<th>Year</th>
<th>Orders</th>
<th>GQ</th>
<th>GY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(NULL)</td>
<td>648</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>226</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>196</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>101</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>125</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>(NULL)</td>
<td>380</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>2000</td>
<td>87</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>2000</td>
<td>77</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>2000</td>
<td>91</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>4</td>
<td>2000</td>
<td>125</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>(NULL)</td>
<td>2001</td>
<td>268</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>2001</td>
<td>139</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>2</td>
<td>2001</td>
<td>119</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>3</td>
<td>2000</td>
<td>10</td>
<td>0</td>
</tr>
</tbody>
</table>

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 T-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 consequence, 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
JOIN SalesOrders ON Employees.EmployeeID = SalesOrders.EmployeeID
GROUP BY Employee, Year,
GROUPING( Employee ) AS GE,
GROUPING( Year ) AS GY
ORDER BY Employee, Year;
```
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:

<table>
<thead>
<tr>
<th>Employees</th>
<th>Year</th>
<th>Orders</th>
<th>GE</th>
<th>GY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (NULL)</td>
<td>(NULL)</td>
<td>54</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2 (NULL)</td>
<td>(NULL)</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3 102</td>
<td>(NULL)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4 390</td>
<td>(NULL)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5 1062</td>
<td>(NULL)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6 1090</td>
<td>(NULL)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7 1507</td>
<td>(NULL)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8 (NULL)</td>
<td>2000</td>
<td>34</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>9 667</td>
<td>2000</td>
<td>34</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10 (NULL)</td>
<td>2001</td>
<td>20</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>11 667</td>
<td>2001</td>
<td>20</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

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 of 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 movement of the window as input rows are processed is shown in the following diagram. 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.

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 of the ranking functions, and almost all of the 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, prior to their 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. The supported window aggregate functions are listed below:
  - AVG
  - COVAR_POP
  - COVAR_SAMP
  - CUME_DIST
  - DENSE_RANK
  - MAX
  - MIN
  - PERCENT_RANK
  - RANK
  - REGR_AVGX
  - REGR_AVGY
  - REGR_COUNT
  - REGR_INTERCEPT
  - REGR_R2
  - REGR_SLOPE
  - REGR_SXX
  - REGR_SXY
  - REGR_SYY
  - STDDEV
  - STDDEV_POP
  - STDDEV_SAMP
  - SUM
  - VAR_POP
  - VAR_SAMP
  - VARIANCE

  For more information about window aggregate functions, see “Window aggregate functions” on page 456.

- **Window ranking functions**  Window ranking functions return the rank of a row relative to the other rows in a partition. The supported window ranking functions are listed below:
  - CUME_DIST
  - DENSE_RANK
  - PERCENT_RANK
  - RANK

  For more information about window ranking functions, see “Window ranking functions” on page 474.

- **Row numbering functions**  Row numbering functions uniquely number the rows in a partition. In SQL Anywhere, the ROW_NUMBER function is an ANSI standard-compliant function that permits much of the same functionality as the SQL Anywhere NUMBER(*) function.

  For more information about using this function in the window context, see “ROW_NUMBER function” on page 482.
Defining a window

SQL windowing extensions allow you to configure the bounds of a window, as well as 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. Consequently, 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

One implication of the evaluation ordering is that predicates on window functions would require the window function to appear in a derived table.

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. Hence, windows provide another opportunity to perform grouping and ordering operations in addition to a query's GROUP BY and ORDER BY clauses.

When you define the window over which a window function operates, you specify one or more of the following:

- **Partitioning** Defines how the input rows are partitioned (or grouped) using the PARTITION BY clause. 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** Defines how the input rows are ordered, prior to 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 input rows are processed in whatever manner the database server considers most efficient.

- **Bounds** Defines the bounds of the window either in terms of a range of data values offset from the value in the current row (RANGE clause), or in terms of the number of rows offset from the current row (ROWS clause). The window size can be one, many, or all rows of a partition.

  Within the ROWS and RANGE clauses, you specify the start and end rows of the window, relative to the current row, using a combination of optional PRECEDING, BETWEEN, and FOLLOWING clauses. These clauses take expressions, as well as the keywords UNBOUNDED and CURRENT ROW.

  If no bounds are defined for a window, the size of the window defaults as follows:
If the window specification contains an ORDER BY clause, the window's start point is UNBOUNDED PRECEDING, and its end point is CURRENT ROW.

If the window specification does not contain an ORDER BY clause, the window's start point is UNBOUNDED PRECEDING, and the end point is UNBOUNDED FOLLOWING.

As a best practice measure, always define a window's bounds to make the window specification explicit. Do not specify window bounds when using a ranking or a row-numbering function.

Sizing the window

The current row provides the reference point for determining the start and end points of a window. The window bounds that you specify define a window relative to the current row.

You can specify the bounds as either an exact number of rows, using the ROWS clause, or as a range of values offset from the value in the current row, using the RANGE clause. In the latter case, the size of the window can vary, depending on the values in the surrounding rows.

Setting bounds by specifying the number of rows (ROWS clause)

Following are some additional examples of window bounds you might specify.

- **ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW**  This window starts at the beginning of the partition, and ends with the current row. Use this construct when computing cumulative results, such as cumulative sums.

- **ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING**  This syntax specifies a fixed window (regardless of the current row) over the entire partition. Use this construct when you want the value of an aggregate function to be identical for each row of a partition.

- **ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING**  This syntax specifies a moving window of fixed-size (three adjacent rows) that includes the current row, and the row preceding and following the current row. Use this construct when, for example, computing a three day or three month moving average. To avoid problems due to gaps in the input to the window function if the set of values is not continuous, use the RANGE clause rather than ROWS clause because a window's bounds, when based on the RANGE clause, automatically handles adjacent rows where there are gaps in the range, as well as rows with duplicate values.

  With a moving window of more than one row, NULLs exist when computing the first and last row in the partition. This is 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**  This syntax specifies a window of one row, the current row.

- **ROWS BETWEEN 1 PRECEDING AND 1 PRECEDING**  This syntax specifies a window of one row, the row preceding the current row. This construction makes it possible to easily compute deltas, or differences in value, between adjacent rows.
Setting bounds by specifying a range (RANGE clause)

When using the RANGE clause, you define the window size based on values in the column specified in the ORDER BY clause. To use the RANGE clause, you must also specify an ORDER BY clause, the ORDER BY clause must contain only a single column, and the column must be in the number domain. The window size is then determined by comparing values in the column specified in the ORDER BY clause to the value in the current row.

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.

RANGE uses unsigned integer values. Truncation of the range expression may occur depending on the domain of the ORDER BY expression and the domain of the value specified in the RANGE clause.

Defining a window inline, or using a WINDOW clause

A window definition can be placed in the OVER clause of a window function. This is referred to as defining the window using inline syntax. You can also define the window separately in a WINDOW clause, and then refer to it from the function. The benefit of using the WINDOW clause is that it allows multiple functions to reference the window, and other computations to be performed on the window, since it is defined independently.

For example, the following SELECT statement shows three different functions referencing the same named window (Qty):

```sql
SELECT Products.ID, Description, Quantity,
       RANK() OVER Qty AS Rank_quantity,
       CUME_DIST() OVER Qty AS Dist_Cume,
       ROW_NUMBER() OVER Qty AS Qty_order
FROM Products
WINDOW Qty AS ( ORDER BY Quantity ASC )
ORDER BY Qty_order;
```

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.

Inline window definition 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:

<table>
<thead>
<tr>
<th>ID</th>
<th>Description</th>
<th>Quantity</th>
<th>ShipDate</th>
<th>Cumulative_qty</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>301 V-neck</td>
<td>24</td>
<td>2001-07-16</td>
<td>24</td>
</tr>
<tr>
<td>2</td>
<td>302 Crew Neck</td>
<td>60</td>
<td>2001-07-02</td>
<td>60</td>
</tr>
<tr>
<td>3</td>
<td>302 Crew Neck</td>
<td>36</td>
<td>2001-07-13</td>
<td>96</td>
</tr>
<tr>
<td>4</td>
<td>400 Cotton Cap</td>
<td>48</td>
<td>2001-07-05</td>
<td>48</td>
</tr>
<tr>
<td>5</td>
<td>400 Cotton Cap</td>
<td>24</td>
<td>2001-07-19</td>
<td>72</td>
</tr>
<tr>
<td>6</td>
<td>401 Wool Cap</td>
<td>48</td>
<td>2001-07-09</td>
<td>48</td>
</tr>
<tr>
<td>7</td>
<td>500 Cloth Visor</td>
<td>12</td>
<td>2001-07-22</td>
<td>12</td>
</tr>
<tr>
<td>8</td>
<td>501 Plastic Visor</td>
<td>60</td>
<td>2001-07-07</td>
<td>60</td>
</tr>
<tr>
<td>9</td>
<td>501 Plastic Visor</td>
<td>12</td>
<td>2001-07-12</td>
<td>72</td>
</tr>
<tr>
<td>10</td>
<td>501 Plastic Visor</td>
<td>12</td>
<td>2001-07-22</td>
<td>84</td>
</tr>
<tr>
<td>11</td>
<td>601 Zipped Sweatshirt</td>
<td>60</td>
<td>2001-07-19</td>
<td>60</td>
</tr>
<tr>
<td>12</td>
<td>700 Cotton Shorts</td>
<td>24</td>
<td>2001-07-26</td>
<td>24</td>
</tr>
</tbody>
</table>

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 window definition example**

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.

```sql
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;
```

**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.

For the purposes of explanation in this chapter, the window aggregate functions are broken up into three categories:

- **Basic aggregate functions**  Following is the list of supported basic aggregate functions:
  - SUM
  - AVG
  - MAX
  - MIN
  - FIRST_VALUE
  - LAST_VALUE
  - COUNT

- **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

- **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
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.
- **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 450

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.

```sql
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.

<table>
<thead>
<tr>
<th>DepartmentID</th>
<th>Surname</th>
<th>StartDate</th>
<th>Salary</th>
<th>Sum_Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Whitney</td>
<td>1984-08-28</td>
<td>45700.00</td>
<td>45700.00</td>
</tr>
<tr>
<td>2</td>
<td>Cobb</td>
<td>1985-01-01</td>
<td>62000.00</td>
<td>107700.00</td>
</tr>
<tr>
<td>3</td>
<td>Shishov</td>
<td>1986-06-07</td>
<td>72995.00</td>
<td>180695.00</td>
</tr>
<tr>
<td>4</td>
<td>Driscoll</td>
<td>1986-07-01</td>
<td>48023.69</td>
<td>228718.69</td>
</tr>
<tr>
<td>5</td>
<td>Guevara</td>
<td>1986-10-14</td>
<td>42998.00</td>
<td>271716.69</td>
</tr>
<tr>
<td>6</td>
<td>Wang</td>
<td>1988-09-29</td>
<td>68400.00</td>
<td>340116.69</td>
</tr>
<tr>
<td>7</td>
<td>Soo</td>
<td>1990-07-31</td>
<td>39075.00</td>
<td>379191.69</td>
</tr>
<tr>
<td>8</td>
<td>Diaz</td>
<td>1990-08-19</td>
<td>54900.00</td>
<td>434091.69</td>
</tr>
<tr>
<td>9</td>
<td>Overbey</td>
<td>1987-02-19</td>
<td>39300.00</td>
<td>39300.00</td>
</tr>
<tr>
<td>10</td>
<td>Martel</td>
<td>1989-10-16</td>
<td>55700.00</td>
<td>95000.00</td>
</tr>
<tr>
<td>11</td>
<td>Savarino</td>
<td>1989-11-07</td>
<td>72300.00</td>
<td>167300.00</td>
</tr>
<tr>
<td>12</td>
<td>Clark</td>
<td>1990-07-21</td>
<td>45000.00</td>
<td>212300.00</td>
</tr>
<tr>
<td>13</td>
<td>Goggin</td>
<td>1990-08-05</td>
<td>37900.00</td>
<td>250200.00</td>
</tr>
</tbody>
</table>

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:

```sql
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' );
```
<table>
<thead>
<tr>
<th>EmployeeNumber</th>
<th>LastName</th>
<th>CurrentRow</th>
<th>PreviousRow</th>
<th>Delta</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 913</td>
<td>Martel</td>
<td>55700.000</td>
<td>(NULL)</td>
<td>(NULL)</td>
</tr>
<tr>
<td>2 1062</td>
<td>Blaikie</td>
<td>54900.000</td>
<td>55700.000</td>
<td>-800.000</td>
</tr>
<tr>
<td>3 249</td>
<td>Guevara</td>
<td>42998.000</td>
<td>54900.000</td>
<td>-11902.000</td>
</tr>
<tr>
<td>4 390</td>
<td>Davidson</td>
<td>57090.000</td>
<td>42998.000</td>
<td>14092.000</td>
</tr>
<tr>
<td>5 102</td>
<td>Whitney</td>
<td>45700.000</td>
<td>57090.000</td>
<td>-11390.000</td>
</tr>
<tr>
<td>6 1507</td>
<td>Wetherby</td>
<td>35745.000</td>
<td>45700.000</td>
<td>-9955.000</td>
</tr>
<tr>
<td>7 1751</td>
<td>Ahmed</td>
<td>34992.000</td>
<td>35745.000</td>
<td>-753.000</td>
</tr>
<tr>
<td>8 1157</td>
<td>Soo</td>
<td>39075.000</td>
<td>34992.000</td>
<td>4083.000</td>
</tr>
</tbody>
</table>

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. Also, the value of PreviousRow is NULL in the first row since it has no predecessor, and hence the Delta value is NULL as well.

Complex analytics

Consider the following query, which lists the top salespeople (defined by total sales) for each product in the database:

```sql
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:

<table>
<thead>
<tr>
<th></th>
<th>Products</th>
<th>SalesRepresentative</th>
<th>total_quantity</th>
<th>total_sales</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>300</td>
<td>299</td>
<td>660</td>
<td>5940.00</td>
</tr>
<tr>
<td>2</td>
<td>301</td>
<td>299</td>
<td>516</td>
<td>7224.00</td>
</tr>
<tr>
<td>3</td>
<td>302</td>
<td>299</td>
<td>336</td>
<td>4704.00</td>
</tr>
<tr>
<td>4</td>
<td>400</td>
<td>299</td>
<td>458</td>
<td>4122.00</td>
</tr>
<tr>
<td>5</td>
<td>401</td>
<td>902</td>
<td>360</td>
<td>3600.00</td>
</tr>
</tbody>
</table>
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.
Consequently, 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
       FROM SalesOrders o KEY JOIN SalesOrderItems s
       KEY JOIN Products p
       GROUP BY s.ProductID, o.SalesRepresentative
```
This rewritten query results in a simpler plan:

Recall that a window operator is computed after the processing of a GROUP BY clause and prior to 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. Consequently, 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. Hence, 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. In the case of ties (rows 7 and 8 in the result set), RANK returns the same value. Consequently, 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 simply by a number of adjacent rows as with the ROWS clause. Using ROWS would yield different results if, for example, some or all of the products happened to have no sales at all in a particular month.

```sql
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:
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:

```sql
SELECT order_qty.ID, o.OrderDate, p.*
FROM ( SELECT s.ID, s.ProductID,
    MAX( s.Quantity ) OVER ( PARTITION BY s.ProductID
    ORDER BY s.ProductID )
    AS max_q
    FROM SalesOrderItems s )
    AS order_qty, Products p, SalesOrders o
WHERE p.ID = ProductID
AND o.ID = order_qty.ID
AND p.Quantity < max_q
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 whether or not it is NULL.

**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.

```sql
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.

```sql
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 ) )
```
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:

```sql
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:

<table>
<thead>
<tr>
<th>OrderDate</th>
<th>rep_of_the_day</th>
</tr>
</thead>
<tbody>
<tr>
<td>2001-04-01</td>
<td>949</td>
</tr>
<tr>
<td>2001-04-02</td>
<td>856</td>
</tr>
<tr>
<td>2001-04-05</td>
<td>902</td>
</tr>
<tr>
<td>2001-04-06</td>
<td>467</td>
</tr>
<tr>
<td>2001-04-07</td>
<td>299</td>
</tr>
</tbody>
</table>

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.

```sql
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
```

Copyright © 2008, iAnywhere Solutions, Inc. - SQL Anywhere 11.0.0 467
ON v.OrderDate = d.dense_order_date AND sales_ranking = 1
ORDER BY d.dense_order_date;

The query returns the following results:

<table>
<thead>
<tr>
<th>OrderDate</th>
<th>rep_of_the_day</th>
</tr>
</thead>
<tbody>
<tr>
<td>2001-04-01</td>
<td>949</td>
</tr>
<tr>
<td>2001-04-02</td>
<td>856</td>
</tr>
<tr>
<td>2001-04-03</td>
<td>856</td>
</tr>
<tr>
<td>2001-04-04</td>
<td>856</td>
</tr>
<tr>
<td>2001-04-05</td>
<td>902</td>
</tr>
<tr>
<td>2001-04-06</td>
<td>467</td>
</tr>
<tr>
<td>2001-04-07</td>
<td>299</td>
</tr>
</tbody>
</table>

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 450

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 of 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. 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 483.

**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.

```sql
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.

<table>
<thead>
<tr>
<th>Employee</th>
<th>Department</th>
<th>Salary</th>
<th>Average</th>
<th>StandardDeviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Lull</td>
<td>100</td>
<td>87900.00</td>
<td>58736.28</td>
</tr>
<tr>
<td>2</td>
<td>Sheffield</td>
<td>100</td>
<td>87900.00</td>
<td>58736.28</td>
</tr>
<tr>
<td>Employee</td>
<td>Department</td>
<td>Salary</td>
<td>Average</td>
<td>StandardDeviation</td>
</tr>
<tr>
<td>----------</td>
<td>------------</td>
<td>-----------</td>
<td>----------</td>
<td>------------------</td>
</tr>
<tr>
<td>Scott</td>
<td>100</td>
<td>96300.00</td>
<td>58736.28</td>
<td>16829.60</td>
</tr>
<tr>
<td>Sterling</td>
<td>200</td>
<td>64900.00</td>
<td>48390.95</td>
<td>13869.60</td>
</tr>
<tr>
<td>Savarino</td>
<td>200</td>
<td>72300.00</td>
<td>48390.95</td>
<td>13869.60</td>
</tr>
<tr>
<td>Kelly</td>
<td>200</td>
<td>87500.00</td>
<td>48390.95</td>
<td>13869.60</td>
</tr>
<tr>
<td>Shea</td>
<td>300</td>
<td>138948.00</td>
<td>59500.00</td>
<td>30752.40</td>
</tr>
<tr>
<td>Blaikie</td>
<td>400</td>
<td>54900.00</td>
<td>43640.67</td>
<td>11194.02</td>
</tr>
<tr>
<td>Morris</td>
<td>400</td>
<td>61300.00</td>
<td>43640.67</td>
<td>11194.02</td>
</tr>
<tr>
<td>Evans</td>
<td>400</td>
<td>68940.00</td>
<td>43640.67</td>
<td>11194.02</td>
</tr>
<tr>
<td>Martinez</td>
<td>500</td>
<td>55500.00</td>
<td>33752.20</td>
<td>9084.50</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Year</th>
<th>Quarter</th>
<th>Average</th>
<th>Variance</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000</td>
<td>1</td>
<td>25.775148</td>
<td>14.2794...</td>
</tr>
<tr>
<td>2000</td>
<td>2</td>
<td>27.050847</td>
<td>15.0270...</td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
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:

```sql
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:

<table>
<thead>
<tr>
<th>Year</th>
<th>Quarter</th>
<th>Average</th>
<th>Variance</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000</td>
<td>1</td>
<td>25.775148</td>
<td>14.3218...</td>
</tr>
<tr>
<td>2000</td>
<td>2</td>
<td>27.050847</td>
<td>15.0696...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

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:

```sql
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:
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:

```sql
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:

<table>
<thead>
<tr>
<th>Year</th>
<th>Quarter</th>
<th>Average</th>
<th>Variance</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000</td>
<td>1</td>
<td>25.775148</td>
<td>205.1158...</td>
</tr>
<tr>
<td>2000</td>
<td>2</td>
<td>27.050847</td>
<td>227.0939...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

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 483.
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 of 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 of 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**

Ranking functions return a value for each row in the input. The ranking functions supported by SQL Anywhere are RANK, DENSE_RANK, PERCENT_RANK, and CUME_DIST. 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. Consequently, 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:

<table>
<thead>
<tr>
<th>Description</th>
<th>Quantity</th>
<th>UnitPrice</th>
<th>Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zipped Sweatshirt</td>
<td>32</td>
<td>24.00</td>
<td>1</td>
</tr>
<tr>
<td>Hooded Sweatshirt</td>
<td>39</td>
<td>24.00</td>
<td>1</td>
</tr>
<tr>
<td>Cotton Shorts</td>
<td>80</td>
<td>15.00</td>
<td>3</td>
</tr>
</tbody>
</table>

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 477.

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:

<table>
<thead>
<tr>
<th>Surname</th>
<th>Salary</th>
<th>Sex</th>
<th>Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shishov</td>
<td>72995.00</td>
<td>F</td>
<td>1</td>
</tr>
<tr>
<td>Wang</td>
<td>68400.00</td>
<td>M</td>
<td>2</td>
</tr>
<tr>
<td>Cobb</td>
<td>62000.00</td>
<td>M</td>
<td>3</td>
</tr>
<tr>
<td>Morris</td>
<td>61300.00</td>
<td>M</td>
<td>4</td>
</tr>
<tr>
<td>Diaz</td>
<td>54900.00</td>
<td>M</td>
<td>5</td>
</tr>
<tr>
<td>Driscoll</td>
<td>48023.69</td>
<td>M</td>
<td>6</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>Surname</th>
<th>Salary</th>
<th>Sex</th>
<th>Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Hildebrand</td>
<td>F</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>Goggin</td>
<td>M</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>Rebeiro</td>
<td>M</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>Bigelow</td>
<td>F</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>Lynch</td>
<td>M</td>
<td>11</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Description</th>
<th>Quantity</th>
<th>UnitPrice</th>
<th>Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hooded Sweatshirt</td>
<td>39</td>
<td>24.00</td>
<td>1</td>
</tr>
<tr>
<td>Zipped Sweatshirt</td>
<td>32</td>
<td>24.00</td>
<td>1</td>
</tr>
<tr>
<td>Cotton Shorts</td>
<td>80</td>
<td>15.00</td>
<td>2</td>
</tr>
</tbody>
</table>

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 474.

**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:
This query returns the following result:

<table>
<thead>
<tr>
<th>SalesRepresentative</th>
<th>Region</th>
<th>total_sales</th>
<th>sales_rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>299</td>
<td>Canada</td>
<td>9312.00</td>
</tr>
<tr>
<td>2</td>
<td>(NULL)</td>
<td>Canada</td>
<td>24768.00</td>
</tr>
<tr>
<td>3</td>
<td>1596</td>
<td>Canada</td>
<td>3564.00</td>
</tr>
<tr>
<td>4</td>
<td>856</td>
<td>Canada</td>
<td>2724.00</td>
</tr>
<tr>
<td>5</td>
<td>299</td>
<td>Central</td>
<td>32592.00</td>
</tr>
<tr>
<td>6</td>
<td>(NULL)</td>
<td>Central</td>
<td>134568.00</td>
</tr>
<tr>
<td>7</td>
<td>856</td>
<td>Central</td>
<td>14652.00</td>
</tr>
<tr>
<td>8</td>
<td>467</td>
<td>Central</td>
<td>14352.00</td>
</tr>
<tr>
<td>9</td>
<td>299</td>
<td>Eastern</td>
<td>21678.00</td>
</tr>
<tr>
<td>10</td>
<td>(NULL)</td>
<td>Eastern</td>
<td>142038.00</td>
</tr>
<tr>
<td>11</td>
<td>902</td>
<td>Eastern</td>
<td>15096.00</td>
</tr>
<tr>
<td>12</td>
<td>690</td>
<td>Eastern</td>
<td>14808.00</td>
</tr>
<tr>
<td>13</td>
<td>1142</td>
<td>South</td>
<td>6912.00</td>
</tr>
<tr>
<td>14</td>
<td>(NULL)</td>
<td>South</td>
<td>45262.00</td>
</tr>
<tr>
<td>15</td>
<td>667</td>
<td>South</td>
<td>6480.00</td>
</tr>
<tr>
<td>16</td>
<td>949</td>
<td>South</td>
<td>5782.00</td>
</tr>
<tr>
<td>17</td>
<td>299</td>
<td>Western</td>
<td>5640.00</td>
</tr>
<tr>
<td>18</td>
<td>(NULL)</td>
<td>Western</td>
<td>37632.00</td>
</tr>
</tbody>
</table>
This query combines multiple groupings through the use of GROUPING SETS. Hence, 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).

**Example 1**

The following example returns a result set that provides a cumulative distribution of the salaries of employees who live in California.

```sql
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:

<table>
<thead>
<tr>
<th>DepartmentID</th>
<th>Surname</th>
<th>Salary</th>
<th>Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>Savarino</td>
<td>72300.00</td>
<td>0.3333333333333333</td>
</tr>
<tr>
<td>200</td>
<td>Clark</td>
<td>45000.00</td>
<td>0.6666666666666667</td>
</tr>
<tr>
<td>200</td>
<td>Overbey</td>
<td>39300.00</td>
<td>1</td>
</tr>
</tbody>
</table>
Example 2

The CUME_DIST function provides a simple method to determine the median of a set of values. CUME_DIST can be used to compute the median value successfully in the face of ties and whether the input contains an even or odd number of rows. Essentially, you need only determine the first row with a CUME_DIST value of greater than or equal to 0.5.

The following query returns the product information for the product with the median unit price:

```
SELECT FIRST *
FROM ( SELECT Description, Quantity, UnitPrice, 
    CUME_DIST( ) OVER ( ORDER BY UnitPrice ASC ) AS CDist
    FROM Products ) As DT
WHERE CDist >= 0.5
ORDER BY CDist;
```

The query returns the following result:

<table>
<thead>
<tr>
<th>Description</th>
<th>Quantity</th>
<th>UnitPrice</th>
<th>CDist</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wool cap</td>
<td>12</td>
<td>10.00</td>
<td>0.5</td>
</tr>
</tbody>
</table>

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)/(n-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:

<table>
<thead>
<tr>
<th>DepartmentID</th>
<th>Surname</th>
<th>Salary</th>
<th>Sex</th>
<th>PctRank</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>200</td>
<td>55700.00</td>
<td>M</td>
<td>0.0</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>Surname</th>
<th>Salary</th>
<th>Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shishov</td>
<td>72995.00</td>
<td>0</td>
</tr>
<tr>
<td>Jordan</td>
<td>51432.00</td>
<td>0.25</td>
</tr>
<tr>
<td>Hildebrand</td>
<td>45829.00</td>
<td>0.5</td>
</tr>
<tr>
<td>Bigelow</td>
<td>31200.00</td>
<td>0.75</td>
</tr>
<tr>
<td>Bertrand</td>
<td>29800.00</td>
<td>1</td>
</tr>
</tbody>
</table>

**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 State IN ( 'UT', 'AZ' ) AND Sex IN ( 'F' )
)
```
WHERE Sex IN ('M') )
AS DerivedTable ( Surname, Salary, Percent )
WHERE Percent < 0.05;

This query returns the following results:

<table>
<thead>
<tr>
<th>Surname</th>
<th>Salary</th>
<th>Percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scott</td>
<td>96300.00</td>
<td>0</td>
</tr>
<tr>
<td>Sheffield</td>
<td>87900.00</td>
<td>0.025</td>
</tr>
<tr>
<td>Lull</td>
<td>87900.00</td>
<td>0.025</td>
</tr>
</tbody>
</table>

For more information about the syntax for the PERCENT_RANK function, see “PERCENT_RANK function [Ranking]” [SQL Anywhere Server - SQL Reference].

Row numbering functions

While there are two row numbering functions supported by SQL Anywhere, NUMBER and ROW_NUMBER, it is recommended that you use the ROW_NUMBER function whenever possible. While both functions perform similar tasks, there are several limitations to the NUMBER function that do not exist for the ROW_NUMBER function. See “NUMBER function [Miscellaneous]” [SQL Anywhere Server - SQL Reference].

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:

<table>
<thead>
<tr>
<th>Description</th>
<th>Quantity</th>
<th>RowNum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tank Top</td>
<td>28</td>
<td>1</td>
</tr>
<tr>
<td>V-neck</td>
<td>54</td>
<td>2</td>
</tr>
</tbody>
</table>
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 in the case of ties.

ROW_NUMBER is designed to work only over the entire partition; hence, 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 of the window aggregate functions supported in SQL Anywhere.

#### Simple aggregate functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Symbol</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUM(X)</td>
<td></td>
<td>$\sum_{i=1}^{n} x_i$</td>
</tr>
<tr>
<td>MAX(X)</td>
<td></td>
<td>$x_i : x_i \geq x_j, i \neq j \forall i,j \in n$</td>
</tr>
<tr>
<td>MIN(X)</td>
<td></td>
<td>$x_i : x_i \leq x_j, i \neq j \forall i,j \in n$</td>
</tr>
<tr>
<td>AVG(X)</td>
<td>$\bar{x}$</td>
<td>$\frac{\sum x_i}{n}$</td>
</tr>
<tr>
<td>COUNT(*)</td>
<td></td>
<td>$n$</td>
</tr>
<tr>
<td>VAR_SAMP(X)</td>
<td>$s_x^2$</td>
<td>$\frac{\sum (x_i-\bar{x})^2}{(n-1)}$</td>
</tr>
<tr>
<td>VAR_POP(X)</td>
<td>$\sigma_x^2$</td>
<td>$\frac{\sum (x_i-\bar{x})^2}{n}$</td>
</tr>
<tr>
<td>VARIANCE(X)</td>
<td></td>
<td>identical to VAR_SAMP(X)</td>
</tr>
<tr>
<td>STDDEV_SAMP(X)</td>
<td>$s_x$</td>
<td>$\sqrt{\frac{\sum (x_i-\bar{x})^2}{(n-1)}}$</td>
</tr>
<tr>
<td>STDDEV_POP(X)</td>
<td>$\sigma_x$</td>
<td>$\sqrt{\frac{\sum (x_i-\bar{x})^2}{n}}$</td>
</tr>
<tr>
<td>STDDEV(X)</td>
<td></td>
<td>identical to STDDEV_SAMP(X)</td>
</tr>
</tbody>
</table>
Statistical aggregate functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>COVAR_SAMP(Y,X)</td>
<td>Co-variance</td>
<td>$s_{xy} = \frac{\sum xy - \left(\sum x\right)\left(\sum y\right)}{(n-1)}$</td>
</tr>
<tr>
<td>COVAR_POP(Y,X)</td>
<td>Co-variance</td>
<td>$\sigma_{xy} = \frac{\sum xy - \frac{1}{n}\left(\sum x\right)\left(\sum y\right)}{n}$</td>
</tr>
<tr>
<td>CORR(Y,X)</td>
<td>Correlation Coefficient</td>
<td>$r = \frac{\sum xy - \frac{1}{n}\left(\sum x\right)\left(\sum y\right)}{(n-1)s_x s_y}$</td>
</tr>
<tr>
<td>REGR_AVGX(Y,X)</td>
<td>Independent mean</td>
<td>$\bar{x}$</td>
</tr>
<tr>
<td>REGR_AVGY(Y,X)</td>
<td>Dependent mean</td>
<td>$\bar{y}$</td>
</tr>
<tr>
<td>REGR_SLOPE(Y,X)</td>
<td>Regression Slope</td>
<td>$b = r \frac{s_y}{s_x}$</td>
</tr>
<tr>
<td>REGR_INTERCEPT(Y,X)</td>
<td>Regression Intercept</td>
<td>$a = \bar{y} - b\bar{x}$</td>
</tr>
<tr>
<td>REGR_R2(Y,X)</td>
<td>‘Goodness-of-fit’</td>
<td>$r^2$</td>
</tr>
<tr>
<td>REGR_COUNT(Y,X)</td>
<td>Sample size</td>
<td>$n$ (non-null (Y,X) pairs)</td>
</tr>
<tr>
<td>REGR_SXX(Y,X)</td>
<td>Sum of squares (x)</td>
<td>$\sum x^2 - \frac{\left(\sum x\right)^2}{n}$</td>
</tr>
<tr>
<td>REGR_SYY(Y,X)</td>
<td>Sum of squares (y)</td>
<td>$\sum y^2 - \frac{\left(\sum y\right)^2}{n}$</td>
</tr>
<tr>
<td>REGR_SXY(Y,X)</td>
<td>Sum of products</td>
<td>$\sum xy - \frac{\left(\sum y\right)\left(\sum x\right)}{n}$</td>
</tr>
</tbody>
</table>
CHAPTER 12

Using subqueries

Contents

Introduction to subqueries ........................................................................................................... 486
Subqueries in the WHERE clause ................................................................................................. 492
Subqueries in the HAVING clause ............................................................................................... 493
Subquery tests ............................................................................................................................. 495
Optimizer automatic conversion of subqueries to joins ............................................................... 502
Using subqueries

Introduction to subqueries

A relational database allows you to 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.

Categorization of 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 SalesOrdersItems)
```
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;
```

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>UnitPrice</th>
</tr>
</thead>
<tbody>
<tr>
<td>401</td>
<td>Baseball Cap</td>
<td>10.00</td>
</tr>
<tr>
<td>300</td>
<td>Tee Shirt</td>
<td>9.00</td>
</tr>
<tr>
<td>400</td>
<td>Baseball Cap</td>
<td>9.00</td>
</tr>
<tr>
<td>500</td>
<td>Visor</td>
<td>7.00</td>
</tr>
<tr>
<td>501</td>
<td>Visor</td>
<td>7.00</td>
</tr>
</tbody>
</table>

### 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:

```
```
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. Thus, 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:

```sql
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 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.

```sql
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.

```sql
SELECT ID, LineID
FROM SalesOrderItems
WHERE ShipDate = ANY (SELECT OrderDate
                       FROM SalesOrders
                       WHERE FinancialCode IN {...
```
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:

```sql
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:

```sql
SELECT CompanyName, State, ( SELECT MAX( ID )
FROM SalesOrders
WHERE SalesOrders.CustomerID = Customers.ID )
FROM Customers WHERE State = 'WA';
```

<table>
<thead>
<tr>
<th>CompanyName</th>
<th>State</th>
<th>MAX(SalesOrders.ID)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Custom Designs</td>
<td>WA</td>
<td>2547</td>
</tr>
<tr>
<td>It's a Hit!</td>
<td>WA</td>
<td>(NULL)</td>
</tr>
</tbody>
</table>

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.

```sql
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.

```sql
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

```sql
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.

```sql
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 chapter “Querying data” on page 291 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

```sql
SELECT Name, AVG( Quantity )
FROM Products
GROUP BY Name
HAVING AVG( Quantity ) > 2* ( 
    SELECT AVG( Quantity ) 
    FROM SalesOrderItems 
); 
```

<table>
<thead>
<tr>
<th>name</th>
<th>AVG( Products.Quantity )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseball Cap</td>
<td>62.000000</td>
</tr>
<tr>
<td>Shorts</td>
<td>80.000000</td>
</tr>
<tr>
<td>Tee Shirt</td>
<td>52.333333</td>
</tr>
</tbody>
</table>

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.

```sql
SELECT ProductID, LineID
FROM SalesOrderItems
GROUP BY ProductID, LineID
HAVING 2* AVG( Quantity ) > ( 
    SELECT Quantity 
); 
```
Using subqueries

```sql
FROM Products
WHERE Products.ID = SalesOrderItems.ProductID );
```

<table>
<thead>
<tr>
<th>ProductID</th>
<th>LineID</th>
</tr>
</thead>
<tbody>
<tr>
<td>601</td>
<td>3</td>
</tr>
<tr>
<td>601</td>
<td>2</td>
</tr>
<tr>
<td>601</td>
<td>1</td>
</tr>
<tr>
<td>600</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

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.
Subquery tests

The chapter “Querying data” on page 291 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:

```sql
SELECT Name, Description, Quantity
FROM Products
WHERE Quantity < 2 * (SELECT AVG(Quantity)
FROM SalesOrderItems);
```

<table>
<thead>
<tr>
<th>name</th>
<th>Description</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tee Shirt</td>
<td>Tank Top</td>
<td>28</td>
</tr>
<tr>
<td>Baseball Cap</td>
<td>Wool cap</td>
<td>12</td>
</tr>
<tr>
<td>Visor</td>
<td>Cloth Visor</td>
<td>36</td>
</tr>
<tr>
<td>Visor</td>
<td>Plastic Visor</td>
<td>28</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

The following subquery retrieves a single value—the average quantity of items of each type per customer's order—from the SalesOrderItems table.
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' ) );
```

<table>
<thead>
<tr>
<th>GivenName</th>
<th>Surname</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mary Anne</td>
<td>Shea</td>
</tr>
<tr>
<td>Jose</td>
<td>Martinez</td>
</tr>
</tbody>
</table>

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.

```sql
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.

```sql
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.

```sql
SELECT ID, CustomerID
FROM SalesOrders
WHERE OrderDate > ANY (
    SELECT ShipDate
    FROM SalesOrderItems
    WHERE ID=2005 );
```
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 thus 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

Like the ANY test, the ALL test is used in conjunction with one of the six 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 of the comparisons yield TRUE results, the ALL test returns TRUE.

Example

Here is a request naturally handled with the ALL test: "Find the order and customer IDs of those 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 
); 
```

<table>
<thead>
<tr>
<th>ID</th>
<th>CustomerID</th>
</tr>
</thead>
<tbody>
<tr>
<td>2002</td>
<td>102</td>
</tr>
<tr>
<td>2003</td>
<td>103</td>
</tr>
<tr>
<td>2004</td>
<td>104</td>
</tr>
<tr>
<td>2005</td>
<td>101</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

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 thus 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)
```
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 ) );
```

<table>
<thead>
<tr>
<th>GivenName</th>
<th>Surname</th>
</tr>
</thead>
<tbody>
<tr>
<td>Almen</td>
<td>de Joie</td>
</tr>
<tr>
<td>Grover</td>
<td>Pendelton</td>
</tr>
<tr>
<td>Ling Ling</td>
<td>Andrews</td>
</tr>
<tr>
<td>Bubba</td>
<td>Murphy</td>
</tr>
</tbody>
</table>

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 489.

```
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' );
```

<table>
<thead>
<tr>
<th>OrderDate</th>
<th>SalesRepresentative</th>
</tr>
</thead>
<tbody>
<tr>
<td>2001-01-05</td>
<td>1596</td>
</tr>
<tr>
<td>2000-01-27</td>
<td>667</td>
</tr>
<tr>
<td>2000-11-11</td>
<td>467</td>
</tr>
<tr>
<td>2001-02-04</td>
<td>195</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

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 );

<table>
<thead>
<tr>
<th>name</th>
<th>Description</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tee Shirt</td>
<td>Tank Top</td>
<td>28</td>
</tr>
<tr>
<td>Baseball Cap</td>
<td>Wool cap</td>
<td>12</td>
</tr>
<tr>
<td>Visor</td>
<td>Cloth Visor</td>
<td>36</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

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 373

Subquery that follows a comparison operator

A subquery that follows a comparison operator (=, >, <, >=, <=, !=, <>, !>, !<) is called a comparison. The optimizer converts these subqueries to joins provided that 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
Using subqueries

```
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 one of the keywords ALL, ANY and SOME is called a quantified comparison. The optimizer converts these subqueries to joins provided that:

- 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 thus 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:

\[
\text{NOT } \text{column-name } \text{operator ANY ( subquery-expression )}
\]

\[\text{column-name inverse-operator ALL ( subquery-expression )}\]

These expressions are generally equivalent as well:

\[
\text{NOT } \text{column-name } \text{operator ALL ( subquery-expression )}
\]

\[\text{column-name inverse-operator ANY ( subquery-expression )}\]

where \text{inverse-operator} is obtained by negating \text{operator}, as shown in the table below:

<table>
<thead>
<tr>
<th>operator</th>
<th>inverse-operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>&lt;&gt;</td>
</tr>
<tr>
<td>&lt;</td>
<td>=&gt;</td>
</tr>
<tr>
<td>&gt;</td>
<td>&lt;=</td>
</tr>
<tr>
<td>&lt;=</td>
<td>&gt;</td>
</tr>
<tr>
<td>=&gt;</td>
<td>&lt;</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>=</td>
</tr>
</tbody>
</table>

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.

```sql
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**

```sql
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.

```sql
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**

In some cases, SQL Anywhere converts a query with the IN operator to one with an ANY operator, and decides accordingly whether to convert the subquery to a join. For example, the following two expressions are equivalent:

```sql
WHERE column-name IN ( subquery-expression )
```

```sql
WHERE column-name = ANY ( subquery-expression )
```

Likewise, the following two queries are equivalent:

```sql
SELECT GivenName, Surname
FROM Employees
WHERE EmployeeID IN (
Subquery that follows EXISTS

The optimizer converts a subquery that follows the EXISTS keyword EXISTS 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:

```sql
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 of the tables in the FROM clause. So, this query returns the same rows as does the one with the subquery:

```sql
SELECT DISTINCT GivenName, Surname
FROM Customers, SalesOrders
WHERE ( SalesOrders.OrderDate > '2001-07-13' ) AND  
( Customers.ID = SalesOrders.CustomerID );
```
CHAPTER 13

Adding, changing, and deleting data

Contents

Data modification statements ................................................................. 510
Adding data using INSERT ........................................................................ 513
Changing data using UPDATE ................................................................. 517
Changing data using INSERT ................................................................. 519
Deleting data using DELETE ................................................................. 520
Data modification statements

The statements you use to add, change, or delete data are called **data modification** statements, also referred to as the **data modification language** (DML) part of SQL. The three main DML statements are:

- **INSERT statement** adds new rows to a table
- **UPDATE statement** changes existing rows in a table
- **DELETE statement** removes specific rows from a table

Any single INSERT, UPDATE, or DELETE statement changes the data in only one table or view.

In addition to the statements above, the LOAD TABLE and TRUNCATE TABLE statements are especially useful for bulk loading and deleting of data.

See also

- “INSERT statement” [SQL Anywhere Server - SQL Reference]
- “UPDATE statement” [SQL Anywhere Server - SQL Reference]
- “DELETE statement” [SQL Anywhere Server - SQL Reference]

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 119.

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 of 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

Suppose that a system failure or power outage suddenly takes your database server down. SQL Anywhere is carefully designed to protect the integrity of your database in such circumstances. It provides you with a number of independent means of restoring your database. For example, it provides you with a log file that you can store on a separate drive so that should the system failure damage one drive, you still have a means of restoring your data. In addition, when you use a log file, SQL Anywhere does not have to update your database as frequently, which improves your database's performance.

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 110.
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 of 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. In some cases, the insert operation may fail and return an error. The following table shows the possible outcomes depending on the value being inserted (if any) and the column settings:

<table>
<thead>
<tr>
<th>Value being inserted</th>
<th>Nullable, NULLable</th>
<th>Nullable, with DEFAULT</th>
<th>Not nullable, with DEFAULT</th>
<th>Not nulltable, with DEFAULT AUTOINCREMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;none&gt;</td>
<td>NULL</td>
<td>SQL error</td>
<td>DEFAULT value</td>
<td>DEFAULT value</td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
<td>SQL error</td>
<td>NULL</td>
<td>SQL error</td>
</tr>
<tr>
<td>specified value</td>
<td>specified value</td>
<td>specified value</td>
<td>specified value</td>
<td>specified value</td>
</tr>
</tbody>
</table>

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 103.
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 97.

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 of 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 some 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]
- “Storing BLOBs in the database” on page 25
- “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].

**The 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.

**The 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

```sql
UPDATE Products
SET Size = 'Extra Large'
WHERE Name = 'Tee Shirt'
  AND Size = 'One Size Fits All';
```

**The 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
```

The 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.

The 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;
```
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;
```
Part IV. Query Optimizer

This part describes the query optimizer and how it works, including: phases of query processing, strategies used by the optimizer, and tips on how to maximize the optimizer's performance. It also describes how to view and analyze execution plans for queries.
CHAPTER 14

Query optimization and execution

Contents

Phases of query processing ........................................................................................................ 526
Semantic query transformations ................................................................................................. 528
How the optimizer works ........................................................................................................... 541
Query execution algorithms ....................................................................................................... 565
Reading execution plans ......................................................................................................... 589
Physical data organization and access ..................................................................................... 615
Phases of query processing

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.

The following list 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 other steps of optimization.

- **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 of the objects referenced in the original query.

  When referencing a temporary table from multiple procedures, a potential issue can arise if the temporary table definitions are inconsistent and statements referencing the table are cached. See “Referencing temporary tables within procedures” on page 53.

- **Query rewrite 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 rewriting, 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.

- **Optimization phase** The optimization phase uses a different internal representation of the query, the query optimization structure, which is built from the parse tree.

  - **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 in order 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 part or all 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 602.

- **Execution phase**  The result of the query is computed using the query execution plan built in the
previous phase.

Statements that have no result sets, such as UPDATE or DELETE statements, also have a query execution
plan.

**Queries that bypass optimization**

Almost all statements pass through the phases described above, with two exceptions: simple queries that
bypass the optimizer, and queries whose plans are already cached by the database server.

If a query is recognized as a simple query, a heuristic rather than cost-based optimization is used—that is,
the optimization phase is skipped and the query execution plan is built directly from the parse tree
representation of the query. A simple query is a DYNAMIC SCROLL or NO SCROLL cursor that does not
contain any kind of subquery, more than one table, a proxy table, a user-defined function, NUMBER(*),
UNION, aggregation, DISTINCT, GROUP BY, or more than one predicate on a single column. Simple
queries can contain ORDER BY only as long as the WHERE clause contains conditions on each of the
primary key columns.

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 562.

You can force the database server to optimize a simple query, or to optimize a stored procedure or user-
defined functions that contains a query, by using the OPTION FORCE OPTIMIZATION clause in the
SELECT statement. When this clause is specified, plan caching is not used. See “SELECT statement” [SQL
Anywhere Server - SQL Reference].
Semantic query transformations

To operate efficiently, SQL Anywhere usually rewrites your query, possibly in several steps, into a new form. It ensures that the new version computes the same result, even though it expresses the query in a new way. In other words, SQL Anywhere rewrites your queries into semantically equivalent, but syntactically different, forms.

SQL Anywhere can perform a number of 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, the optimizer tries as much as possible to rewrite subqueries with joins. It is important to realize that the optimizer rewrites your SQL statements to make them more efficient.

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:

```c
if ( X != NULL ) {
    if ( X->var != 0 ) {
        ... statements ...
    }
}
```

```c
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.

Types of semantic transformations

In the Query Rewrite phase, SQL Anywhere performs a number of 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.

### 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;
```

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. Hence, 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';
```
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.

```sql
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:

```sql
SELECT s.*
FROM Products p JOIN SalesOrderItems s
     ON p.ID = s.ProductID
WHERE p.ID = 300 AND p.Quantity > 20;
```

Similarly, the following query contains a conjunctive EXISTS predicate in the subquery. This subquery can match more than one row.

```sql
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.

```sql
SELECT DISTINCT p.*
FROM Products p JOIN SalesOrderItems s
     ON p.ID = s.ProductID
WHERE s.ID = 2001;
```

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.

```sql
SELECT *
FROM Products p
WHERE EXISTS
  ( SELECT *
      FROM SalesOrderItems s
      WHERE s.ProductID = p.ID
      AND s.ID = 2001);
```

```
Work[ DistH[ s<FK_ID_ID> JNL p<Products> ] ]
```
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. In cases where 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:

```sql
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 of the orders. Now consider the following query over this view:

```sql
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:

```sql
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:

```sql
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 of the orders for two sales reps:

```sql
SELECT *
FROM SalesOrders
WHERE SalesRepresentative = 902 OR SalesRepresentative = 195;
```

This query is semantically equivalent to:

```sql
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. Consequently, 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:
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. Consequently, 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. Hence 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:

```sql
SELECT s.ID, s.LineID, p.ID
FROM SalesOrderItems s KEY JOIN Products p
FOR READ ONLY;
```

The query would be rewritten as:

```sql
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.

```sql
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:

```sql
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 simply \( X \).
- \( X+0 \) 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:

```sql
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; hence, 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:

```sql
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:

```sql
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:

```sql
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.

```sql
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. Thus, 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.

```sql
SELECT * FROM Customers c WHERE NOT EXISTS ( SELECT * FROM SalesOrders o WHERE o.OrderDate = '2001-01-01' AND o.CustomerID = c.ID );
```

This query is semantically equivalent to the one above, but when expressed in this new syntax, several advantages become apparent:

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. In the case of query above, caching does not help because customer identification numbers are unique in the Customers table; hence, 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 583.

See also

- “Correlated and uncorrelated subqueries” on page 489
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:

  ```sql
  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:

  ```sql
  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:

  ```sql
  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:

```sql
SELECT F1( e.EmployeeID, 2.5 ) FROM Employees e;
```

The database server may rewrite the query as follows:

```sql
SELECT CAST( e.EmployeeID AS INT ) * CAST( 2.5 AS INT ) FROM Employees e;
```

See also

- “Introduction to user-defined functions” on page 814
- “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 user-defined functions” on page 814
- “CREATE FUNCTION 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 the Plan tab in the Results pane. To change the degree of detail that is displayed, change the setting on the Plan tab of the Options window (available from the Tools menu). See “Reading graphical plans” on page 602, and “Reading execution plans” on page 589.

Cost based

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 [database]” [SQL Anywhere Server - Database Administration].

Syntax independent

Most commands can be expressed in many different ways using the SQL language. These expressions are semantically equivalent in that they accomplish 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 or not a query contains a non-materialized view.

A good plan, not necessarily the best plan

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 [database]” [SQL Anywhere Server - Database Administration].
In the case of expensive and complicated queries, or when the optimization level is set high, the optimizer works longer. In the case of 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” on page 543.

**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%.

**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.

In many cases, the optimizer uses more sophisticated heuristics. For example, the optimizer uses default estimates only in cases where 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.

**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 587.

**See also**

For more information about column statistics, see “SYSCOLSTAT system view” [SQL Anywhere Server - SQL Reference].

For information about obtaining the selectivities of predicates and the distribution of column values, see:

- “sa_get_histogram system procedure” [SQL Anywhere Server - SQL Reference]
- “Histogram utility (dbhist)” [SQL Anywhere Server - Database Administration]
- “ESTIMATE function [Miscellaneous]” [SQL Anywhere Server - SQL Reference]
- “ESTIMATE_SOURCE function [Miscellaneous]” [SQL Anywhere Server - SQL Reference]

**Updating column statistics**

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. 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 accordingly.

With more accurate column statistics available to it, the optimizer can compute better estimates, thus improving 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.

For more information about column statistics, see:
- “SYSCOLSTAT system view” [SQL Anywhere Server - SQL Reference]
- “DROP STATISTICS statement” [SQL Anywhere Server - SQL Reference]
- “CREATE STATISTICS statement” [SQL Anywhere Server - SQL Reference]
- “update_statistics option [database]” [SQL Anywhere Server - Database Administration]
- “collect_statistics_on_dml_updates option [database]” [SQL Anywhere Server - Database Administration]

**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.

```sql
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” on page 543.

**Underlying assumptions**

A number of 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 thus 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” on page 543, and “Working with indexes” on page 86.

**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 [database]” [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 [database]” [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” on page 543.

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 194.

See also

● “Predicate analysis” on page 548

Virtual memory is a scarce resource

The operating system and a number of applications frequently share the memory of a typical 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, as well as 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 of 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. Moreover, 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 property: “Server-level properties” [SQL Anywhere Server - Database Administration]
- QueryMemPages property: “Server-level properties” [SQL Anywhere Server - Database Administration]
- QueryMemPercentOfCache property: “Server-level properties” [SQL Anywhere Server - Database Administration]
- “query_mem_timeout option [database]” [SQL Anywhere Server - Database Administration]
- “Setting the database server's multiprogramming level” [SQL Anywhere Server - Database Administration]
- “Graphical plan with statistics” on page 603
- “-ch server option” [SQL Anywhere Server - Database Administration]

Predicate analysis

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';
```

You can sometimes rewrite a query to avoid using a function, thus making 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 consequently 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 48.
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.

<table>
<thead>
<tr>
<th>Sargable</th>
<th>Non-sargable</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x = 10$</td>
<td>$x &lt; &gt; 10$</td>
</tr>
<tr>
<td>$x$ IS NULL</td>
<td>$x$ IS NOT NULL</td>
</tr>
<tr>
<td>$x &gt; 25$</td>
<td>$x = 4$ OR $y = 5$</td>
</tr>
<tr>
<td>$x = z$</td>
<td>$x = y$</td>
</tr>
<tr>
<td>$x$ IN (4, 5, 6)</td>
<td>$x$ NOT IN (4, 5, 6)</td>
</tr>
<tr>
<td>$x$ LIKE 'pat%'</td>
<td>$x$ LIKE '%tern'</td>
</tr>
<tr>
<td>$x = 20 - 2$</td>
<td>$x + 2 = 20$</td>
</tr>
</tbody>
</table>

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 $t$: $x \geq 'pat'$ 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 a number of 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.

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). Consequently, 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.

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 “Enabling and disabling materialized views” on page 80.
- the materialized view is enabled for use in optimization. See “Enabling and disabling optimizer use of a materialized view” on page 82.
- the materialized view has been initialized. See “Initializing materialized views” on page 74.
- the materialized view meets all of the optimizer requirements for consideration. See “Requirements for View Matching algorithm” on page 552.
- 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 71.
- 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 83.

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 for the current connection

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 71.

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 whether a materialized view was considered by the optimizer**

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 589.

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, in order to see the access plans enumerated by the optimizer. For more information about this tracing type, see “Application profiling” on page 188, and “Choosing a diagnostic tracing level” on page 201.

For more information about the enumeration phase of optimization, see “Phases of query processing” on page 526.

**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.

**Requirements for View Matching algorithm**

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, as well as 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 589
- “Phases of query processing” on page 526
- “Application profiling” on page 188

View matching

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.

Query evaluation step

During the query evaluation step, 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 of the tables referenced by the query are updatable (see “View matching” on page 553).

  Consequently, the optimizer will 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 thus 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].

- The query's execution plan has been cached, as in the case of queries contained inside stored procedures and user-defined functions. 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 562.

Materialized view evaluation step

The materialized view evaluation step involves determining which of the existing materialized views could 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 557.

- 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 71.

- 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 83.

- 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
The composite aggregate functions that can be computed from the simple aggregate functions are:

- $\text{SUM}(x)$
- $\text{COUNT}(x)$
- $\text{SUM}(	ext{CAST}(x \text{ AS DOUBLE}))$
- $\text{SUM}(	ext{CAST}(x \text{ AS DOUBLE}) \times \text{CAST}(x \text{ AS DOUBLE}))$
- $\text{VAR}_\text{SAM}(x)$
- $\text{VAR}_\text{POP}(x)$
- $\text{VARIANCE}(x)$
- $\text{STDDEV}_\text{SAM}(x)$
- $\text{STDDEV}_\text{POP}(x)$
- $\text{STDDEV}(x)$

The following statistical aggregate functions:

- $\text{COVAR}_\text{SAM}(y, x)$
- $\text{COVAR}_\text{POP}(y, x)$
- $\text{CORR}(y, x)$
- $\text{REGR}_\text{AVGX}(y, x)$
- $\text{REGR}_\text{AVGY}(y, x)$
- $\text{REGR}_\text{SLOPE}(y, x)$
- $\text{REGR}_\text{INTERCEPT}(y, x)$
- $\text{REGR}_\text{R2}(y, x)$
- $\text{REGR}_\text{COUNT}(y, x)$
- $\text{REGR}_\text{SXX}(y, x)$
- $\text{REGR}_\text{SYY}(y, x)$
- $\text{REGR}_\text{SXY}(y, x)$

can be computed from the following simple aggregate functions:

- $\text{SUM}(y_1)$
- $\text{SUM}(x_1)$
- $\text{COUNT}(x_1)$
- $\text{COUNT}(y_1)$
- $\text{SUM}(x_1 \times y_1)$
- $\text{SUM}(y_1 \times x_1)$
- $\text{SUM}(x_1 \times x_1)$
- $\text{SUM}(y_1 \times y_1)$

where $x_1 = \text{CAST}((\text{IFNULL}(x, x, y) \text{ AS DOUBLE}))$ and $y_1 = \text{CAST}((\text{IFNULL}(y, y, x) \text{ AS DOUBLE})).$

**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 $V_{\text{Canada}}$ 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:

```sql
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:

```sql
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, hence it is advisable to store only simple aggregate functions in the materialized views.
The materialized view \textit{V\_quantity}, below, pre-computes 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 \textit{Work[ GrByH[ V\_quantity<seq> ] ]}, corresponding to Query 3\textsubscript{v}).

Query 4, which doesn’t reference the extension table \textit{SalesOrders}, can still use \textit{V\_quantity} as the view contains all the data necessary to compute Query 4 (the short plan is \textit{Work[ GrByH[ V\_quantity<seq> ] ]}, corresponding to Query 4\textsubscript{v}).

\begin{verbatim}
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\textsubscript{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\textsubscript{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;
\end{verbatim}
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_ship_date, 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:

```
CREATE MATERIALIZED VIEW V_ship_date 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';

CREATE INDEX V_ship_date_date ON V_ship_date(ShipDate);
```

```
REFRESH MATERIALIZED VIEW V_ship_date;
```

```
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_ship_date.ID, V_ship_date.Description,
        V_ship_date.Quantity, V_ship_date.ShipDate,
        so.CustomerID, V_Canada.CompanyName,
        SUM( V_ship_date.Quantity ) OVER ( PARTITION BY V_ship_date.ProductID
                                           ORDER BY V_ship_date.ShipDate
                                           ROWS BETWEEN UNBOUNDED PRECEDING
                                           AND CURRENT ROW ) AS cumulative_qty
  FROM V_ship_date
      LEFT OUTER JOIN ( SalesOrders so JOIN V_Canada
                       ON ( V_Canada.ID = so.CustomerID AND V_Canada.State = 'ON' )
                       ON ( V_ship_date.ID = so.ID ) WHERE V_ship_date.ShipDate >= '2000-01-01'
      AND V_ship_date.ShipDate <= '2000-12-31'
      AND V_ship_date.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:

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_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
FROM V_SalesOrderItems_2000 as V
FOR READ ONLY;
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;

Optimization for 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 prod_qty (ShipDate ASC, Quantity ASC) exists on the SalesOrderItems table. Then the query

```sql
SELECT MIN( Quantity )
FROM SalesOrderItems
WHERE ShipDate = '2000-03-25';
```

is rewritten internally as

```sql
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<prod_qty> ] ]
```

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, as well as 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 optimizer caches plans for:
queries and INSERT, UPDATE, and DELETE statements performed inside stored procedures, user-defined functions, and triggers

SELECT, INSERT, UPDATE, or DELETE statement that qualify to bypass optimization. See “Queries that bypass optimization” on page 527.

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 uniquely identify a row using the primary key. 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. A reusable plan does not use the values of host variables for selectivity estimation or rewrite optimizations. The reusable plan may have a higher cost because of this. If the reusable plan has the same structure as the plans observed in the previous executions of the statement, the database server chooses to add the reusable plan to a plan cache. Otherwise, the benefit of optimizing on each execution outweighs the savings from avoiding optimization, and the execution plan is not cached.

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. This is typically substantially faster than optimizing the statement. 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 relatively 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 [database]” [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-level 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

● “Improving performance with materialized views” on page 550
● “materialized_view_optimization option [database]” [SQL Anywhere Server - Database Administration]
Query optimization and execution

- “DB_PROPERTY function [System]” [SQL Anywhere Server - SQL Reference]
- “CONNECTIONPROPERTY function [System]” [SQL Anywhere Server - SQL Reference]
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.

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 589, and “Reading graphical plans” on page 602.

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-level 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]
- “Setting the database server's multiprogramming level” [SQL Anywhere Server - Database Administration]
- “The memory governor” on page 547

- 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-level 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 585). 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 [database]” in 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 [database]” in 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 “Server-level properties” in 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 in some cases, 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 of 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 more accurately the benefits of parallel execution, and in the case of multiple spindles, the optimizer is much more likely to choose execution plans with some degree of parallelism.

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 (see “-gn server option” [SQL Anywhere Server - Database Administration]). 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**

- “max_query_tasks option [database]” [SQL Anywhere Server - Database Administration]
- “Threading in SQL Anywhere” [SQL Anywhere Server - Database Administration]
- “-gtc server option” [SQL Anywhere Server - Database Administration]
- “Setting the database server's multiprogramming level” [SQL Anywhere Server - Database Administration]
- “Exchange algorithm (Exchange)” on page 585
- “Reading execution plans” on page 589
- “ALTER DATABASE statement” [SQL Anywhere Server - SQL Reference]

**Circumstances under which parallelism is used**

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.
Query optimization and execution

- 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 566
- “Threading in SQL Anywhere” [SQL Anywhere Server - Database Administration]
- “max_query_tasks option [database]” [SQL Anywhere Server - Database Administration]
- “priority option [database]” [SQL Anywhere Server - Database Administration]
- max_query_tasks, priority, NumLogicalProcessorsUsed, and ActiveReq properties: “Server-level 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 of 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]
- “Predicate analysis” on page 548
- “optimization_goal option [database]” [SQL Anywhere Server - Database Administration]
- “Reading execution plans” on page 589

**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]
- “Predicate analysis” on page 548
- “Reading execution plans” on page 589

**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 of 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 a majority of 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 of 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 of 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 \texttt{correlation-name\textless \textit{ROWID}}\textgreater, where \textit{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. Consequently, 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]

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. In a nutshell, 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 of 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 of 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 “Setting 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-level properties” [SQL Anywhere Server - Database Administration], and “Setting 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 573, and “Recursive common table expressions” on page 426.

---

**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 573, and “Recursive common table expressions” on page 426.

---

**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 utilized in cases where 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 “Setting 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-level 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 in cases where 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 “Setting 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-level 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 of 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, provided that 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 [database]” [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 as soon as 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 577, and “OrderedDistinct algorithm (DistO)” on page 578.

**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 as soon as 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 “Setting 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 578, “OrderedGroupBy algorithm (GrByO)” on page 579, and “SingleRowGroupBy algorithm (GrByS)” on page 580.

**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 of 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].
Anywhere” [SQL Anywhere Server - Database Administration], and “Setting the database server's multiprogramming level” [SQL Anywhere Server - Database Administration].

ClusteredHashGroupBy algorithm (GrByHClust)

In some cases, 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 [database]” [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 437.

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 578.

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 579.

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. In many cases 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.
In the case of 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 clause” [SQL Anywhere Server - SQL Reference]
- “Performing set operations on query results with UNION, INTERSECT, and EXCEPT” on page 365
- QueryLowMemoryStrategy connection property: “Connection-level properties” [SQL Anywhere Server - Database Administration]
- QueryLowMemoryStrategy database property: “Database-level properties” [SQL Anywhere Server - Database Administration]
- Query: Low Memory Strategies statistic: “Performance Monitor statistics” on page 227

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. In many cases 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.

In the case of INTERSECT, MergeIntersect or HashIntersect is 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 clause” [SQL Anywhere Server - SQL Reference]
- “Performing set operations on query results with UNION, INTERSECT, and EXCEPT” on page 365
- QueryLowMemoryStrategy connection property: “Connection-level properties” [SQL Anywhere Server - Database Administration]
- QueryLowMemoryStrategy database property: “Database-level properties” [SQL Anywhere Server - Database Administration]
- Query: Low Memory Strategies statistic: “Performance Monitor statistics” on page 227
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 419.

RecursiveUnion algorithm (RU)

RecursiveUnion is employed during the execution of recursive union queries. See “Recursive common table expressions” on page 426.

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 365.

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 clauses. 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 577, and “OrderedDistinct algorithm (DistO)” on page 578.

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 582 and “UnionAll algorithm (UA)” on page 582.

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 of 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 “Setting 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. Hence, 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, NEW_ID, 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 323.

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 based operations 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 303 and “FROM clause” [SQL Anywhere Server - SQL Reference].

Example

The following query has derived tables in its graphical plan:

```sql
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 a number of 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 pre-filter.

See also
- “The WHERE clause: Specifying rows” on page 305
- “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 quickly determine whether or not there is a matching row of that value (subject to the existence of hash collisions).

For example, consider the plan:

\[ R<idx> \ast JH S<seq> JH* T<idx> \]

Here you are joining R to S and T. The database server reads all of 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 in cases where 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.

```sql
SELECT *
FROM Employees
WHERE EmployeeID IN ( 102, 105, 129 )
```
To accomplish 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.

```sql
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 in cases where 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, as well as 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]
- “Procedure statistics” on page 543
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.

In the case of 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 589

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 “Altering text configuration objects” on page 341.

Window algorithm (Window)

Window is used when evaluating OLAP queries that employ window functions. See “Window functions” on page 450.
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 “Phases of query processing” on page 526.

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 537
- “Improving performance with materialized views” on page 550

For information about the algorithms and methods that the optimizer uses to implement your query, see “Query execution algorithms” on page 565.

For more information about how to access a graphical plan, see “Viewing graphical plans” on page 610.

For information about how to read execution plans, see “Reading text plans” on page 600, and “Reading graphical plans” on page 602.

Abbreviations in the plan

Following are the abbreviations that you see in execution plans.
<table>
<thead>
<tr>
<th>Short text plan</th>
<th>Long text plan</th>
<th>Additional information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Costed Best Plan</td>
<td></td>
<td>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 541.</td>
</tr>
<tr>
<td>Costed Plans</td>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>DELETE</td>
<td>Delete</td>
<td>The root node of a DELETE operation. See “DELETE statement” [SQL Anywhere Server - SQL Reference].</td>
</tr>
<tr>
<td>DistH</td>
<td>HashDistinct</td>
<td>See “HashDistinct algorithm (DistH)” on page 577.</td>
</tr>
<tr>
<td>DistO</td>
<td>Ordered-Distinct</td>
<td>See “OrderedDistinct algorithm (DistO)” on page 578.</td>
</tr>
<tr>
<td>DP</td>
<td>Decode-Postings</td>
<td>See “DecodePostings (DP)” on page 584.</td>
</tr>
<tr>
<td>DT</td>
<td>DerivedTable</td>
<td>See “DerivedTable algorithm (DT)” on page 584.</td>
</tr>
<tr>
<td>EAH</td>
<td>HashExceptAll</td>
<td>See “Except algorithms (EAH, EAM, EH, EM)” on page 580.</td>
</tr>
<tr>
<td>EAM</td>
<td>MergeExceptAll</td>
<td>See “Except algorithms (EAH, EAM, EH, EM)” on page 580.</td>
</tr>
<tr>
<td>EH</td>
<td>HashExcept</td>
<td>See “Except algorithms (EAH, EAM, EH, EM)” on page 580.</td>
</tr>
<tr>
<td>EM</td>
<td>MergeAccept</td>
<td>See “Except algorithms (EAH, EAM, EH, EM)” on page 580.</td>
</tr>
<tr>
<td>Filter</td>
<td>Filter</td>
<td>See “Filter algorithms (Filter, PreFilter)” on page 585.</td>
</tr>
<tr>
<td>Short text plan</td>
<td>Long text plan</td>
<td>Additional information</td>
</tr>
<tr>
<td>----------------</td>
<td>----------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>GrByH</td>
<td>Hash-GroupBy</td>
<td>See “HashGroupBy algorithm (GrByH)” on page 578.</td>
</tr>
<tr>
<td>GrByH-Clust</td>
<td>Hash-GroupBy-Clustered</td>
<td>See “ClusteredHashGroupBy algorithm (GrByHClust)” on page 579.</td>
</tr>
<tr>
<td>GrByHSets</td>
<td>Hash-GroupBy-Sets</td>
<td>See “HashGroupBySets algorithm (GrByHSets)” on page 579.</td>
</tr>
<tr>
<td>GrByO</td>
<td>Ordered-GroupBy</td>
<td>See “OrderedGroupBy algorithm (GrByO)” on page 579.</td>
</tr>
<tr>
<td>GrByOSets</td>
<td>Ordered-GroupBy-Sets</td>
<td>See “OrderedGroupBySets algorithm (GrByOSets)” on page 580.</td>
</tr>
<tr>
<td>GrByS</td>
<td>Single-Row-GroupBy</td>
<td>See “SingleRowGroupBy algorithm (GrByS)” on page 580.</td>
</tr>
<tr>
<td>GrBySSets</td>
<td>Sorted-GroupBy-Sets</td>
<td>See “SortedGroupBySets algorithm (GrBySSets)” on page 580.</td>
</tr>
<tr>
<td>HF</td>
<td>HashFilter</td>
<td>See “Hash filter algorithms (HF, HFP)” on page 586.</td>
</tr>
<tr>
<td>HFP</td>
<td>Parallel-HashFilter</td>
<td>See “Hash filter algorithms (HF, HFP)” on page 586.</td>
</tr>
<tr>
<td>HTS</td>
<td>HashTableScan</td>
<td>See “HashTableScan method (HTS)” on page 571.</td>
</tr>
<tr>
<td>IAH</td>
<td>HashIntersectAll</td>
<td>See “Intersect algorithms (IH, IM, IAH, IAM)” on page 581.</td>
</tr>
<tr>
<td>IAM</td>
<td>MergeIntersectAll</td>
<td>See “Intersect algorithms (IH, IM, IAH, IAM)” on page 581.</td>
</tr>
<tr>
<td>IH</td>
<td>HashIntersect</td>
<td>See “Intersect algorithms (IH, IM, IAH, IAM)” on page 581.</td>
</tr>
<tr>
<td>IM</td>
<td>MergeIntersect</td>
<td>See “Intersect algorithms (IH, IM, IAH, IAM)” on page 581.</td>
</tr>
<tr>
<td>Short text plan</td>
<td>Long text plan</td>
<td>Additional information</td>
</tr>
<tr>
<td>----------------</td>
<td>---------------</td>
<td>------------------------</td>
</tr>
<tr>
<td>IN</td>
<td>InList</td>
<td>See “InList algorithm (IN)” on page 586.</td>
</tr>
<tr>
<td>table-name&lt;index-name&gt;</td>
<td>Index-Scan, ParallelIndexScan</td>
<td>In a graphical plan, an index scan appears as an index name in a trapezoid. See “IndexScan method” on page 568.</td>
</tr>
<tr>
<td>INSENSITIVE</td>
<td>Insensitive</td>
<td>See “Intersect algorithms (IH, IM, IAH, IAM)” on page 581.</td>
</tr>
<tr>
<td>INSERT</td>
<td>Insert</td>
<td>Root node of an insert operation. See “INSERT statement” [SQL Anywhere Server - SQL Reference].</td>
</tr>
<tr>
<td>IO</td>
<td>IndexOnlyScan, ParallelIndexOnlyScan</td>
<td>See “IndexOnlyScan method (IO)” on page 569, and “ParallelIndexScan method” on page 570.</td>
</tr>
<tr>
<td>JH</td>
<td>HashJoin</td>
<td>See “HashJoin algorithms (JH, JHSP, JHFO, JHAP, JHO, JHPO)” on page 573.</td>
</tr>
<tr>
<td>JHS</td>
<td>HashSemijoin</td>
<td>See “HashSemijoin algorithm (JHS)” on page 574.</td>
</tr>
<tr>
<td>JHSP</td>
<td>Parallel-HashSemijoin</td>
<td>See “HashJoin algorithms (JH, JHSP, JHFO, JHAP, JHO, JHPO)” on page 573.</td>
</tr>
<tr>
<td>JHFO</td>
<td>Full Outer HashJoin</td>
<td>See “HashJoin algorithms (JH, JHSP, JHFO, JHAP, JHO, JHPO)” on page 573.</td>
</tr>
<tr>
<td>JHA</td>
<td>HashAnti-semijoin</td>
<td>See “HashAntisemijoin algorithm (JHA)” on page 575.</td>
</tr>
<tr>
<td>JHAP</td>
<td>Parallel-HashAnti-semijoin</td>
<td>See “HashJoin algorithms (JH, JHSP, JHFO, JHAP, JHO, JHPO)” on page 573.</td>
</tr>
<tr>
<td>JHO</td>
<td>Left Outer HashJoin</td>
<td>See “HashJoin algorithms (JH, JHSP, JHFO, JHAP, JHO, JHPO)” on page 573.</td>
</tr>
<tr>
<td>Short text plan</td>
<td>Long text plan</td>
<td>Additional information</td>
</tr>
<tr>
<td>----------------</td>
<td>---------------</td>
<td>------------------------</td>
</tr>
<tr>
<td>JHR</td>
<td>RecursiveHashJoin</td>
<td>See “RecursiveHashJoin algorithm (JHR)” on page 574.</td>
</tr>
<tr>
<td>JHRO</td>
<td>RecursiveLeftOuterHashJoin</td>
<td>See “RecursiveLeftOuterHashJoin algorithm (JHRO)” on page 574.</td>
</tr>
<tr>
<td>JM</td>
<td>MergeJoin</td>
<td>See “MergeJoin algorithms (JM, JMFO, JMO)” on page 576.</td>
</tr>
<tr>
<td>JMFO</td>
<td>Full Outer MergeJoin</td>
<td>See “MergeJoin algorithms (JM, JMFO, JMO)” on page 576.</td>
</tr>
<tr>
<td>JMO</td>
<td>Left Outer MergeJoin</td>
<td>See “MergeJoin algorithms (JM, JMFO, JMO)” on page 576.</td>
</tr>
<tr>
<td>JNL</td>
<td>Nested-LoopsJoin</td>
<td>See “NestedLoopsJoin algorithms (JNL, JNLFO, JNLO)” on page 576.</td>
</tr>
<tr>
<td>JNLA</td>
<td>Nested-LoopsAntisemijoin</td>
<td>See “NestedLoopsAntisemijoin algorithm (JNLA)” on page 577.</td>
</tr>
<tr>
<td>JNLFO</td>
<td>Full Outer Nested-LoopsJoin</td>
<td>See “NestedLoopsJoin algorithms (JNL, JNLFO, JNLO)” on page 576.</td>
</tr>
<tr>
<td>JNLO</td>
<td>Left Outer Nested-LoopsJoin</td>
<td>See “NestedLoopsJoin algorithms (JNL, JNLFO, JNLO)” on page 576.</td>
</tr>
<tr>
<td>JNLS</td>
<td>Nested-LoopsSemijoin</td>
<td>See “NestedLoopsSemijoin algorithm (JNLS)” on page 576.</td>
</tr>
<tr>
<td>KEYSET</td>
<td>Keyset</td>
<td>Indicates a keyset-driven cursor. See “SQL Anywhere cursors” [SQL Anywhere Server - Programming].</td>
</tr>
<tr>
<td>LOAD</td>
<td>Load</td>
<td>Root node of a load operation. See “LOAD TABLE statement” [SQL Anywhere Server - SQL Reference].</td>
</tr>
<tr>
<td>Short text plan</td>
<td>Long text plan</td>
<td>Additional information</td>
</tr>
<tr>
<td>----------------</td>
<td>---------------------------------------------------</td>
<td>----------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>MultiIdx</td>
<td>Multi-pleIndexScan</td>
<td>See “MultipleIndexScan method (MultiIdx)” on page 569.</td>
</tr>
<tr>
<td>OpenString</td>
<td>Open-String</td>
<td>See “OpenString algorithm (OpenString)” on page 587.</td>
</tr>
<tr>
<td>Optimization Time</td>
<td></td>
<td>The total time spent by the optimizer during all enumeration processes for a given statement.</td>
</tr>
<tr>
<td>PC</td>
<td>ProcCall</td>
<td>Procedure call (table function). See “ProcCall algorithm (PC)” on page 587.</td>
</tr>
<tr>
<td>PreFilter</td>
<td>PreFilter</td>
<td>See “Filter algorithms (Filter, PreFilter)” on page 585.</td>
</tr>
<tr>
<td>RL</td>
<td>RowLimit</td>
<td>See “RowLimit algorithm (RL)” on page 588.</td>
</tr>
<tr>
<td>ROWID</td>
<td>RowIdS-can</td>
<td>In a graphical plan, a row ID scan appears as a table name in a rectangle. See “RowIdScan method (ROWID)” on page 571.</td>
</tr>
<tr>
<td>ROWS</td>
<td>RowConstructor</td>
<td>See “RowConstructor algorithm (ROWS)” on page 588.</td>
</tr>
<tr>
<td>RR</td>
<td>RowReplicate</td>
<td>See “RowReplicate algorithm (RR)” on page 582.</td>
</tr>
<tr>
<td>RT</td>
<td>RecursiveTable</td>
<td>See “RecursiveTable algorithm (RT)” on page 582.</td>
</tr>
<tr>
<td>RU</td>
<td>RecursiveUnion</td>
<td>See “RecursiveUnion algorithm (RU)” on page 582.</td>
</tr>
<tr>
<td>SELECT</td>
<td>Select</td>
<td>Root node of a SELECT operation. See “SELECT statement” [SQL Anywhere Server - SQL Reference].</td>
</tr>
<tr>
<td>seq</td>
<td>TableS-can, ParallelTableS-can</td>
<td>In a graphical plan, table scans appear as a table name in a rectangle. See “TableScan method (seq)” on page 570, and “ParallelTableScan method” on page 571.</td>
</tr>
<tr>
<td>Sort</td>
<td>Sort</td>
<td>Indexed or merge sort. See “Sort algorithm (Sort)” on page 582.</td>
</tr>
<tr>
<td>SrtN</td>
<td>SortTopN</td>
<td>See “SortTopN algorithm (SrtN)” on page 583.</td>
</tr>
<tr>
<td>TermBreak</td>
<td>TermBreak</td>
<td>The full text search termbreaker algorithm. See “Alter a text index” on page 338.</td>
</tr>
</tbody>
</table>
### Short text

<table>
<thead>
<tr>
<th>Plan</th>
<th>Long text plan</th>
<th>Additional information</th>
</tr>
</thead>
<tbody>
<tr>
<td>UA</td>
<td>UnionAll</td>
<td>See “UnionAll algorithm (UA)” on page 582.</td>
</tr>
<tr>
<td>UPDATE</td>
<td>Update</td>
<td>The root node of an UPDATE operation. See “UPDATE statement” [SQL Anywhere Server - SQL Reference].</td>
</tr>
<tr>
<td>Window</td>
<td>Window</td>
<td>See “Window algorithm (Window)” on page 588.</td>
</tr>
<tr>
<td>Work</td>
<td>Work table</td>
<td>An internal node that represents an intermediate result.</td>
</tr>
</tbody>
</table>

### Common statistics used in the plan

The following statistics are actual, measured amounts.

<table>
<thead>
<tr>
<th>Statistic</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invocations</td>
<td>Number of times a row was requested from the sub tree.</td>
</tr>
<tr>
<td>RowsReturned</td>
<td>Number of rows returned for the current node.</td>
</tr>
<tr>
<td>RunTime</td>
<td>Time required for execution of the sub-tree, including time for children.</td>
</tr>
<tr>
<td>CacheHits</td>
<td>Number of successful reads of the cache.</td>
</tr>
<tr>
<td>CacheRead</td>
<td>Number of database pages that have been looked up in the cache.</td>
</tr>
<tr>
<td>CacheReadTable</td>
<td>Number of table pages that have been read from the cache.</td>
</tr>
<tr>
<td>CacheReadIndLeaf</td>
<td>Number of index leaf pages that have been read from the cache.</td>
</tr>
<tr>
<td>CacheReadIndInt</td>
<td>Number of index internal node pages that have been read from the cache.</td>
</tr>
<tr>
<td>DiskRead</td>
<td>Number of pages that have been read from disk.</td>
</tr>
<tr>
<td>DiskReadTable</td>
<td>Number of table pages that have been read from disk.</td>
</tr>
<tr>
<td>DiskReadIndLeaf</td>
<td>Number of index leaf pages that have been read from disk.</td>
</tr>
<tr>
<td>DiskReadIndInt</td>
<td>Number of index internal node pages that have been read from disk.</td>
</tr>
<tr>
<td>DiskWrite</td>
<td>Number of pages that have been written to disk (work table pages or modified table pages).</td>
</tr>
<tr>
<td>IndAdd</td>
<td>Number of entries that have been added to indexes.</td>
</tr>
<tr>
<td>IndLookup</td>
<td>Number of entries that have been looked up in indexes.</td>
</tr>
<tr>
<td>Statistic</td>
<td>Explanation</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>FullCompare</td>
<td>Number of comparisons that have been performed beyond the hash value in an index.</td>
</tr>
</tbody>
</table>

**Common estimates used in the plan**

<table>
<thead>
<tr>
<th>Statistic</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>EstRowCount</td>
<td>Estimated number of rows that the node will return each time it is invoked.</td>
</tr>
<tr>
<td>AvgRowCount</td>
<td>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.</td>
</tr>
<tr>
<td>EstRunTime</td>
<td>Estimated time required for execution (sum of EstDiskReadTime, EstDiskWriteTime, and EstCpuTime).</td>
</tr>
<tr>
<td>AvgRunTime</td>
<td>Average time required for execution (measured).</td>
</tr>
<tr>
<td>EstDiskReads</td>
<td>Estimated number of read operations from the disk.</td>
</tr>
<tr>
<td>AvgDiskReads</td>
<td>Average number of read operations from the disk (measured).</td>
</tr>
<tr>
<td>EstDiskWrites</td>
<td>Estimated number of write operations to the disk.</td>
</tr>
<tr>
<td>AvgDiskWrites</td>
<td>Average number of write operations to the disk (measured).</td>
</tr>
<tr>
<td>EstDiskReadTime</td>
<td>Estimated time required for reading rows from the disk.</td>
</tr>
<tr>
<td>EstDiskWriteTime</td>
<td>Estimated time required for writing rows to the disk.</td>
</tr>
<tr>
<td>EstCpuTime</td>
<td>Estimated processor time required for execution.</td>
</tr>
</tbody>
</table>

**Items in the plan related to SELECT, INSERT, UPDATE, and DELETE**

<table>
<thead>
<tr>
<th>Item</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimization Goal</td>
<td>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 [database]” [SQL Anywhere Server - Database Administration].</td>
</tr>
<tr>
<td>Optimization workload</td>
<td>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 [database]” [SQL Anywhere Server - Database Administration].</td>
</tr>
</tbody>
</table>
### ANSI update constraints

Controls the range of updates that are permitted (options are Off, Cursors, and Strict). See “ansi_update_constraints option [compatibility]” ([SQL Anywhere Server - Database Administration](#)).

### Optimization level

Reserved.

### Select list

List of expressions selected by the query.

### 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 550, and “Restrictions on materialized views” on page 71.

### Items in the plan related to locks

<table>
<thead>
<tr>
<th>Item</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Locked tables</td>
<td>List of all locked tables and their isolation levels.</td>
</tr>
</tbody>
</table>
### Items in the plan related to scans

<table>
<thead>
<tr>
<th>Item</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table name</td>
<td>Actual name of the table.</td>
</tr>
<tr>
<td>Correlation name</td>
<td>Alias for the table.</td>
</tr>
<tr>
<td>Estimated rows</td>
<td>Estimated number of rows in the table.</td>
</tr>
<tr>
<td>Estimated pages</td>
<td>Estimated number of pages in the table.</td>
</tr>
<tr>
<td>Estimated row size</td>
<td>Estimated row size for the table.</td>
</tr>
<tr>
<td>Page maps</td>
<td>YES when a page map is used to read multiple pages.</td>
</tr>
</tbody>
</table>

### Items in the plan related to index scans

<table>
<thead>
<tr>
<th>Item</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index name</td>
<td>Name of the index.</td>
</tr>
<tr>
<td>Key type</td>
<td>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.</td>
</tr>
<tr>
<td>Depth</td>
<td>Height of the index. See “Table and page sizes” on page 616.</td>
</tr>
<tr>
<td>Estimated leaf pages</td>
<td>Estimated number of leaf pages.</td>
</tr>
<tr>
<td>Cardinality</td>
<td>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.</td>
</tr>
<tr>
<td>Selectivity</td>
<td>Estimated number of rows that match the range bounds.</td>
</tr>
<tr>
<td>Direction</td>
<td>FORWARD or BACKWARD.</td>
</tr>
<tr>
<td>Range bounds</td>
<td>Range bounds are shown as a list (col_name=value) or col_name IN [low, high].</td>
</tr>
</tbody>
</table>

### Items in the plan related to joins, filter, and pre-filter

<table>
<thead>
<tr>
<th>Item</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Predicate</td>
<td>Search condition that is evaluated in this node, along with selectivity estimates and measurement. See “Viewing selectivity in the graphical plan” on page 607</td>
</tr>
</tbody>
</table>
### Items in the plan related to hash filter

<table>
<thead>
<tr>
<th>Item</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Build values</td>
<td>Estimated number of distinct values in the input.</td>
</tr>
<tr>
<td>Probe values</td>
<td>Estimated number of distinct values in the input when checking the predicate.</td>
</tr>
<tr>
<td>Bits</td>
<td>Number of bits selected to build the hash map.</td>
</tr>
<tr>
<td>Pages</td>
<td>Number of pages required to store the hash map.</td>
</tr>
</tbody>
</table>

### Items in the plan related to Union

<table>
<thead>
<tr>
<th>Item</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Union List</td>
<td>Columns involved in a UNION statement.</td>
</tr>
</tbody>
</table>

### Items in the plan related to GROUP BY

<table>
<thead>
<tr>
<th>Item</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aggregates</td>
<td>All the aggregate functions.</td>
</tr>
<tr>
<td>Group-by list</td>
<td>All the columns in the group by clause.</td>
</tr>
</tbody>
</table>

### Items in the plan related to DISTINCT

<table>
<thead>
<tr>
<th>Item</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distinct list</td>
<td>All the columns in the distinct clause.</td>
</tr>
</tbody>
</table>

### Items in the plan related to IN LIST

<table>
<thead>
<tr>
<th>Item</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>In List</td>
<td>All the expressions in the specified set.</td>
</tr>
<tr>
<td>Expression SQL</td>
<td>Expressions to compare to the list.</td>
</tr>
</tbody>
</table>

### Items in the plan related to SORT

<table>
<thead>
<tr>
<th>Item</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Order-by</td>
<td>List of all expressions to sort by.</td>
</tr>
</tbody>
</table>
Items in the plan related to row limits

<table>
<thead>
<tr>
<th>Item</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Row limit count</td>
<td>Maximum number of rows returned as specified by FIRST or TOP n.</td>
</tr>
</tbody>
</table>

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 601.

There is also a graphical version of the plan. See “Reading graphical plans” on page 602.

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 JM 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[ HF[ Customers<CustomersKey> ] JM
SalesOrders<FK_CustomerID_ID> ] ]
```

For more information about code words used in the plan, see “Abbreviations in the plan” on page 589.

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 “Abbreviations in the plan” on page 589.

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: 1.040539E-5] 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 plan indicates that the results are sorted, and that a Merge 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 HashFilter indicates that a hash filter is being built on the right-hand side of the Merge Join. The IndexScan lines indicate that the Customers and SalesOrders tables are accessed via indexes CustomersKey and FK_CustomerID_ID respectively.

```sql
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: 1.040539E-5 ]
  ( WorkTable
    ( Sort
      ( MergeJoin [ ( ((Customers.Country LIKE 'Canada' : 100% Computed)
          AND (Customers.Country = 'Canada' : 7.936508209% Statistics))
          OR ((SalesOrders.Region LIKE 'Eastern' : 100% Computed)
          AND (SalesOrders.Region = 'Eastern' : 5% Guess)) ] : 100% Guess ]
      ( 1 HashFilter
        ( IndexScan Customers CustomersKey )
      )
      ( IndexScan SalesOrders
        FK_CustomerID_ID[ hash( SalesOrders.CustomerID )
        in hashmap( Customers.ID ) : 100% Guess ]
      )
    )
  )
)
```

For more information about the abbreviations used in a plan, see “Abbreviations in the plan” on page 589.

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 609.

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 610. 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 188.

For more information about text plans, see “Reading text plans” on page 600.

**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 sufficient information to correctly estimate the query's cost, which may result in 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. In some cases 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 611, and “Optimizer Statistics field descriptions” on page 612.

**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 simple query, so that the optimizer
is bypassed. For more information about simple queries, see “How the optimizer works” on page 541, 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 607.

  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 of
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 “Increase the cache size” on page 241.

**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.
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. Hence, 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 “Reducing table fragmentation” on page 246.

For more information about code words used in the plan, see “Abbreviations in the plan” on page 589.

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 “Abbreviations in the plan” on page 589.

**Note**

If a query is recognized as a simple 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 541.
See also

- “Reading graphical plans” on page 602
- “Viewing graphical plans” on page 610
- “Reading execution plans” on page 589
- “Node Statistics field descriptions” on page 611
- “Optimizer Statistics field descriptions” on page 612

**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 simple queries that qualify for optimization bypass. For more information about simple queries, see “How the optimizer works” on page 541.

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

\[ \text{Departments.DepartmentName} = 'Sales' : 20\% \text{ Column; true 1/5 20\%} \]

This can be read as follows:

- \text{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 \text{ESTIMATE} function. For more information, see “\text{ESTIMATE function [Miscellaneous]}” [\text{SQL Anywhere Server - SQL Reference}].

- Column is the source of the estimate. This is the same output as is provided by the \text{ESTIMATE_SOURCE} function. For a complete list of the possible sources of selectivity estimates, see “\text{ESTIMATE_SOURCE function [Miscellaneous]}” [\text{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, hence 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, it is possible that the incorrect estimates are causing a significant problem in query performance. Collecting statistics on the predicate may improve performance by giving the optimizer better information on which to base its choices.

**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. To access text plans, see “Reading text plans” on page 600.

Viewing graphical plans

To view a graphical plan (Interactive SQL)

1. Start Interactive SQL and connect to the SQL Anywhere database.
2. Type a statement in the SQL Statements pane.
4. Select a Statistics level, a Cursor type and an Update status and click Get Plan.

To view a graphical plan (SQL)

Use the GRAPHICAL_PLAN function to view a graphical plan in XML format, as a string.
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:
- “Abbreviations in the plan” on page 589

### Node Statistics field descriptions

Below are descriptions of the fields displayed in the Node Statistics section of a graphical plan.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CacheHits</td>
<td>The total number of cache read requests by this operator which were satisfied by the buffer pool, and hence did not require a disk read operation for that page.</td>
</tr>
<tr>
<td>CacheRead</td>
<td>Total number of attempts made by this operator to read a page of the database file, typically for table and/or index pages.</td>
</tr>
<tr>
<td>CPUTime</td>
<td>The CPU time incurred by the processing algorithm represented by this node.</td>
</tr>
<tr>
<td>DiskRead</td>
<td>The cumulative number of pages that have been read from disk as a result of this node's processing.</td>
</tr>
<tr>
<td>DiskRead-Time</td>
<td>The cumulative elapsed time required to perform disk reads for database pages required by this node for processing.</td>
</tr>
<tr>
<td>DiskWrite</td>
<td>The commutative number of pages that have been written to disk as a result of this node's processing.</td>
</tr>
<tr>
<td>DiskWrite-Time</td>
<td>The cumulative elapsed time required to perform disk writes for database pages as required by this node's processing algorithm.</td>
</tr>
<tr>
<td>FirstRow-RunTime</td>
<td>The FirstRowRunTime value is the actual elapsed time required to produce the first row of the intermediate result of this node.</td>
</tr>
<tr>
<td>Invocations</td>
<td>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.</td>
</tr>
</tbody>
</table>
| PercentTo-
  talCost     | The RunTime spent computing the result within this particular node, expressed as a percentage of the total RunTime for the statement.                                                                           |
### 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 server and about the optimization of the selected statement.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Query-MemMax-Useful</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>RowsReturned</strong></td>
<td>The number of rows returned to the parent node as a result of processing the request. <strong>RowsReturned</strong> 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 <strong>RowsReturned</strong> value to be smaller or larger than the number of rows in the table. <strong>RowsReturned</strong> are smaller if the parent node fails to request all of the table's rows in computing the final result. <strong>RowsReturned</strong> 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.</td>
</tr>
<tr>
<td><strong>RunTime</strong></td>
<td>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 <strong>RunTime</strong> depends on the statistics section in which it appears. In Node Statistics, <strong>RunTime</strong> 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 <strong>RunTime</strong> 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.</td>
</tr>
<tr>
<td>Field</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Costed Best Plans</td>
<td>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 prior to 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, <strong>Costed Best Plans</strong> represents the cumulative count. See “How the optimizer works” on page 541. If the values for <strong>Costed Best Plans</strong>, <strong>Costed Plans</strong>, and <strong>Optimization Time</strong> 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 “Queries that bypass optimization” on page 527.</td>
</tr>
<tr>
<td>Costed Plans</td>
<td>The number of different access plans considered by the optimizer for this request whose costs were partially or fully estimated. As with <strong>Costed Best Plans</strong>, smaller values normally indicate faster optimization times and larger values indicate more complex SQL queries. If the values for <strong>Costed Best Plans</strong>, <strong>Costed Plans</strong>, and <strong>Optimization Time</strong> 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 “Queries that bypass optimization” on page 527.</td>
</tr>
<tr>
<td>Optimization Time</td>
<td>The elapsed time spent optimizing the statement. If the values for <strong>Costed Best Plans</strong>, <strong>Costed Plans</strong>, and <strong>Optimization Time</strong> 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 “Queries that bypass optimization” on page 527.</td>
</tr>
<tr>
<td>Estimated Cache Pages</td>
<td>Estimated current cache size available for processing the statement. For the purposes of reducing inefficient access plans, the optimizer assumes that one-half of the current cache size is available for processing the selected statement.</td>
</tr>
<tr>
<td>CurrentCacheSize</td>
<td>The database server's cache size in kilobytes at the time of optimization.</td>
</tr>
</tbody>
</table>
| 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 547.
<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>QueryMemNeeds-Grant</td>
<td>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 547.</td>
</tr>
<tr>
<td>QueryMemLikely-Grant</td>
<td>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 server's multiprogramming level, and the number of concurrently-executing memory-intensive requests. See “The memory governor” on page 547.</td>
</tr>
<tr>
<td>QueryMemPages</td>
<td>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 547.</td>
</tr>
<tr>
<td>QueryMemActive-Max</td>
<td>The maximum number of tasks that can actively use query memory at any particular time. See “The memory governor” on page 547.</td>
</tr>
<tr>
<td>QueryMemActiveEst</td>
<td>The server's estimate of the steady state average of the number of tasks actively using query memory. See “The memory governor” on page 547.</td>
</tr>
<tr>
<td>isolation_level</td>
<td>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 [compatibility]” [SQL Anywhere Server - Database Administration].</td>
</tr>
<tr>
<td>optimization_goal</td>
<td>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 [database]” [SQL Anywhere Server - Database Administration].</td>
</tr>
<tr>
<td>optimization_level</td>
<td>Controls amount of effort made by the query optimizer to find an access plan. See “optimization_level option [database]” [SQL Anywhere Server - Database Administration].</td>
</tr>
<tr>
<td>optimization_workload</td>
<td>The Mixed or OLAP value of the optimization_workload setting. See “optimization_workload option [database]” [SQL Anywhere Server - Database Administration].</td>
</tr>
<tr>
<td>max_query_tasks</td>
<td>Maximum number of tasks that may be used by a parallel execution plan for a single query. See “max_query_tasks option [database]” [SQL Anywhere Server - Database Administration].</td>
</tr>
<tr>
<td>user_estimates</td>
<td>Controls whether to respect or ignore user estimates that are specified in individual predicates in the query text. See “user_estimates option [database]” [SQL Anywhere Server - Database Administration].</td>
</tr>
</tbody>
</table>
Physical data organization and access

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 sufficient 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. Thus, 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, however, if you delete a number of rows and don't 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 particularly 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. Consequently, 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 (as well as other behavior aspects) when choosing a page size. Then, choose the smallest page size that gives satisfactory results. It is particularly 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 250
- “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 of 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 194.

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 and column order must be identical, as well as the ordering (ascending, descending) of data for each column.

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, “SYSSIDX 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, as well as 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 619.

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 of 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:

```sql
SELECT tab.table_name, idx.table_id, phys.phys_index_id, COUNT(*)
FROM SYSSIDX 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>
<thead>
<tr>
<th>table_name</th>
<th>table_id</th>
<th>phys_index_id</th>
<th>COUNT(*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISYSCHECK</td>
<td>57</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>ISYSCOLSTAT</td>
<td>50</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>ISYSFKEY</td>
<td>6</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>ISYSSSOURCE</td>
<td>58</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>MAINLIST</td>
<td>94</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>MAINLIST</td>
<td>94</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

The number of rows for each table indicates the number of shared physical indexes for the tables. In this example, all of 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

- “Rebuilding indexes” on page 90
- “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 or not 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 621.

- **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 194.

**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 86.

**Improving index performance**

If your index is not performing as well 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 595.

For more information about the FullCompare function, see “Database-level properties” [SQL Anywhere Server - Database Administration].

Index structure and index fan-out

Indexes are organized in a number of 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 616.

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:

```sql
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 of 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.

```sql
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.

<table>
<thead>
<tr>
<th>Index columns</th>
<th>Optimizable ORDER BY queries</th>
<th>Not optimizable ORDER BY queries</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASC, ASC</td>
<td>ASC, ASC or DESC, DESC</td>
<td>ASC, DESC or DESC, ASC</td>
</tr>
<tr>
<td>ASC, DESC</td>
<td>ASC, DESC or DESC, ASC</td>
<td>ASC, ASC or DESC, DESC</td>
</tr>
<tr>
<td>DESC, ASC</td>
<td>DESC, ASC or ASC, DESC</td>
<td>ASC, ASC or DESC, DESC</td>
</tr>
<tr>
<td>DESC, DESC</td>
<td>DESC, DESC or ASC, ASC</td>
<td>ASC, DESC or DESC, ASC</td>
</tr>
</tbody>
</table>

An index with more than two columns follows the same general rule as above. For example, suppose you
have the following index:

```sql
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 142.

- **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.

Types of index

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.

SQL Anywhere implements B-link indexes.

See also

- “Using clustered indexes” on page 87
- “ALTER INDEX statement” [SQL Anywhere Server - SQL Reference]
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.

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.
Part V. SQL Dialects and Compatibility

This part describes Transact-SQL compatibility and those features of SQL Anywhere that are not commonly found in other SQL implementations.
CHAPTER 15

Other SQL dialects

Contents

Introduction to SQL Anywhere compliance ............................................................... 628
Testing SQL compliance using the SQL Flagger ...................................................... 629
Features not found in other SQL implementations .................................................... 631
Transact-SQL Compatibility ....................................................................................... 633
Adaptive Server architectures ................................................................................... 636
Configuring databases for Transact-SQL compatibility ............................................. 641
Writing compatible SQL statements .......................................................................... 647
Transact-SQL procedure language overview ............................................................... 651
Automatic translation of stored procedures .............................................................. 653
Returning result sets from Transact-SQL procedures ............................................... 654
Variables in Transact-SQL procedures ..................................................................... 655
Error handling in Transact-SQL procedures .............................................................. 656
Introduction to SQL Anywhere compliance

SQL Anywhere complies completely 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-2003 core specifications. 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 non-compliant with respect to a specific ISO/ANSI SQL standard, or are not supported by UltraLite. This functionality is called the SQL Flagger, and is part of the SQL/1999 and SQL/2003 ISO/ANSI SQL standards. 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.

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/2003 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]
- “Running the SQL preprocessor” [SQL Anywhere Server - Programming]
- “INSERT statement” [SQL Anywhere Server - SQL Reference]
- “Key joins” on page 406

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 packages from the ANSI 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 [compatibility]” [*SQL Anywhere Server - Database Administration*] and “sql_flagger_warning_level option [compatibility]” [*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 827

### 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/2003 International Standard. The SQL Flagger can determine the compliance of SQL constructs with respect to the following ANSI SQL standards:

- 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

<table>
<thead>
<tr>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL Flagger support for SQL/1992 (all levels) is deprecated.</td>
</tr>
</tbody>
</table>

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 thus 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 reference” [*UltraLite - Database Management and Reference*]
- “SET OPTION statement” [*SQL Anywhere Server - SQL Reference*]
Features not found in other SQL implementations

The following features of the SQL supported by SQL Anywhere are not found in many other SQL implementations.

Dates

SQL Anywhere has date, time and timestamp types that includes 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.

Also, many functions are provided for manipulating dates and times. See “SQL functions” [SQL Anywhere Server - SQL Reference] for a description of these.

Integrity

SQL Anywhere supports both entity and referential integrity. This has been implemented via the following two extensions to the CREATE TABLE and ALTER TABLE commands.

```
PRIMARY KEY ( column-name, ... )
[NOT NULL] FOREIGN KEY [role-name]
[ (column-name, ...)]
REFERENCES table-name [ (column-name, ...)]
[ CHECK ON COMMIT ]
```

The PRIMARY KEY clause declares the primary key for the relation. SQL Anywhere then enforces the uniqueness of the primary key, and ensure that no column in the primary key contains the NULL value.

The FOREIGN KEY clause defines a relationship between this table and another table. This relationship is represented by a column (or columns) in this table which must contain values in the primary key of another table. The system then ensures referential integrity for these columns - whenever these columns are modified or a row is inserted into this table, these columns are checked to ensure that either one or more is NULL or the values match the corresponding columns in the primary key for some row of the other table. For more information, see “CREATE TABLE statement” [SQL Anywhere Server - SQL Reference].

Joins

SQL Anywhere allows automatic joins between tables. In addition to the NATURAL and OUTER join operators supported in other implementations, SQL Anywhere allows KEY joins between tables based on foreign key relationships. This reduces the complexity of the WHERE clause when performing joins.

Updates

SQL Anywhere allows more than one table to be referenced by the UPDATE command. Views defined on more than one table can also be updated. Many SQL implementations do not allow updates on joined tables.
Altering tables

The ALTER TABLE command has been extended. In addition to changes for entity and referential integrity, the following types of alterations are allowed:

- ADD column data-type
- ALTER column data-type
- DELETE column
- RENAME new-table-name
- RENAME old-column TO new-column

The ALTER clause can be used to change the maximum length of a character column as well as convert from one data type to another. See “ALTER TABLE statement” [SQL Anywhere Server - SQL Reference].

Subqueries where expressions are allowed

SQL Anywhere allows subqueries to appear wherever expressions are allowed. Many SQL implementations only allow subqueries on the right side of a comparison operator. For example, the following command is valid in SQL Anywhere but not valid in most other SQL implementations.

```sql
SELECT Surname,
       BirthDate,
       (  SELECT DepartmentName
           FROM Departments
           WHERE EmployeeID = Employees.EmployeeID
           AND DepartmentID = 200 )
FROM Employees;
```

Additional functions

SQL Anywhere supports several functions not in the ANSI SQL definition. See “SQL functions” [SQL Anywhere Server - SQL Reference] for a full list of available functions.

Cursors

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.

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 do not allow this.
Transact-SQL Compatibility

SQL Anywhere supports a large subset of Transact-SQL, the dialect of SQL supported by Sybase Adaptive Server Enterprise. This chapter 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. In this chapter, those dialects are called Transact-SQL and Watcom-SQL.

- 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 very high percentage of Transact-SQL language elements, functions, and statements for working with existing data. For example, SQL Anywhere supports all of the numeric functions, all but one of the string functions, all aggregate functions, and all date and time functions. As another example, SQL Anywhere supports extended DELETE and UPDATE statements using joins.

Further, SQL Anywhere supports a very 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 of the more common administrative tasks.

This chapter 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:
- control statements CASE, LOOP, and FOR
- SQL Anywhere versions of IF and WHILE
- CALL 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 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 overview” [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, and each user requires a login ID and password to the server as well as 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].

<table>
<thead>
<tr>
<th>System procedure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sp_addlogin</td>
<td>In Adaptive Server Enterprise, this adds a user to the server. In SQL Anywhere, this adds a user to a database.</td>
</tr>
<tr>
<td>sp_adduser</td>
<td>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.</td>
</tr>
<tr>
<td>sp_addgroup</td>
<td>Adds a group to a database.</td>
</tr>
<tr>
<td>sp_changegroup</td>
<td>Adds a user to a group, or moves a user from one group to another.</td>
</tr>
<tr>
<td>sp_droplogin</td>
<td>In Adaptive Server Enterprise, removes a user from the server. In SQL Anywhere, removes a user from the database.</td>
</tr>
<tr>
<td>sp_dropuser</td>
<td>Removes a user from the database.</td>
</tr>
<tr>
<td>sp_dropgroup</td>
<td>Removes a group from the database.</td>
</tr>
</tbody>
</table>
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:

```sql
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 639.

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. Start Sybase Central.
2. Choose Tools » SQL Anywhere 11 » Create Database.
3. Follow the instructions in the wizard.
4. When you see the button, Emulate Adaptive Server Enterprise, click it and then 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:

\[
\begin{align*}
&\text{CREATE DATABASE 'dbname.db' } \\
&\text{ASE COMPATIBLE} \\
&\text{CASE RESPECT} \\
&\text{BLANK Padding ON;}
\end{align*}
\]

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/2003 ISO standard.
To make Adaptive Server Enterprise behave in a SQL/2003-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/2003 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 [compatibility]” [SQL Anywhere Server - Database Administration].

For SQL/2003 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 [compatibility]” [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 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:

```sql
CREATE TABLE tablename (  
  column_1 INTEGER,
)```
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.

A typical UPDATE statement using the tsequal function looks like this:

```
UPDATE publishers
SET City = 'Springfield'
WHERE pub_id = '0736'
AND TSEQUAL(timestamp, '2005/10/25 11:08:34.173226');
```

The first argument to the tsequal function is the name of the special timestamp column; the second argument is the timestamp retrieved in the SELECT statement. In embedded SQL, the second argument is likely to be a host variable containing a TIMESTAMP value from a recent FETCH on the column.

### The special IDENTITY column

To create an IDENTITY column, use the following CREATE TABLE syntax:

```
CREATE TABLE table-name (  
```
column-name numeric(n,0) IDENTITY NOT NULL,

where \( n \) is large enough to hold the value of the maximum number of rows that may be inserted into the table.

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.

**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 non-compliant with respect to a specific ISO/ANSI SQL standard, or are not supported by UltraLite. This functionality is called the SQL Flagger. See “Testing SQL compliance using the SQL Flagger” on page 629.

General guidelines applicable to writing compatible SQL include:

- Spell out all of 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 642.

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 629
- “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 SELECT statement itself must be compatible.

This section explains compatible SELECT statement syntax, and assumes compatible data types, expressions, and search conditions. The examples assume the quoted_identifier setting is Off: the default Adaptive Server Enterprise setting, but not the default SQL Anywhere setting.

SQL Anywhere supports the following subset of the Transact-SQL SELECT statement:

**Syntax**

```
SELECT [ ALL | DISTINCT ] select-list
...[ INTO temporary-table-name ]
...[ FROM table-spec [ HOLDLOCK | NOHOLDLOCK ],
... table-spec [ HOLDLOCK | NOHOLDLOCK ], ... ]
...[ WHERE search-condition ]
...[ GROUP BY column-name, ... ]
...[ HAVING search-condition ]
[ ORDER BY { expression | integer } [ ASC | DESC ], ... ]
```

**Parameters**

```
select-list:
  table-name.*
| *
  expression
| alias-name = expression
```
expression as identifier
| expression as string

table-spec:
[ owner , ]table-name
...[ [ AS ] correlation-name ]
...[ ( INDEX index_name [ PREFETCH size ][ LRU | MRU ] ) ]

alias-name:
identifier | 'string' | "string"

SQL Anywhere does not support the following keywords and clauses of the Transact-SQL SELECT statement syntax:

- SHARED keyword
- COMPUTE clause
- FOR BROWSE clause
- FOR UPDATE clause
- GROUP BY ALL clause

Notes

- SQL Anywhere does not support the Transact-SQL extension to the GROUP BY clause allowing references to columns and expressions that are not used for creating groups. In Adaptive Server Enterprise, this extension produces summary reports.

- The performance parameters part of the table specification is parsed, but has no effect.

- The HOLDLOCK keyword is supported by SQL Anywhere. It makes a shared lock on a specified table or view more restrictive by holding it until the completion of a transaction (instead of releasing the shared lock as soon as the required data page is no longer needed, whether or not the transaction has been completed). For the purposes of the table for which the HOLDLOCK is specified, the query is performed at isolation level 3.

- 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 clauses of the SELECT statement syntax:
  - INTO host-variable-list
  - INTO variable-list
  - Parenthesized queries
Adaptive Server Enterprise uses join operators in the WHERE clause, rather than the FROM clause and the ON condition for joins.

See also
- “Testing SQL compliance using the SQL Flagger” on page 629
- “SELECT statement” [SQL Anywhere Server - SQL Reference]
- “OLAP support” on page 437

Compatibility of joins

In Transact-SQL, joins appear in the WHERE clause, using the following syntax:

```
start of select, update, insert, delete, or subquery
    FROM { table-list | view-list } WHERE [ NOT ]
        [ table-name | view-name ] column-name
    join-operator
        [ table-name | view-name ] column-name
        [ { AND | OR } [ NOT ]
        [ table-name | view-name ] column-name
    join-operator
        [ table-name | view-name ] column-name
...end of select, update, insert, delete, or subquery
```

The `join-operator` in the WHERE clause may be any of the comparison operators, or may be either of the following outer-join operators:

- *=  Left outer join operator
- =*     Right outer join operator

SQL Anywhere supports the Transact-SQL outer join operators as an alternative to the native SQL/2003 syntax. You cannot mix dialects within a query. This rule applies also to views used by a query—an outer-join query on a view must follow the dialect used by the view-defining query.

**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 373, and “FROM clause” [SQL Anywhere Server - SQL Reference].

For more information about Transact-SQL compatibility of joins, see “Transact-SQL outer joins (*= or =*)” on page 390.

See also
- “Testing SQL compliance using the SQL Flagger” on page 629
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/2003.

Transact-SQL stored procedure overview

Based on the ISO/ANSI draft standard, the SQL Anywhere stored procedure language 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. 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 818.

Row-level triggers are not part of the Transact-SQL compatibility features, and are discussed in “Using procedures, triggers, and batches” on page 805.

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** 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].

### 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. Functions returning information about SQL statements and enabling automatic translation of SQL statements include:

- **SQLDialect(statement)**  Returns Watcom-SQL or Transact-SQL.
- **WatcomSQL(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:

```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.

**To translate a stored procedure using Sybase Central**

1. Connect to a database using Sybase Central, either as owner of the procedure you want to change, or as a DBA user.
2. Open the **Procedures & Functions** folder.
3. Click the **SQL** tab in the right pane and then click in the editor.
4. From the **File** menu, choose one of the **Translate To** commands, depending on the dialect you want to use.

   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 as needed.
6. When finished, choose **File » Save** to save the translated version to the database. You can also export the text to a file for editing outside of Sybase Central.
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 Transact-SQL procedure

The following Transact-SQL procedure illustrates how Transact-SQL stored procedures returns result sets:

```sql
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 Watcom-SQL procedure

The following is the corresponding SQL Anywhere procedure:

```sql
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 836.
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:

```sql
CREATE PROCEDURE multiply
    @mult1 int,
    @mult2 int,
    @result int output
AS
    SELECT @result = @mult1 * @mult2;
```

This procedure can be called as follows:

```sql
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 648. 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.

```sql
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:

```sql
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:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Procedure executed without error</td>
</tr>
<tr>
<td>-1</td>
<td>Missing object</td>
</tr>
<tr>
<td>-2</td>
<td>Data type error</td>
</tr>
<tr>
<td>-3</td>
<td>Process was chosen as deadlock victim</td>
</tr>
<tr>
<td>-4</td>
<td>Permission error</td>
</tr>
<tr>
<td>-5</td>
<td>Syntax error</td>
</tr>
<tr>
<td>-6</td>
<td>Miscellaneous user error</td>
</tr>
<tr>
<td>-7</td>
<td>Resource error, such as out of space</td>
</tr>
<tr>
<td>-8</td>
<td>Non-fatal internal problem</td>
</tr>
<tr>
<td>-9</td>
<td>System limit was reached</td>
</tr>
</tbody>
</table>
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_raisererror 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_raisererror 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:

```sql
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 using these two clauses together.
Part VI. XML in the Database

This part describes how to use XML in the database.
CHAPTER 16

Using XML in the database

Contents

Introduction to XML ................................................................. 662
Storing XML documents in relational databases ...................... 663
Exporting relational data as XML ............................................. 664
Importing XML documents as relational data .......................... 665
Obtaining query results as XML ............................................... 672
Using SQL/XML to obtain query results as XML ...................... 690
Introduction to XML

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.

For more detailed information about XML, see http://www.w3.org/XML/.

XML and SQL Anywhere

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
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.

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:

\[
<\text{hat}>bowler</\text{hat}>
\]

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:

```sql
SELECT XMLFOREST( CAST( '<hat>bowler</hat>' AS XML ) AS product );
```

You get the following result:

\[
<\text{product}><\text{hat}>bowler</\text{hat}></\text{product}>
\]

However, if the query does not specify that the element content is of type XML, for example:

```sql
SELECT XMLFOREST( '<hat>bowler</hat>' AS product );
```

In this case, the less than and greater than signs are replaced with entity references as follows:

\[
<\text{product}>\&lt;\text{hat}&gt;bowler\&lt;/\text{hat}&gt;</\text{product}>
\]

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 674.

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:

```xml
<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:

```sql
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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Quantity</th>
<th>Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tee Shirt</td>
<td>54</td>
<td>Orange</td>
</tr>
<tr>
<td>Baseball Cap</td>
<td>112</td>
<td>Black</td>
</tr>
</tbody>
</table>

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 677.

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>
  </Employees>
  <SalesOrders>
    <ID>2001</ID>
    <CustomerID>101</CustomerID>
    <OrderDate>2000-03-16</OrderDate>
    <FinancialCode>r1</FinancialCode>
    <Region>Eastern</Region>
    <SalesRepresentative>299</SalesRepresentative>
  </SalesOrders>
  <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>
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. Note that metaproperty names are case sensitive, thus 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.

<table>
<thead>
<tr>
<th>ID</th>
<th>parent</th>
<th>name</th>
<th>text</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>(NULL)</td>
<td>root</td>
<td>(NULL)</td>
</tr>
<tr>
<td>16</td>
<td>5</td>
<td>Employees</td>
<td>(NULL)</td>
</tr>
<tr>
<td>28</td>
<td>16</td>
<td>EmployeeID</td>
<td>299</td>
</tr>
<tr>
<td>55</td>
<td>16</td>
<td>ManagerID</td>
<td>902</td>
</tr>
<tr>
<td>79</td>
<td>16</td>
<td>Surname</td>
<td>Overbey</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

**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:

<table>
<thead>
<tr>
<th>ManagerID</th>
<th>Reports</th>
</tr>
</thead>
</table>
| 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>       |
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:

```sql
SELECT ManagerID, EmployeeID
FROM test, LATERAL( openxml( test.Reports, '//e' )
WITH (EmployeeID INT '.') ) DerivedTable
ORDER BY ManagerID, EmployeeID;
```

This query generates the following result:

<table>
<thead>
<tr>
<th>ManagerID</th>
<th>EmployeeID</th>
</tr>
</thead>
<tbody>
<tr>
<td>501</td>
<td>102</td>
</tr>
<tr>
<td>501</td>
<td>105</td>
</tr>
<tr>
<td>501</td>
<td>160</td>
</tr>
<tr>
<td>501</td>
<td>243</td>
</tr>
</tbody>
</table>

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 S SqlDataAdapter object” [SQL Anywhere Server - Programming].
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 672 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 690.

The FOR XML clause and the SQL/XML functions supported by SQL Anywhere give you two alternatives for generating XML from your relational data. In most cases, you can use either one to generate the same XML.

For example, this query uses FOR XML AUTO to generate XML:

```sql
SELECT ID, Name
FROM Products
WHERE Color='black'
FOR XML AUTO;
```

The following query uses the XMLELEMENT function to generate XML:

```sql
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):

```xml
<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 675.

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 677.

- **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 680.

The following sections describe the behavior of all three modes of the FOR XML clause regarding binary data, NULL values, and invalid XML names, as well as providing examples of how the FOR XML clause can be used.

### 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.

```sql
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 [database]” [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:

  ```sql
  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:

  ```sql
  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:

  ```sql
  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; D</DepartmentName>
</row>
<row>
  <EmployeeID>105</EmployeeID>
  <DepartmentName>R &amp; D</DepartmentName>
</row>
<row>
  <EmployeeID>160</EmployeeID>
  <DepartmentName>R &amp; D</DepartmentName>
</row>
<row>
  <EmployeeID>243</EmployeeID>
  <DepartmentName>R &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 673.

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:

```sql
SELECT ID AS product_ID
FROM Products
WHERE Color='black'
FOR XML RAW;
```

This query gives the following result:

```xml
<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:

```sql
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:

```sql
SELECT Employees.EmployeeID, Departments.DepartmentName
FROM Employees JOIN Departments
ON Employees.DepartmentID=Departments.DepartmentID
FOR XML RAW;
```

The following XML document is returned:

```xml
<row EmployeeID="102" DepartmentName="R &amp; D"/>
<row EmployeeID="105" DepartmentName="R &amp; D"/>
<row EmployeeID="160" DepartmentName="R &amp; D"/>
<row EmployeeID="243" DepartmentName="R &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**

```sql
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,

```sql
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.

```xml
<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 [database]” [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,

```sql
SELECT XMLELEMENT( NAME root, 
  (SELECT EmployeeID AS id, GivenName AS name 
   FROM Employees FOR XML AUTO ) )
```

---

Using XML in the database

Copyright © 2008, iAnywhere Solutions, Inc. - SQL Anywhere 11.0.0
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:

```sql
SELECT ID AS product_ID
FROM Products
WHERE Color='Black'
FOR XML AUTO;
```

The following XML is generated:

```xml
<Product product_ID="302"/>
<Product product_ID="400"/>
<Product product_ID="501"/>
<Product product_ID="700"/>
```

You can also rename the table with an alias. The following query renames the table to product_info:

```sql
SELECT ID AS product_ID
FROM Products AS product_info
WHERE Color='Black'
FOR XML AUTO;
```

The following XML is generated:

```xml
<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.

```sql
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:

```xml
<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:
Using XML in the database

```sql
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:

```xml
<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 682.

**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 681.)
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 682.

**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** 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>`.

- **TagName** 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!!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 for the purpose of 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 686.

- **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 685.

- **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 `&lt;a/&gt;`.

For an example using the xml directive, see “Using the xml directive” on page 687.

- **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 688.

**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 [database]” [SQL Anywhere Server - Database Administration] and “FOR XML and NULL values” on page 673.

**Writing an EXPLICIT mode query**

Suppose you want to write a query using FOR XML EXPLICIT that generates the following XML document:

```xml
<employee EmployeeID='129'>
  <customer CustomerID='107' Region='Eastern'/>
  <customer CustomerID='119' Region='Western'/>
  <customer CustomerID='131' Region='Central'/>
</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.

<table>
<thead>
<tr>
<th>Tag</th>
<th>Parent</th>
<th>employee!1!EmployeeID</th>
<th>customer!2!CustomerID</th>
<th>customer!2!Region</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>NULL</td>
<td>129</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>129</td>
<td>107</td>
<td>Eastern</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>129</td>
<td>119</td>
<td>Western</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>129</td>
<td>131</td>
<td>Central</td>
</tr>
<tr>
<td>1</td>
<td>NULL</td>
<td>195</td>
<td>NULL</td>
<td>NULL</td>
</tr>
</tbody>
</table>
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.

   To generate the <employee> elements for the table above, your first SELECT statement is as follows:

   ```sql
   SELECT 1 AS tag, NULL AS parent, EmployeeID AS [employee!1!EmployeeID], NULL AS [customer!2!CustomerID], NULL AS [customer!2!Region]
   FROM Employees;
   ```

   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:

   ```sql
   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 680.

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,
Using XML in the database

You get the following result from this query:

```
<employee>
  <id>102</id>
  <name>Fran</name>
  <department>
    <name>R &amp; D</name>
  </department>
</employee>
<employee>
  <id>105</id>
  <name>Matthew</name>
  <department>
    <name>R &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],
```

---

686 Copyright © 2008, iAnywhere Solutions, Inc. - SQL Anywhere 11.0.0
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 (for information, see “Encoding illegal XML names” on page 674) unless the column is of type XML. 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 &amp; 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>
...

Obtaining query results as XML
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 674.

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 XMLEMENT( NAME Departments,
    XMLATTRIBUTES ( DepartmentID ),
    XMLAGG( XMLEMENT( 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>
```
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:

```sql
SELECT XMLCONCAT( XMLELEMENT( NAME given_name, GivenName ),
                 XMLELEMENT( NAME surname, Surname ) ) AS "Employee_Name"
FROM Employees;
```

This query returns the following result:

<table>
<thead>
<tr>
<th>Employee_Name</th>
</tr>
</thead>
</table>
| <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:

```sql
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:

<table>
<thead>
<tr>
<th>ID</th>
<th>results</th>
</tr>
</thead>
<tbody>
<tr>
<td>301</td>
<td><code>&lt;product_info&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;item_name&gt;Tee Shirt&lt;/item_name&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;quantity_left&gt;54&lt;/quantity_left&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;description&gt;Medium Orange Tee Shirt&lt;/description&gt;</code></td>
</tr>
<tr>
<td>302</td>
<td><code>&lt;product_info&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;item_name&gt;Tee Shirt&lt;/item_name&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;quantity_left&gt;75&lt;/quantity_left&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;description&gt;One Size fits all Black Tee Shirt&lt;/description&gt;</code></td>
</tr>
<tr>
<td>400</td>
<td><code>&lt;product_info&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;item_name&gt;Baseball Cap&lt;/item_name&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;quantity_left&gt;112&lt;/quantity_left&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;description&gt;One Size fits all Black Baseball Cap&lt;/description&gt;</code></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

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`.

```sql
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, XM.getTextElement( 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, XM.getTextElement( NAME item_description,
XMLATTRIBUTES ( UnitPrice AS price ),
Products.name
) AS products
FROM Products
WHERE ID > 400;
```

For more information, see “XM.getTextElement function [String]” [SQL Anywhere Server - SQL Reference].

Example

The following example uses XM.getTextElement 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 ||
            XM.getTextElement( name "tr",
                XMLATTRIBUTES( 'left' AS "align", 'top' AS "valign" ),
                XM.getTextElement( name "td", var),
                XM.getTextElement( name "td", varval ) )
        END LOOP;

    SET res = XM.getTextElement( NAME "table",
        XMLATTRIBUTES(" " AS "BORDER", '10' as "CELLPADDING", '0' AS "CELLSPACING" ),
        XM.getTextElement( name "th",
            XMLATTRIBUTES(" left" AS "align", 'top' AS "valign" ),
            'Header Name' )
    }

```
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:

<table>
<thead>
<tr>
<th>ID</th>
<th>product_info</th>
</tr>
</thead>
<tbody>
<tr>
<td>401</td>
<td>&lt;item_description&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;name&gt;Baseball Cap&lt;/name&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;color&gt;White&lt;/color&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;price&gt;10.00&lt;/price&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;/item_description&gt;</td>
</tr>
<tr>
<td>500</td>
<td>&lt;item_description&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;name&gt;Visor&lt;/name&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;color&gt;White&lt;/color&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;price&gt;7.00&lt;/price&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;/item_description&gt;</td>
</tr>
<tr>
<td>501</td>
<td>&lt;item_description&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;name&gt;Visor&lt;/name&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;color&gt;Black&lt;/color&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;price&gt;7.00&lt;/price&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;/item_description&gt;</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>order_info</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;order&gt;</td>
</tr>
<tr>
<td>&lt;ID&gt;2001&lt;/ID&gt;</td>
</tr>
<tr>
<td>&lt;date&gt;2000-03-16&lt;/date&gt;</td>
</tr>
<tr>
<td>&lt;customer&gt;The Power Group&lt;/customer&gt;</td>
</tr>
<tr>
<td>&lt;/order&gt;</td>
</tr>
<tr>
<td>&lt;order&gt;</td>
</tr>
<tr>
<td>&lt;ID&gt;2005&lt;/ID&gt;</td>
</tr>
<tr>
<td>&lt;date&gt;2001-03-26&lt;/date&gt;</td>
</tr>
<tr>
<td>&lt;customer&gt;The Power Group&lt;/customer&gt;</td>
</tr>
<tr>
<td>&lt;/order&gt;</td>
</tr>
<tr>
<td>&lt;order&gt;</td>
</tr>
<tr>
<td>&lt;ID&gt;2125&lt;/ID&gt;</td>
</tr>
<tr>
<td>&lt;date&gt;2001-06-24&lt;/date&gt;</td>
</tr>
<tr>
<td>&lt;customer&gt;The Power Group&lt;/customer&gt;</td>
</tr>
<tr>
<td>&lt;/order&gt;</td>
</tr>
<tr>
<td>&lt;order&gt;</td>
</tr>
<tr>
<td>&lt;ID&gt;2206&lt;/ID&gt;</td>
</tr>
<tr>
<td>&lt;date&gt;2000-04-16&lt;/date&gt;</td>
</tr>
<tr>
<td>&lt;customer&gt;The Power Group&lt;/customer&gt;</td>
</tr>
<tr>
<td>&lt;/order&gt;</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>order_info</th>
</tr>
</thead>
</table>
| <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 690.

### 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.

```sql
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" ?>
<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].
Part VII. Remote Data and Bulk Operations

This part describes how to load and unload your database, and how to access remote data.
CHAPTER 17

Importing and exporting data

Contents

Transferring data into and out of databases .............................................................. 702
Importing data ............................................................................................................ 704
Exporting data ........................................................................................................... 722
Accessing data on client computers .......................................................................... 733
Rebuilding databases ................................................................................................. 736
Extracting databases ................................................................................................. 745
Migrating databases to SQL Anywhere ..................................................................... 746
Using SQL command files ......................................................................................... 750
Adaptive Server Enterprise compatibility ................................................................. 752
Transferring data into and out of databases

Transferring data into and out of your database may be necessary in several situations. For example:

- 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

These tasks are called bulk operations. They are carried out by a user with DBA authority and are not part of typical end-user applications. Bulk operations may put particular demands on concurrency and transaction logs and are often best performed when no other users are connected to the database.

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 is flexible, allowing you to write out the result set of a SELECT statement to any one of a number of 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.
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:

<table>
<thead>
<tr>
<th>Function</th>
<th>Implication</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maintain a transaction log</td>
<td>There is no record of the changes. Each COMMIT causes a checkpoint.</td>
</tr>
<tr>
<td>Lock any records</td>
<td>There are no serious implications.</td>
</tr>
</tbody>
</table>

Alternatively, you may also wish 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 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. SQL Anywhere lets you:
- import entire tables or portions of tables from text files
- import data from a variable
- import a number of 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 736.

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 705
- “-b server option” [SQL Anywhere Server - Database Administration]

Import considerations

Before you begin importing data into your database, consider in detail what resources you have and exactly what you want to accomplish.

<table>
<thead>
<tr>
<th>Consideration...</th>
<th>For information, see...</th>
</tr>
</thead>
<tbody>
<tr>
<td>The import tools that are available.</td>
<td>“Import tools” on page 705</td>
</tr>
<tr>
<td>Impact on database performance that importing might have.</td>
<td>“Performance aspects of bulk operations” on page 702</td>
</tr>
<tr>
<td>Compatibility issues between your data and the destination table.</td>
<td>“Table structures for import” on page 720</td>
</tr>
<tr>
<td>Consideration...</td>
<td>For information, see...</td>
</tr>
<tr>
<td>------------------</td>
<td>------------------------</td>
</tr>
<tr>
<td>Whether the data is on a client computer.</td>
<td>“Accessing data on client computers” on page 733</td>
</tr>
</tbody>
</table>

**Performance tips for importing data**

Although importing large volumes of data can be time consuming, there are a few things you can do to save time:

- 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. As well as improving performance for loading large amounts of data, it also serves to keep 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.
- If you are loading using the LOAD TABLE statement, do not start the database server with the -b option (bulk operations mode).
- If you are using the INPUT or OUTPUT statement, run Interactive SQL or the client application on the same computer as the database server. 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 server option” [SQL Anywhere Server - Database Administration]

**Import tools**

There are a variety of tools available to help you import data into a SQL Anywhere database. A discussion of each import tool follows.

**Using the Interactive SQL Import Wizard**

The Interactive SQL Import Wizard lets you choose a source and format, and a destination table, for the data. You can import data from TEXT and FIXED format files. You can choose to import this data into an existing table, or you can use the wizard to create and configure a completely 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 version 11.0.0 database and a SQL Anywhere version 10.0.0 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

Examples

This example shows you how to import data using the Interactive SQL Import Wizard.

To import data (Interactive SQL Data menu)
1. In Interactive SQL, from the Data menu choose Import.
2. Follow the instructions in the wizard.

This example adds data about an employee to the Employees table of the SQL Anywhere sample database.

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, from the Data menu choose 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(,) and 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:

```sql
-- Generated by the Import Wizard
INPUT INTO "GROUPO"."Employees" from 'C:\\Tobedeleted\\import.txt'
FORMAT TEXT ESCAPES ON ESCAPE CHARACTER '\' DELIMITED BY ',' ENCODING 'Cp1252'
```

The following example shows you how to add a table from an UltraLite database to a SQL Anywhere database.
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 11\Samples\UltraLite\CustDB\custdb.udb
2. In Interactive SQL, from the Data menu choose Import.
3. Select In A Database and click Next.
4. From the Database Type list choose SQL Anywhere.
   On the Identification tab, select ODBC Data Source Name, type SQL Anywhere 11 Demo, and click Next.
5. Select Customers and click Next.
6. Select In A New Table, type dba and SQLAnyCustomers, for the Owner and Table Name respectively, and then click Import.
7. Click Close.
8. View the generated SQL statement; from the SQL menu, choose 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 11 Demo;CON='''
FROM "GROUPO.Customers" INTO "dba"."SQLAnyCustomers"
CREATE TABLE ON
```

Using the INPUT statement to import data

The INPUT statement lets you import data in a variety of file formats a table. You can choose to import this data into an existing table, or you can create and configure a completely new table. You can use the USING clause to import data from different types of databases as well as from different versions of SQL Anywhere databases (as long as you have the ODBC drivers for the databases).

The INPUT statement lets you import data from TEXT and FIXED formats. If you want to import data from another file format, you can use the USING clause with an ODBC data source to import the data.

You can either use the default input format, or you can specify the file format on 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
● import data 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 becomes 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 is more flexible in terms of the databases and file formats that it supports.

Example

This example shows you how to import data using the INPUT statement.

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 11 Demo database.

3. Enter an INPUT statement in the SQL Statements pane.

   ```sql
   INPUT INTO Employees
   FROM c:\new_employees.txt
   FORMAT TEXT;
   SELECT * FROM Employees;
   ```
In this statement, Employees is the name of the destination table in the SQL Anywhere 11 Demo database, 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 took to import the data. If the import is unsuccessful, a message appears indicating why the import was unsuccessful.

This example shows you how to import data from an Excel spreadsheet into a SQL Anywhere database

To import data from a Microsoft Excel spreadsheet into a SQL Anywhere database

1. Open the spreadsheet in Microsoft Excel.
2. In Excel, select the cells you want to import, then choose Insert » Name » Define.

   The Define Name window appears.

   Type a name, such as myData, for the selected cells.

   Click OK.
3. Save and close the spreadsheet.
4. Create an ODBC data source for the spreadsheet.
   ● Start the ODBC Administrator by choosing Start » Programs » SQL Anywhere 11 » ODBC Administrator.
   ● Select the User DSN tab to create a DSN for the current user or select the System DSN tab to create a system-wide DSN.
   ● Click Add.

   The Create New Data Source Wizard appears.

   From the list of drivers, choose the Microsoft Excel Driver, and then click Finish.

   The ODBC Microsoft Excel Setup window appears.
   ● Specify the parameters you need, click OK to close the window and create the data source.

   For example, type myExcelFile in the Data Source Name field. Click Select Workbook and browse to find your Excel spreadsheet file.
   ● Click OK to save the DSN.
5. Open Interactive SQL and connect to a SQL Anywhere database.
6. Execute the following INPUT statement to import the data from the Excel spreadsheet and to save it in a new table called t:

   ```sql
   INPUT USING 'dsn=myExcelFile;DSN=myExcelFile'
   FROM "myData" INTO "t"
   CREATE TABLE ON
   ```

Using the LOAD TABLE statement to import data

The LOAD TABLE statement lets you efficiently import data into an existing table in text/ASCII format.
You can use the LOAD TABLE statement to import data from a file. The file can reside on the database server computer or on the client computer. 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 733
- “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]

Using the INSERT statement to import data

The INSERT statement lets you add rows to the database. Because you include the import data for your destination table directly 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.

Example

This example shows you how to import data using the INSERT statement. With this INSERT statement, you add data to the Departments table of the SQL Anywhere sample database.

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.

   ```sql
   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]

Using the MERGE statement

Another method for updating large amounts of data in a table is to perform an update operation using the MERGE statement. 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 purposes of explanation. For the full syntax of the MERGE statement, see “MERGE statement” [SQL Anywhere Server - SQL Reference].

MERGE INTO target-object
USING source-object
ON merge-search-condition
{ WHEN MATCHED | WHEN NOT MATCHED } [...]
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.

```sql
CREATE TABLE myProducts (product_id NUMERIC(10),
```
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);
SEEELECT * FROM myProducts;

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;

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;
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;
```

```
MERGE INTO myTargetTable
USING mySourceTable ON PRIMARY KEY
```

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 two fictitious tables, mySourceTable and myTargetTable, using the primary key values of myTargetTable to match rows. That is, if a row in mySourceTable has the same value as the primary key column of myTargetTable, the row is considered a match.

```
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, .. In) and that the primary key is defined on columns (I1, I2). The mySourceTable has the columns (U1, U2, .. Un).

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:
Using proxy tables to import data

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 753.

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 dates that are not valid dates and numbers that are not valid numbers. The conversion_error database option allows you to ignore conversion errors by converting 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 [compatibility]” [SQL Anywhere Server - Database Administration].
Importing tables

You can import tables into an existing database using either the Interactive SQL Data menu or the SQL Statements pane.

To import a table (Interactive SQL Data menu)

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:

   ```sql
   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,

   ```sql
   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:

```sql
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 instead.

Merging different table structures

You can rearrange the import data to fit the table using a variation of the INSERT statement and a global temporary 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 516.
Exporting data

Exporting data is an administrative task that involves writing data out of your database. Exporting is a useful tool if you need to share large portions of your database, or extract portions of your database according to particular criteria. SQL Anywhere lets you:

- export individual tables, query results, or table schema
- create scripts that automate exporting so that you can export a number of 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

For performance reasons, if you want to export an entire database, unload the database instead of exporting the data. See “Rebuilding databases” on page 736.

See also

- “UNLOAD statement” [SQL Anywhere Server - SQL Reference]
- “OUTPUT statement [Interactive SQL]” [SQL Anywhere Server - SQL Reference]

Export considerations

Before you begin exporting data, spend some time and consider in detail what resources you have and exactly what type of information you want to export from your database.

<table>
<thead>
<tr>
<th>Consideration...</th>
<th>For information, see...</th>
</tr>
</thead>
<tbody>
<tr>
<td>The export tools that are available.</td>
<td>“Export tools” on page 722</td>
</tr>
<tr>
<td>Impact of exporting on the database performance.</td>
<td>“Performance aspects of bulk operations” on page 702</td>
</tr>
<tr>
<td>How to handle NULL values.</td>
<td>“Outputting NULLs using the OUTPUT statement” on page 730</td>
</tr>
<tr>
<td>Whether the data file is on a client computer.</td>
<td>“Accessing data on client computers” on page 733</td>
</tr>
</tbody>
</table>

Export tools

There are a variety of tools available to help you export your data.
Using the Interactive SQL Export Wizard

You can use the Interactive SQL Export Wizard to export query results.

Use this export tool when you want to save result sets to a file or database. In Interactive SQL, from the Data menu, choose Export. This allows you to choose the file format of the exported query results.

Examples

The following example shows you how to export a result set from a SQL query to a file.

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 wizard.

The following example shows you how to export a result set from a SQL query to a database.

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.

   ```sql
   SELECT * FROM Employees
   WHERE State = 'GA';
   ```

   The result set includes a list of all employees who live in Georgia.

2. Choose Data » Export.

   The Export Wizard appears.

3. Select In A Database.

4. From the Database Type list, select UltraLite.

   On the Identification tab, type dba and sql in the User Id and Password fields, respectively. In the Database File, type C:\Documents and Settings\All Users\Documents\SQL Anywhere 11\Samples \UltraLite\CustDB\custdb.udb.

5. Click Next.

6. Select In A New Table, type dba and NewTable, for the Owner and Table Name respectively, and then click Export.

   The result set is exported to the table that you specified.

7. From the SQL menu, choose Previous SQL.

   The IMPORT statement created and used by the Import Wizard appears in the SQL Statements pane:

   ```sql
   --  Generated by the Export Wizard
   OUTPUT USING 'driver=UltraLite 11;UID=dba;PWD=sql;
   DBF=C:\Documents and Settings\All Users\Documents\SQL Anywhere 11\Samples \UltraLite\CustDB\custdb.udb'
   INTO "dba"."NewTable"
   CREATE TABLE ON
   ```
Using the OUTPUT statement to export data

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 either 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.

```sql
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

```sql
SELECT *
FROM Employees;
OUTPUT USING 'driver=SQL Anywhere 11;UID=dba;PWD=sql;DBF=C:\Tobedeleted \mydatabase.db;CON="#"
INTO "dba"."newcustomers"
CREATE TABLE ON
```

Using 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. The data is exported in order by primary key values to make reloading faster.
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 733
- “UNLOAD statement” [SQL Anywhere Server - SQL Reference]
- “OUTPUT statement [Interactive SQL]” [SQL Anywhere Server - SQL Reference]

**Using the UNLOAD statement**

The UNLOAD statement is similar to the OUTPUT statement in that they both export query results to a file. The UNLOAD statement, however, allows you to export data in a more efficient manner and in text formats only. 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 the permissions required to execute the SELECT that is specified as part of the statement.

For more information about controlling who can use the UNLOAD statement, see “-gl 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:

```sql
UNLOAD
SELECT * FROM Employees
WHERE State = 'GA'
TO 'employee_data.csv';
```

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 733
- “UNLOAD statement” [SQL Anywhere Server - SQL Reference]
- “OUTPUT statement [Interactive SQL]” [SQL Anywhere Server - SQL Reference]

Using the dbunload utility to export data

The dbunload utility lets you export one, many, or all of the tables in the database. You can export table data, as well as table schemas. If you want to rearrange your tables in the database, 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].

Using the Unload Database Wizard

You can use the Unload Database Wizard to unload an existing database into a new database.
When using this 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 this 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 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. From the Tools menu, choose SQL Anywhere 11 » Unload Database.
2. Follow the instructions in the Unload Database Wizard.

**Tip**
You can use any of the following methods to access the Unload Database Wizard from within Sybase Central:
- Select the database, and choose File » Unload Database.
- Right-click the database, and choose Unload Database.
Using 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. Connect to the database as a user with DBA authority.
2. Double-click Tables.
3. Select the table you want to export data from, or to select multiple tables, press Ctrl and select the tables.
4. From the File menu, choose Unload Data.
5. Complete the Unload Data window and click OK.

Exporting query results

You can export queries (including queries on views) to a file from Interactive SQL with the Data menu, the OUTPUT statement, or the UNLOAD statement.

You can import and export files between SQL Anywhere and Adaptive Server Enterprise using the BCP FORMAT clause.

For more information, see “Adaptive Server Enterprise compatibility” on page 752.

To export query results (Interactive SQL Data menu)
1. Enter your query in the SQL Statements pane of Interactive SQL.
2. Choose SQL » Execute to display the result set.
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. Do any of the following:

<table>
<thead>
<tr>
<th>To do this...</th>
<th>Use this clause...</th>
<th>Example...</th>
</tr>
</thead>
</table>
| Export query results and append the results to another file | APPEND | SELECT * FROM Employees;
OUTPUT TO 'employees.txt'
APPEND; |
| Export query results and include messages | VERBOSE | 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 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 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.
Outputting NULLs using the OUTPUT statement

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):

  ```sql
  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.

Exporting databases

Note
If the database you want to unload is already running, and you start the Unload Database Wizard, the SQL Anywhere plug-in automatically stops the database before you can unload it.

To unload all or part of a database (Sybase Central)

1. From the Tools menu, choose SQL Anywhere 11 » Unload Database.
2. Follow the instructions in the wizard.

To unload all or part of a database (Command line)

- Run the dbunload utility, specifying connection parameters using the -c option.

  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].

Exporting tables

In addition to the methods described in the sections that follow, you can also export a table by selecting all the data in a table and exporting the query results. See “Exporting query results” on page 728.

You can export views just as you would export tables.

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

● “Exporting query results” on page 728
● “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 benefiting 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 server option” [SQL Anywhere Server - Database Administration].

## Security of client side data

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 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 [database]” [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

In the event that 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 11 - 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 “Introduction to backup and 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 745.

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 and Replication Agent require this information. If the database is not involved in replication, the process is simpler.

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. The New Features documentation states whether an unload and reload is required to obtain a particular feature.

  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 11 - 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 11 - 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.

<table>
<thead>
<tr>
<th>Consideration...</th>
<th>For information, see...</th>
</tr>
</thead>
</table>
| What is involved in rebuilding a database | “Rebuilding databases” on page 736  
“Reasons to rebuild databases” on page 737 |
| The available tools for rebuilding | “Rebuild and reload tools” on page 741 |
| The impact that rebuilding has on the database users | “Minimizing downtime during rebuilding” on page 743 |
| Whether or not the database is involved in synchronization or replication | “Rebuilding databases involved in synchronization or replication” on page 740  
“Rebuilding databases not involved in synchronization or replication” on page 739 |

**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 considerations**

The rebuild (unload and reload) and extract procedures rebuild databases and create new databases from part of an old one. Before rebuilding a database, you should consider how to go about the process.
Rebuilding 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:

<table>
<thead>
<tr>
<th>To do this...</th>
<th>Use this option...</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rebuild to a new database</td>
<td>-an</td>
<td>dbunload -c &quot;DBF=demo.db;UID=DBA;PWD=sql&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-an DemoBackup.db</td>
</tr>
<tr>
<td>Reload to an existing database</td>
<td>-ac</td>
<td>dbunload -c &quot;DBF=demo.db;UID=DBA;PWD=sql&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-ac &quot;UID=DBA;PWD=sql;DBF=NewDemo.db&quot;</td>
</tr>
<tr>
<td>Replace an existing database</td>
<td>-ar</td>
<td>dbunload -c &quot;DBF=demo.db;UID=DBA;PWD=sql&quot;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-ar</td>
</tr>
</tbody>
</table>

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, as well as connection parameter options that allow you to specify a running or non-running database and database parameters.

**Rebuilding databases involved in synchronization or replication**

This section applies to SQL Anywhere MobiLink clients (clients using dbmlsync), as well as SQL Remote and Replication Agent.

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 accomplishing the same task.

**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 “Validating 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.

**Notes**

There are additional options available for the dbunload utility that allow you to tune the unload, as well as 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].
If the above procedure does not meet your needs, you can manually adjust the transaction log offsets. The following procedure describes how to perform that operation.

**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 736.
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 -il -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.

**Rebuild and reload tools**

There are a variety of tools to help you rebuild a database. There are also some tools to help you reload data into an existing 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:
Importing and exporting data

- 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:
- “Rebuilding databases not involved in synchronization or replication” on page 739
- “Rebuilding databases involved in synchronization or replication” on page 740

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].

Exporting 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.

### Reloading databases

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 `newdemo.db`.
   
   ```
   dbinit newdemo.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=newdemo.db;UID=DBA;PWD=sql" reload.sql
   ```

### Minimizing downtime during rebuilding

The following steps help you rebuild a database while minimizing downtime. This can be especially useful if your database is in operation 24 hours a day.

It is wise to do a practice run of steps 1-4, and determine the times required for each step, prior to beginning the actual rebuild. You may also want to save copies of your files at various points during the rebuild.

**Caution**

Make sure that no other scheduled backups rename the production database's log. If this happens in error, you will need to apply the transactions from these renamed logs 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].
You now have a rebuilt database that contains all transactions up to the end of the backup in Step 3.

5. Shut down the production server and make copies of the database and log.

6. Copy the rebuilt database onto the production server.

7. Run dbtran on the log from Step 5.

   This should be a relatively small file.

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:

- “Database Extraction utility” [SQL Remote]
- “Using the Extraction utility (dbxtract)” [SQL Remote]
- “Extracting groups” [SQL Remote]
- “Deploying remote databases” [MobiLink - Client Administration]
- “Extracting a remote database in Sybase Central” [SQL Remote]
Migrating databases to SQL Anywhere

Using the sa_migrate set of system procedures or the Migrate Database Wizard, you can import tables from remote Oracle, DB2, Microsoft SQL Server, Sybase Adaptive Server Enterprise, and SQL Anywhere databases into SQL Anywhere.

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 database” [SQL Anywhere Server - Database Administration].

Using the Migrate Database Wizard

You can create a remote server to connect to the remote database, as well as 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. Connect to the database as a user with DBA authority.
2. Choose Tools » SQL Anywhere 11 » 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 “Creating remote servers” on page 757.
   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.

**Using the sa_migrate system procedures**

The sa_migrate system procedures can allow you to migrate remote data. If you do not want to modify the tables in any way, you can use the two-step method. Alternatively, if you would like to remove tables or foreign key mappings, you can use the extended method.

When using the sa_migrate set of system procedures, you must complete the following steps before you can import a remote database:

1. Create a target database.
   
   For more information, see “Creating a database” [SQL Anywhere Server - Database Administration].

2. Create a remote server to connect to the remote database.
   
   For more information, see “Creating remote servers” on page 757.

3. 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.
   
   For more information, see “Creating external logins” on page 765.

4. Create a local user who will own the migrated tables in the target database.
   
   For more information, see “Creating new users” [SQL Anywhere Server - Database Administration].

**Migrating a database in two steps using sa_migrate**

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.

**To import remote tables (two-step method)**

1. From Interactive SQL, connect to the target database.

2. In the SQL Statements pane, run the sa_migrate system procedure. For example,

   ```sql
   CALL sa_migrate( 'local_a', 'ase', NULL, l_smith, NULL, 1, 1, 1 );
   ```

   This procedure calls several procedures in turn and migrates all the remote tables belonging to the user l_smith 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. From Interactive SQL, connect to the target database.
2. Run the sa_migrate_create_remote_table_list system procedure. For example,
   
   ```
   CALL sa_migrate_create_remote_table_list( 'ase', NULL, 'remote_a', '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].
   
3. Run the sa_migrate_create_tables system procedure. For example:
   
   ```
   CALL sa_migrate_create_tables( 'local_a' );
   
   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].
   
4. 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_a' );
   
   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 7.
   
5. Run the sa_migrate_create_remote_fks_list system procedure. For example,
   
   ```
   CALL sa_migrate_create_remote_fks_list( 'ase' );
   
   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].

6. Run the sa_migrate_create_fks system procedure. For example,

   CALL sa_migrate_create_fks('local_a');

   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].

7. 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_a');

   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.

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. Command files are also commonly called scripts. See “Comments” [SQL Anywhere Server - SQL Reference].

Opening SQL command files in Interactive SQL

On Microsoft Windows platforms you can make Interactive SQL the default editor for .sql files. Thus, you can double-click the file so that its contents appears in the SQL Statements pane of Interactive SQL.

For more information about making Interactive SQL the default editor for .sql files, see “Setting Interactive SQL as the default editor for .sql files” [SQL Anywhere Server - Database Administration].

Running 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:

    ```sql
    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 prompt)**

- 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 11 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 load a 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`:

```sql
SELECT * FROM Employees;
OUTPUT TO 'C:\My Documents\Employees.txt';
```

For more information, see “OUTPUT statement [Interactive SQL]” [SQL Anywhere Server - SQL Reference].
Adaptive Server Enterprise compatibility

You can import and export files between SQL Anywhere and Adaptive Server Enterprise using the BCP FORMAT clause. Simply ensure that the BCP output is in delimited text format. 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.

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].
## CHAPTER 18

**Accessing remote data**

### Contents

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction to remote data access</td>
<td>754</td>
</tr>
<tr>
<td>Working with remote servers</td>
<td>757</td>
</tr>
<tr>
<td>Using directory access servers</td>
<td>762</td>
</tr>
<tr>
<td>Working with external logins</td>
<td>765</td>
</tr>
<tr>
<td>Working with proxy tables</td>
<td>767</td>
</tr>
<tr>
<td>Joining remote tables</td>
<td>771</td>
</tr>
<tr>
<td>Joining tables from multiple local databases</td>
<td>773</td>
</tr>
<tr>
<td>Sending native statements to remote servers</td>
<td>774</td>
</tr>
<tr>
<td>Using remote procedure calls (RPCs)</td>
<td>775</td>
</tr>
<tr>
<td>Transaction management and remote data</td>
<td>777</td>
</tr>
<tr>
<td>Internal operations</td>
<td>778</td>
</tr>
<tr>
<td>Troubleshooting remote data access</td>
<td>781</td>
</tr>
</tbody>
</table>
Introduction to remote data access

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 DB2.
- Access desktop data such as Excel spreadsheets, MS-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
- Oracle
- IBM DB2
- Microsoft SQL Server
- Other ODBC data sources

For platform availability, see SQL Anywhere Supported Platforms and Engineering Status.

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 757.

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 765.

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 767.

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 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, DB/2, 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.

There are two groups of server classes. The first is ODBC-based, and the second is JDBC-based.

The ODBC-based server classes are:

- **saodbc** for SQL Anywhere.
- **ulodbc** for UltraLite.
- **aseodbc** for Sybase SQL Server and Adaptive Server Enterprise (version 10 and later).
- **adsodbc** for Advantage Database Server.
- **db2odbc** for IBM DB2.
- **mssodbc** for Microsoft SQL Server.
- **oraodbc** for Oracle servers (version 8.0 and later).
- **mysqlodbc** for MySQL.
Accessing remote data

- msaccessodbc for Microsoft Access.
- odbc for all other ODBC data sources.

**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.

The JDBC-based server classes are:
- sajdbc for SQL Anywhere.
- asejdbc for Sybase SQL Server and Adaptive Server Enterprise (version 10 and later).

**Note**
The JDBC classes have a significant performance impact and should only be used in situations where the ODBC classes cannot be used.

For a full description of remote server classes, see “Server classes for remote data access” on page 783.

**Accessing remote data from PowerBuilder DataWindows**

You can access remote data from a PowerBuilder DataWindow by setting the DBParm Block parameter to 1 on connect.

- 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.

Creating remote servers

Use the CREATE SERVER statement to set up remote server definitions. You can execute the statements directly, or use Sybase Central.

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 11;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.

Creating remote servers using Sybase Central

You can create a remote server using a wizard in Sybase Central.

**To create a remote server (Sybase Central)**

1. Connect to the host database as a user with DBA authority.
2. In the left pane, double-click **Remote Servers**.
3. From the **File** menu, choose **New » Remote Server**.
   
   The **Create Remote Server Wizard** appears.
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.

Deleting 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. 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 choose 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;
```
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.

```sql
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].

**Listing 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.

```sql
sp_remote_tables(
@server-name
[, @table-name
[, @table-owner
[, @table-qualifier
```

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 of the Microsoft Excel worksheets available from a remote server named excel:

```sql
CALL sp_remote_tables excel;
```

Or to get a list of all of the tables in the production database in an Adaptive Server Enterprise server named asetest, owned by fred:

```sql
CALL sp_remote_tables asetest, null, fred, production;
```

For more information, see “`sp_remote_tables` system procedure” [SQL Anywhere Server - SQL Reference].
Listing 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.

Creating 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. Connect to the database as a user with DBA authority.
2. In the left pane, double-click Directory Access Servers.

The Create Directory Access Server Wizard appears.

4. Follow the instructions in the 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:
   
   ```sql
   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]

Deleting 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.
Dropping directory access servers

To drop a directory access server (Sybase Central)

1. Connect to the 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.

To 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;
   ```

Dropping proxy tables

Use the DROP TABLE statement to drop a proxy table used by the directory access server.

To drop a proxy table (Sybase Central)

1. Connect to the 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.

To 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 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 integrated logins” [SQL Anywhere Server - Database Administration].

Creating external logins

Use one of the following procedures to create an external login.

To create an external login (Sybase Central)

1. 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 or as the owner of the external login.
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].

Dropping 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. 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 or as the owner of the external login.

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 123.

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.

Specifying 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. In some cases 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. In the case of 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'`

### Creating 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. 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**.

### Creating 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';
```

Creating 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';
```

Listing 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].
Joining 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:

   ```
   dbeng11 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 DSN SQL Anywhere 11 Demo:

   ```
   CREATE SERVER RemoteSA
   CLASS 'saodbc'
   USING 'SQL Anywhere 11 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.

   In some cases 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;
Joining 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 799.

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.

```sql
CREATE SERVER remote_db2
CLASS 'saodbc'
USING 'RemoteSA_db2';
CREATE SERVER remote_db3
CLASS 'saodbc'
USING 'RemoteSA_db3';
```

Alternatively, using JDBC:

```sql
CREATE SERVER remote_db2
CLASS 'sajdbc'
USING 'mypc1:2638/db2';
CREATE SERVER remote_db3
CLASS 'sajdbc'
USING 'mypc1:2638/db3';
```

4. Create proxy table definitions by executing CREATE EXISTING TABLE statements for the tables in the other databases you want to access.

```sql
CREATE EXISTING TABLE Employees
AT 'remote_db2...Employees';
```
Sending 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 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.

Creating 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 return one in all cases, then the local procedure definition must contain a RESULT clause.

To create a remote procedure (Sybase Central)

1. 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.
5. Follow the instructions in the Create Remote Procedure Wizard.

To create a remote procedure (SQL)

1. Connect to the database as a user with DBA authority.
2. Define the procedure to SQL Anywhere.

The syntax is the same as a local procedure definition, except instead of using SQL statements to make up the body of the procedure, a location string is given defining the location where the procedure resides.

```sql
CREATE PROCEDURE remotewho()
AT 'bostonase.master.dbo.sp_who';
```

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:

```sql
CALL remotewho();
```
Example

This example specifies a parameter when calling a remote procedure:

```sql
CREATE PROCEDURE remoteuser ( IN uname CHAR( 30 ) )
AT 'bostonase.master.dbo.sp_helpuser';
CALL remoteuser( 'joe' );
```

Data types for remote procedures

The following data types are allowed for RPC 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

Dropping remote procedures

Use one of the following procedures to delete a remote procedure.

**To delete a remote procedure (Sybase Central)**

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

For more information, see “DROP statement” [SQL Anywhere Server - SQL Reference].

Example

Delete a remote procedure called remoteproc.

```sql
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 119.

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:

```sql
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 datetime, 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 of 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 is possible to the remote server. In many cases this will be the complete statement as 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 is capable of processing 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 cannot 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 \texttt{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 \texttt{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 \texttt{WHERE} clause as possible, followed by a positioned UPDATE or DELETE statement that specifies \texttt{WHERE CURRENT OF \textit{cursor-name}}.

For example, when the function \texttt{ATAN2} is not supported by a remote server:

```
UPDATE t1
SET a = \texttt{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 \texttt{a} and issue:

```
UPDATE t1
SET a = 'new value'
WHERE CURRENT OF CURSOR;
```

If \texttt{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 (\texttt{WHERE CURRENT OF \textit{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 \texttt{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.
- 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.
- Calls to functions that contain an expression that references a proxy table.

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 [database]” [SQL Anywhere Server - Database Administration].

For more information about the -o server option, see “-o 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, as well as 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 [database]” [SQL Anywhere Server - Database Administration].

For more information about the -o server option, see “-o 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 138.

## 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 in order 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].
CHAPTER 19

Server classes for remote data access

Contents

Introduction to server classes for remote data access .............................................. 784
ODBC-based server classes ..................................................................................... 785
JDBC-based server classes ...................................................................................... 799
Introduction to 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

Each server class has a set of unique characteristics that database administrators and programmers 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
- aseodbc
- db2odbc
- mssodbc
- oraadbc
- msaccessodbc
- mysqlodbc
- ulodbc
- adsodbc
- odbc

**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 an ODBC data source” [*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 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 11;ENG=TestSA;DBN=sample;LINKS=tcpip()';
```

This defines a connection to a SQL Anywhere database server named TestSA 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 786
- “Server class ulodbc” on page 786
- “Server class aseodbc” on page 786
- “Server class db2odbc” on page 789
- “Server class oraodbc” on page 791
- “Server class mssodbc” on page 793
- “Server class msaccessodbc” on page 795
- “Server class mysqlodbc” on page 794
- “Server class adsodbc” on page 789
- “Server class odbc” on page 796

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. See “USING parameter in the CREATE SERVER statement” on page 799.

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].

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_identifiers option controls the use of quoted identifiers for Adaptive Server Enterprise. For example, if you set the quoted_identifiers 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.

<table>
<thead>
<tr>
<th>SQL Anywhere data type</th>
<th>Adaptive Server Enterprise default data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIT</td>
<td>bit</td>
</tr>
<tr>
<td>TINYINT</td>
<td>tinyint</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>smallint</td>
</tr>
<tr>
<td>INT</td>
<td>int</td>
</tr>
<tr>
<td>INTEGER</td>
<td>integer</td>
</tr>
<tr>
<td>DECIMAL [defaults p=30, s=6]</td>
<td>numeric(30,6)</td>
</tr>
<tr>
<td>DECIMAL(128,128)</td>
<td>not supported</td>
</tr>
<tr>
<td>SQL Anywhere data type</td>
<td>Adaptive Server Enterprise default data type</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>---------------------------------------------</td>
</tr>
<tr>
<td>NUMERIC [defaults p=30 s=6]</td>
<td>numeric(30,6)</td>
</tr>
<tr>
<td>NUMERIC(128,128)</td>
<td>not supported</td>
</tr>
<tr>
<td>FLOAT</td>
<td>real</td>
</tr>
<tr>
<td>REAL</td>
<td>real</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>float</td>
</tr>
<tr>
<td>SMALLMONEY</td>
<td>numeric(10,4)</td>
</tr>
<tr>
<td>MONEY</td>
<td>numeric(19,4)</td>
</tr>
<tr>
<td>DATE</td>
<td>datetime</td>
</tr>
<tr>
<td>TIME</td>
<td>datetime</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>datetime</td>
</tr>
<tr>
<td>SMALLDATETIME</td>
<td>datetime</td>
</tr>
<tr>
<td>DATETIME</td>
<td>datetime</td>
</tr>
<tr>
<td>CHAR((n))</td>
<td>varchar((n))</td>
</tr>
<tr>
<td>CHARACTER((n))</td>
<td>varchar((n))</td>
</tr>
<tr>
<td>VARCHAR((n))</td>
<td>varchar((n))</td>
</tr>
<tr>
<td>CHARACTER VARYING((n))</td>
<td>varchar((n))</td>
</tr>
<tr>
<td>LONG VARCHAR</td>
<td>text</td>
</tr>
<tr>
<td>TEXT</td>
<td>text</td>
</tr>
<tr>
<td>BINARY((n))</td>
<td>binary((n))</td>
</tr>
<tr>
<td>LONG BINARY</td>
<td>image</td>
</tr>
<tr>
<td>IMAGE</td>
<td>image</td>
</tr>
<tr>
<td>BIGINT</td>
<td>numeric(20,0)</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>SQL Anywhere data type</th>
<th>ADS default data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIT</td>
<td>Logical</td>
</tr>
<tr>
<td>TINYINT, SMALLINT, INT, INTEGER</td>
<td>Integer</td>
</tr>
<tr>
<td>BIGINT</td>
<td>Numeric(32)</td>
</tr>
<tr>
<td>DECIMAL(p,s), NUMERIC(p,s)</td>
<td>Numeric(p+3)</td>
</tr>
<tr>
<td>DATE</td>
<td>Date</td>
</tr>
<tr>
<td>TIME</td>
<td>Time</td>
</tr>
<tr>
<td>DATETIME, TIMESTAMP</td>
<td>TimeStamp</td>
</tr>
<tr>
<td>MONEY, SMALLMONEY</td>
<td>Money</td>
</tr>
<tr>
<td>FLOAT, REAL</td>
<td>Double</td>
</tr>
<tr>
<td>CHAR(n), VARCHAR(n), LONG VARCHAR</td>
<td>Char(n)</td>
</tr>
<tr>
<td>BINARY(n), VARBINARY(n), LONG BINARY</td>
<td>Blob</td>
</tr>
</tbody>
</table>

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 DB2 data sources.
- The following is an example of a CREATE EXISTING TABLE statement for a DB2 server with an ODBC data source named mydb2:

```
CREATE EXISTING TABLE ibmcol
AT 'mydb2..sysibm.syscolumns';
```
**Data type conversions: DB2**

When you issue a CREATE TABLE statement, SQL Anywhere automatically converts the data types to the corresponding DB2 data types. The following table describes the SQL Anywhere to DB2 data type conversions.

<table>
<thead>
<tr>
<th>SQL Anywhere data type</th>
<th>DB2 default data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIT</td>
<td>smallint</td>
</tr>
<tr>
<td>TINYINT</td>
<td>smallint</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>smallint</td>
</tr>
<tr>
<td>INT</td>
<td>int</td>
</tr>
<tr>
<td>INTEGER</td>
<td>int</td>
</tr>
<tr>
<td>BIGINT</td>
<td>decimal(20,0)</td>
</tr>
<tr>
<td>CHAR(1-254)</td>
<td>varchar(n)</td>
</tr>
<tr>
<td>CHAR(255-4000)</td>
<td>varchar(n)</td>
</tr>
<tr>
<td>CHAR(4001-32767)</td>
<td>long varchar</td>
</tr>
<tr>
<td>CHARACTER(1-254)</td>
<td>varchar(n)</td>
</tr>
<tr>
<td>CHARACTER(255-4000)</td>
<td>varchar(n)</td>
</tr>
<tr>
<td>CHARACTER(4001-32767)</td>
<td>long varchar</td>
</tr>
<tr>
<td>VARCHAR(1-4000)</td>
<td>varchar(n)</td>
</tr>
<tr>
<td>VARCHAR(4001-32767)</td>
<td>long varchar</td>
</tr>
<tr>
<td>CHARACTER VARYING(1-4000)</td>
<td>varchar(n)</td>
</tr>
<tr>
<td>CHARACTER VARYING(4001-32767)</td>
<td>long varchar</td>
</tr>
<tr>
<td>LONG VARCHAR</td>
<td>long varchar</td>
</tr>
<tr>
<td>TEXT</td>
<td>long varchar</td>
</tr>
<tr>
<td>BINARY(1-4000)</td>
<td>varchar for bit data</td>
</tr>
<tr>
<td>BINARY(4001-32767)</td>
<td>long varchar for bit data</td>
</tr>
<tr>
<td>LONG BINARY</td>
<td>long varchar for bit data</td>
</tr>
</tbody>
</table>
SQL Anywhere data type | DB2 default data type
--- | ---
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)
DATE | date
TIME | time
SMALLDATETIME | timestamp
DATETIME | timestamp
TIMESTAMP | timestamp

Server class oraodbc

A server with server class oraodbc is Oracle version 8.0 or later.

Notes

- Sybase certifies the use of version 8.0.03 of Oracle's 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.

<table>
<thead>
<tr>
<th>SQL Anywhere data type</th>
<th>Oracle data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIT</td>
<td>number(1,0)</td>
</tr>
<tr>
<td>TINYINT</td>
<td>number(3,0)</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>number(5,0)</td>
</tr>
<tr>
<td>INT</td>
<td>number(11,0)</td>
</tr>
<tr>
<td>BIGINT</td>
<td>number(20,0)</td>
</tr>
<tr>
<td>DECIMAL(prec, scale)</td>
<td>number(prec, scale)</td>
</tr>
<tr>
<td>NUMERIC(prec, scale)</td>
<td>number(prec, scale)</td>
</tr>
<tr>
<td>FLOAT</td>
<td>float</td>
</tr>
<tr>
<td>REAL</td>
<td>real</td>
</tr>
<tr>
<td>SMALLMONEY</td>
<td>numeric(13,4)</td>
</tr>
<tr>
<td>MONEY</td>
<td>number(19,4)</td>
</tr>
<tr>
<td>DATE</td>
<td>date</td>
</tr>
<tr>
<td>TIME</td>
<td>date</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>date</td>
</tr>
<tr>
<td>SMALLDATETIME</td>
<td>date</td>
</tr>
<tr>
<td>DATETIME</td>
<td>date</td>
</tr>
<tr>
<td>CHAR(n)</td>
<td>if (n &gt; 255) long else varchar(n)</td>
</tr>
<tr>
<td>VARCHAR(n)</td>
<td>if (n &gt; 2000) long else varchar(n)</td>
</tr>
<tr>
<td>LONG VARCHAR</td>
<td>long or clob</td>
</tr>
<tr>
<td>BINARY(n)</td>
<td>if (n &gt; 255) long raw else raw(n)</td>
</tr>
<tr>
<td>VARBINARY(n)</td>
<td>if (n &gt; 255) long raw else raw(n)</td>
</tr>
<tr>
<td>LONG BINARY</td>
<td>long raw</td>
</tr>
</tbody>
</table>
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 version 3.60.0319 of Microsoft SQL Server's 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_identifiers option controls the use of quoted identifiers for Microsoft SQL Server. For example, if you set the quoted_identifiers 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.

<table>
<thead>
<tr>
<th>SQL Anywhere data type</th>
<th>Microsoft SQL Server default data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIT</td>
<td>bit</td>
</tr>
<tr>
<td>TINYINT</td>
<td>tinyint</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>smallint</td>
</tr>
<tr>
<td>INT</td>
<td>int</td>
</tr>
<tr>
<td>BIGINT</td>
<td>numeric(20,0)</td>
</tr>
<tr>
<td>DECIMAL [defaults p=30, s=6]</td>
<td>decimal(prec, scale)</td>
</tr>
<tr>
<td>NUMERIC [defaults p=30 s=6]</td>
<td>numeric(prec, scale)</td>
</tr>
<tr>
<td>FLOAT</td>
<td>if (prec) float(prec) else float</td>
</tr>
<tr>
<td>REAL</td>
<td>real</td>
</tr>
<tr>
<td>SMALLMONEY</td>
<td>smallmoney</td>
</tr>
<tr>
<td>MONEY</td>
<td>money</td>
</tr>
<tr>
<td>DATE</td>
<td>datetime</td>
</tr>
<tr>
<td>TIME</td>
<td>datetime</td>
</tr>
</tbody>
</table>
### SQL Anywhere data type

<table>
<thead>
<tr>
<th>SQL Anywhere data type</th>
<th>Microsoft SQL Server default data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIMESTAMP</td>
<td>datetime</td>
</tr>
<tr>
<td>SMALLDATETIME</td>
<td>datetime</td>
</tr>
<tr>
<td>DATETIME</td>
<td>datetime</td>
</tr>
<tr>
<td>CHAR(n)</td>
<td>if (length &gt; 255) text else varchar(length)</td>
</tr>
<tr>
<td>CHARACTER(n)</td>
<td>char(n)</td>
</tr>
<tr>
<td>VARCHAR(n)</td>
<td>if (length &gt; 255) text else varchar(length)</td>
</tr>
<tr>
<td>LONG VARCHAR</td>
<td>text</td>
</tr>
<tr>
<td>BINARY(n)</td>
<td>if (length &gt; 255) image else binary(length)</td>
</tr>
<tr>
<td>LONG BINARY</td>
<td>image</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>float</td>
</tr>
<tr>
<td>UNIQUEIDENTIFIERSTR</td>
<td>uniqueidentifier</td>
</tr>
</tbody>
</table>

### 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.

<table>
<thead>
<tr>
<th>SQL Anywhere data type</th>
<th>MySQL default data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIT</td>
<td>bit(1)</td>
</tr>
<tr>
<td>TINYINT</td>
<td>tinyint unsigned</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>smallint</td>
</tr>
<tr>
<td>INT, INTEGER</td>
<td>int</td>
</tr>
<tr>
<td>BIGINT</td>
<td>bigint</td>
</tr>
<tr>
<td>DECIMAL(p,s), NUMERIC(p,s)</td>
<td>decimal(p,s)</td>
</tr>
<tr>
<td>DATE</td>
<td>date</td>
</tr>
<tr>
<td>TIME</td>
<td>time</td>
</tr>
<tr>
<td>SQL Anywhere data type</td>
<td>MySQL default data type</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>--------------------------------</td>
</tr>
<tr>
<td>DATETIME, TIMESTAMP</td>
<td>datetime</td>
</tr>
<tr>
<td>MONEY</td>
<td>decimal(19,4)</td>
</tr>
<tr>
<td>SMALLMONEY</td>
<td>decimal(10,4)</td>
</tr>
<tr>
<td>FLOAT</td>
<td>float</td>
</tr>
<tr>
<td>REAL</td>
<td>real</td>
</tr>
<tr>
<td>CHAR(n)</td>
<td>char(n) if n is less than 254</td>
</tr>
<tr>
<td></td>
<td>varchar(n) if n is greater than or equal to 254 but less than 4000</td>
</tr>
<tr>
<td></td>
<td>longtext if n is greater than or equal to 4000</td>
</tr>
<tr>
<td>VARCHAR(n)</td>
<td>varchar(n) if n is less than 4000</td>
</tr>
<tr>
<td></td>
<td>longtext if n is greater than or equal to 4000</td>
</tr>
<tr>
<td>LONG VARCHAR</td>
<td>longtext</td>
</tr>
<tr>
<td>BINARY(n), VARBINARY(n)</td>
<td>varbinary(n) if n is less than 4000</td>
</tr>
<tr>
<td></td>
<td>longblob if n is greater than or equal to 4000</td>
</tr>
<tr>
<td>LONG BINARY</td>
<td>longblob</td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th>SQL Anywhere data type</th>
<th>Microsoft Access default data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIT, TINYINT</td>
<td>TINYINT</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>INT, INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td>BIGINT</td>
<td>DECIMAL(19,0)</td>
</tr>
<tr>
<td>DECIMAL(p,s), NUMERIC(p,s)</td>
<td>DECIMAL(p,s)</td>
</tr>
<tr>
<td>DATE, TIME, DATETIME, TIMESTAMP</td>
<td>DATETIME</td>
</tr>
<tr>
<td>MONEY, SMALLMONEY</td>
<td>MONEY</td>
</tr>
<tr>
<td>FLOAT</td>
<td>FLOAT</td>
</tr>
<tr>
<td>REAL</td>
<td>REAL</td>
</tr>
<tr>
<td>CHAR(n), VARCHAR(n)</td>
<td>CHARACTER(n) if n is less than 254</td>
</tr>
<tr>
<td></td>
<td>TEXT if n is greater than or equal to 254</td>
</tr>
<tr>
<td>LONG VARCHAR</td>
<td>TEXT</td>
</tr>
<tr>
<td>BINARY, VARBINARY</td>
<td>BINARY(n) if n is less than 4000</td>
</tr>
<tr>
<td></td>
<td>IMAGE if n is greater than or equal to 4000</td>
</tr>
<tr>
<td>LONG BINARY</td>
<td>IMAGE</td>
</tr>
</tbody>
</table>

### 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 797
- “Microsoft FoxPro (Microsoft 3.51.171300)” on page 797
- “Lotus Notes SQL 2.0” on page 798

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, an ODBC data source 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

You can obtain this driver from the Lotus web site, 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 » UserID » 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.
**JDBC-based server classes**

JDBC-based server classes are used when SQL Anywhere internally uses a Java Virtual Machine 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).

**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**

A server with server class sajdbc is a SQL Anywhere server. No special requirements exist for the configuration of a SQL Anywhere data source.

**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:

```sql
CREATE SERVER TestSAdb1
CLASS 'sajdbc'
USING 'banana:2638/db1'
CREATE SERVER TestSAdb2
CLASS 'sajdbc'
USING 'banana:2638/db2'
CREATE SERVER TestSAdb3
CLASS 'sajdbc'
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 asejdbc

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_identifiers option controls the use of quoted identifiers for Adaptive Server Enterprise. For example, if you set the quoted_identifiers 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.

<table>
<thead>
<tr>
<th>SQL Anywhere data type</th>
<th>Adaptive Server Enterprise default data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIT</td>
<td>bit</td>
</tr>
<tr>
<td>TINYINT</td>
<td>tinyint</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>smallint</td>
</tr>
<tr>
<td>INT</td>
<td>int</td>
</tr>
<tr>
<td>INTEGER</td>
<td>integer</td>
</tr>
<tr>
<td>DECIMAL [defaults p=30, s=6]</td>
<td>numeric(30,6)</td>
</tr>
<tr>
<td>DECIMAL(128,128)</td>
<td>not supported</td>
</tr>
<tr>
<td>NUMERIC [defaults p=30 s=6]</td>
<td>numeric(30,6)</td>
</tr>
<tr>
<td>NUMERIC(128,128)</td>
<td>not supported</td>
</tr>
<tr>
<td>FLOAT</td>
<td>real</td>
</tr>
<tr>
<td>REAL</td>
<td>real</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>float</td>
</tr>
<tr>
<td>SMALLMONEY</td>
<td>numeric(10,4)</td>
</tr>
<tr>
<td>MONEY</td>
<td>numeric(19,4)</td>
</tr>
<tr>
<td>DATE</td>
<td>datetime</td>
</tr>
<tr>
<td>TIME</td>
<td>datetime</td>
</tr>
<tr>
<td>SQL Anywhere data type</td>
<td>Adaptive Server Enterprise default data type</td>
</tr>
<tr>
<td>------------------------</td>
<td>---------------------------------------------</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>datetime</td>
</tr>
<tr>
<td>SMALLDATETIME</td>
<td>datetime</td>
</tr>
<tr>
<td>DATETIME</td>
<td>datetime</td>
</tr>
<tr>
<td>CHAR(n)</td>
<td>varchar(n)</td>
</tr>
<tr>
<td>CHARACTER(n)</td>
<td>varchar(n)</td>
</tr>
<tr>
<td>VARCHAR(n)</td>
<td>varchar(n)</td>
</tr>
<tr>
<td>CHARACTER VARYING(n)</td>
<td>varchar(n)</td>
</tr>
<tr>
<td>LONG VARCHAR</td>
<td>text</td>
</tr>
<tr>
<td>TEXT</td>
<td>text</td>
</tr>
<tr>
<td>BINARY(n)</td>
<td>binary(n)</td>
</tr>
<tr>
<td>LONG BINARY</td>
<td>image</td>
</tr>
<tr>
<td>IMAGE</td>
<td>image</td>
</tr>
<tr>
<td>BIGINT</td>
<td>numeric(19,0)</td>
</tr>
</tbody>
</table>
Part VIII. Stored Procedures and Triggers

This part 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 part also describes how to use the SQL Anywhere debugger—a powerful tool for debugging all kinds of logic.
CHAPTER 20

Using procedures, triggers, and batches

Contents

Procedure and trigger overview ................................................................. 806
Benefits of procedures and triggers .......................................................... 807
Introduction to procedures ........................................................................ 808
Introduction to user-defined functions ....................................................... 814
Introduction to triggers ............................................................................ 818
Introduction to batches ............................................................................. 827
Control statements ................................................................................... 830
The structure of procedures and triggers .................................................. 833
Returning results from procedures ............................................................ 836
Using cursors in procedures and triggers ................................................ 841
Errors and warnings in procedures and triggers ........................................ 844
Using the EXECUTE IMMEDIATE statement in procedures ...................... 851
Transactions and savepoints in procedures and triggers ....................... 852
Tips for writing procedures ...................................................................... 853
Statements allowed in procedures, triggers, events, and batches .......... 855
Hiding the contents of procedures, functions, triggers and views ............ 856
Procedure and trigger overview

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 857.

You can profile stored procedures to analyze performance characteristics in Sybase Central. See “Procedure profiling using system procedures” on page 219.
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 a number of 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

To use procedures, you need to understand how to:

- Create procedures
- Call procedures from a database application
- Drop or remove procedures
- Control who has permissions to use procedures

This section discusses the above aspects of using procedures, as well as some different applications of procedures.

Creating procedures

SQL Anywhere provides a number of tools that let you create a new procedure.

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. Connect to the database as a user with DBA or Resource authority.
2. In the left pane, double-click Procedures & Functions.
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. In the case of 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:

- “Creating remote procedures” on page 775
- “Creating remote servers using Sybase Central” on page 758

**See also**

- “Connecting to a database” [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 831
- “Creating remote procedures” on page 775

**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. 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 Generator** 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**
- “Setting properties for database objects” on page 36
- “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 808
- “Using Sybase Central to translate stored procedures” on page 653

**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:

```sql
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.

```sql
SELECT t.ID, t.QuantityOrdered AS q
FROM ShowCustomerProducts( 149 ) t;
```

**See also**
- “Database permissions and authorities overview” [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. Connect to the database as a DBA user 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

- “Connecting to a database” [SQL Anywhere Server - Database Administration]
- “DROP 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:

   ```sql
   CREATE PROCEDURE AverageSalary( OUT avgsal NUMERIC(20,3) )
   BEGIN
   SELECT AVG( Salary )
   INTO avgsal
   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:

   ```sql
   CREATE VARIABLE Average NUMERIC(20,3);
   ```

4. Call the procedure using the created variable to hold the result:

   ```sql
   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:

   ```sql
   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
- “Connecting to a database” [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:

```sql
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 );
```

<table>
<thead>
<tr>
<th>Employee ID</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>102</td>
<td>45700.000</td>
</tr>
<tr>
<td>105</td>
<td>62000.000</td>
</tr>
<tr>
<td>160</td>
<td>57490.000</td>
</tr>
<tr>
<td>243</td>
<td>72995.000</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

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 839
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. 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]
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:

```sql
SELECT FullName(GivenName, Surname)
AS "Full Name"
FROM Employees;
```

<table>
<thead>
<tr>
<th>Full Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fran Whitney</td>
</tr>
<tr>
<td>Matthew Cobb</td>
</tr>
<tr>
<td>Philip Chin</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

The following statement in Interactive SQL returns a full name from a supplied first and last name:

```sql
SELECT FullName('Jane', 'Smith')
AS "Full Name";
```

<table>
<thead>
<tr>
<th>Full Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jane Smith</td>
</tr>
</tbody>
</table>

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 some Canadian customers as well as customers from the USA. The user-defined function Nationality forms a 3-letter country code based on the Country column.

```sql
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;
```
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:

```sql
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:

```sql
DROP FUNCTION FullName;
```

**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:

```sql
GRANT EXECUTE ON Nationality TO BobS;
```

The following statement revokes permissions to use the function:

```sql
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 584.
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 93, 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 server option” [SQL Anywhere Server - Database Administration]
- “fire_triggers option [compatibility]” [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 824.

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 particularly 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 back to their pre-operation state. See “Atomic compound statements” on page 831.

### 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. Connect to the database as a user with DBA or Resource authority.
2. In the left pane, click Triggers.
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, as well as any previous effects of the trigger, 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.

You may find it easier in some cases 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

- “Connecting to a database” [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 831
## 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. 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 Generator** 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**
- “Setting properties for database objects” on page 36
- “Connecting to a database” [SQL Anywhere Server - Database Administration]
- “Using Sybase Central to translate stored procedures” on page 653
- “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. 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**
- “Connecting to a database” [SQL Anywhere Server - Database Administration]
- “DROP 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 overview” [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 or not 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 the purposes of checking updatability for 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 semi-colons, in which case the batch is conforming to the Watcom-SQL dialect. A multi-statement batch that does not use semi-colons 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 652.

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
           WHERE EmployeeID=207;

Copyright © 2008, iAnywhere Solutions, Inc. - SQL Anywhere 11.0.0
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 a number of control statements for logical flow and decision making in the body of the procedure or trigger, or in a batch. Available control statements include:

<table>
<thead>
<tr>
<th>Control statement</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compound statements</td>
<td>BEGIN [ ATOMIC ] Statement-list END</td>
</tr>
<tr>
<td>See “BEGIN statement” [SQL Anywhere Server - SQL Reference].</td>
<td></td>
</tr>
<tr>
<td>Conditional execution: IF</td>
<td>IF condition THEN Statement-list ELSEIF condition THEN Statement-list ELSE Statement-list END IF</td>
</tr>
<tr>
<td>See “IF statement” [SQL Anywhere Server - SQL Reference].</td>
<td></td>
</tr>
<tr>
<td>Conditional execution: CASE</td>
<td>CASE expression WHEN value THEN Statement-list WHEN value THEN Statement-list ELSE Statement-list END CASE</td>
</tr>
<tr>
<td>See “CASE statement” [SQL Anywhere Server - SQL Reference].</td>
<td></td>
</tr>
<tr>
<td>Repetition: WHILE, LOOP</td>
<td>WHILE condition LOOP Statement-list END LOOP</td>
</tr>
<tr>
<td>See “LOOP statement” [SQL Anywhere Server - SQL Reference].</td>
<td></td>
</tr>
<tr>
<td>Repetition: FOR cursor loop</td>
<td>FOR loop-name AS cursor-name CURSOR FOR select-statement DO Statement-list END FOR</td>
</tr>
<tr>
<td>See “FOR statement” [SQL Anywhere Server - SQL Reference].</td>
<td></td>
</tr>
<tr>
<td>Break: LEAVE</td>
<td>LEAVE label</td>
</tr>
<tr>
<td>See “LEAVE statement” [SQL Anywhere Server - SQL Reference].</td>
<td></td>
</tr>
<tr>
<td>CALL</td>
<td>CALL procname( arg, ... )</td>
</tr>
<tr>
<td>See “CALL statement” [SQL Anywhere Server - SQL Reference].</td>
<td></td>
</tr>
</tbody>
</table>
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 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.

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 852).

There is a case where some, but not all, of the 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 848.
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 831. 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 (see “SQL data types” [SQL Anywhere Server - SQL Reference]), 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:

```sql
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.

```sql
CALL CustomerProducts();
```
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 814 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;
  ```

<table>
<thead>
<tr>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fran Whitney</td>
</tr>
</tbody>
</table>

Copyright © 2008, iAnywhere Solutions, Inc. - SQL Anywhere 11.0.0
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 808.

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();
```

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 somewhat artificial 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 837.

If the query results in no rows being selected, 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.

```sql
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;
```

- Type the following:

```
CALL ListCustomerValue ( );
```

<table>
<thead>
<tr>
<th>Company</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Hat Company</td>
<td>5016</td>
</tr>
<tr>
<td>The Igloo</td>
<td>3564</td>
</tr>
<tr>
<td>The Ultimate</td>
<td>3348</td>
</tr>
<tr>
<td>North Land Trading</td>
<td>3144</td>
</tr>
<tr>
<td>Molly's</td>
<td>2808</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

**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 set, 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 (Sybase Central)**

1. Connect to the database as a user with DBA authority.
2. In the left pane, select the database and choose **File» Open Interactive SQL**.
3. In Interactive SQL, choose **Tools» Options**.
4. Click **SQL Anywhere**.
5. On the **Results** tab, select **Show All Result Sets**.
6. Click **OK**.

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:

```sql
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:

```sql
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:

```sql
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;
```

The use of variable result sets in procedures is subject to some limitations, depending on the interface used by the client application.

- **Embedded SQL** You must DESCRIBE the procedure call after the cursor for the result set is opened, but before any rows are returned, in order to get the proper shape of result set.

  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 837, it illustrates several features of the stored procedure language.

```sql
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
```

Copyright © 2008, iAnywhere Solutions, Inc. - SQL Anywhere 11.0.0 841
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 844.

- 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 TopCustomerValue 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 SQLCODE or SQLSTATE status descriptions. For full descriptions of SQLCODE and SQLSTATE error and warning values and their meanings, see Error Messages [Error Messages].

Whenever a SQL statement executes, a value appears in special procedure variables called SQLSTATE and SQLCODE. That value indicates whether or not there were any unusual conditions encountered while the statement was being performed. 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 841 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 848.

Warnings are handled in a slightly different manner from errors: for a description, see “Default handling of warnings in procedures and triggers” on page 847.

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 [compatibility]” [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. In the case of a trigger, 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.

```sql
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 [compatibility]” [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:

```sql
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:

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 \texttt{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 844.

In this case, the SIGNAL statement generates a row not found condition, which is a warning rather than an error.

```sql
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 831). 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.

The demonstration procedures used to illustrate exception handling are based on those used in “Default error handling in procedures and triggers” on page 844.

In this example, additional code handles the column not found error in the InnerProc procedure.

```sql
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 this 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 simply a RESIGNAL statement, control passes back 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 of the statements in the atomic compound statement are executed.

**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 844.

```sql
DROP PROCEDURE OuterProc;
DROP PROCEDURE InnerProc;

CREATE PROCEDURE InnerProc()
BEGIN
  BEGIN
    DECLARE column_not_found
```

Copyright © 2008, iAnywhere Solutions, Inc. - SQL Anywhere 11.0.0
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.

```sql
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:

```sql
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:

```sql
CALL DynamicResult('table_id,table_name', 'SYSTAB', 'table_id <= 10');
```

<table>
<thead>
<tr>
<th>table_id</th>
<th>table_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ISYSTAB</td>
</tr>
<tr>
<td>2</td>
<td>ISYSTABCOL</td>
</tr>
<tr>
<td>3</td>
<td>ISYSIDX</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

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 119
- “Atomic compound statements” on page 831
- “Savepoints within transactions” on page 123
**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`:

```sql
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 860.
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 DECRIPTED FILE
- CREATE ENCRYPTED FILE
- DISCONNECT
- DROP CONNECTION
- DROP DATABASE
- FORWARD TO
- Interactive SQL commands such as INPUT or OUTPUT
- PREPARE TO COMMIT
- STOP ENGINE

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 852.

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:

```sql
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

In some cases, 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 scrambles 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]
CHAPTER 21

Debugging procedures, functions, triggers, and events

Contents

Introduction to the SQL Anywhere debugger ............................................................ 858
Tutorial: Getting started with the debugger ............................................................... 859
Working with breakpoints .......................................................................................... 863
Working with variables ............................................................................................... 866
Working with connections .......................................................................................... 868
Introduction to the SQL Anywhere debugger

You can use the SQL Anywhere debugger during the development of SQL stored procedures, triggers, event handlers, and user-defined functions.

You can perform many tasks with the SQL Anywhere debugger, including:

- **Debug procedures and triggers** You can debug SQL stored procedures or triggers.
- **Debug event handlers** Event handlers are an extension of SQL stored procedures. The material in this chapter 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** When execution is stopped at a breakpoint in a SQL procedure, you can execute queries. This permits you to look at intermediate results held in temporary tables, as well as to check values in base tables and to view the query execution plan.

Tip
By default, SOAP connections time out after 60 seconds before they time out. You can specify -xs http(kto=0) so that the connection does not time out when you are trying to debug SOAP functions and procedures. See “-xs server option” [SQL Anywhere Server - Database Administration].

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.
Tutorial: Getting started with the debugger

This tutorial describes how to connect to a database, how to start the debugger, and how to debug a simple stored procedure.

This tutorial uses the SQL Anywhere sample database, demo.db. This file is located in the SQL Anywhere samples directory, samples-dir. For more information about samples-dir, see “Samples directory” [SQL Anywhere Server - Database Administration].

Lesson 1: Connect to a database and start the debugger

To start the debugger

1. Click Start » Programs » SQL Anywhere 11 » Sybase Central.
2. From the Connections menu, choose Connect With SQL Anywhere 11.
3. In the ODBC Data Source Name field, type SQL Anywhere 11 Demo
4. Click OK.
5. Choose Mode » Debug.

The Specify User To Debug window appears.

6. 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.
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:

<table>
<thead>
<tr>
<th>top_company</th>
<th>top_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>(NULL)</td>
<td>(NULL)</td>
</tr>
</tbody>
</table>

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:

   ```sql
   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:

   ```sql
   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:

```sql
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.

<table>
<thead>
<tr>
<th>top_company</th>
<th>top_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chadwicks</td>
<td>8076</td>
</tr>
</tbody>
</table>

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:

```sql
OPEN cursor_this_customer;
```
3. Type a new line that initializes the top_value variable:

```sql
SET top_value = 0;
```
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.

Setting 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. Connect to the database as a user with DBA authority.
2. In the left pane, double-click Procedures & Functions.
3. Select a procedure.
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
   - A red circle appears to the left of the line of code.

To set a breakpoint (Debug menu)

1. Choose Debug » Breakpoints.
   - The Breakpoints window appears.
2. Click New.
   - The New Breakpoint window appears.
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:

\[
\text{CURRENT USER} = 'user-name'
\]

The Count is a 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.

**Disabling and enabling 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. 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. 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**.

**Editing 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. 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:

   ```sql
   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.

Local variables

To view variable values

1. Connect to the database as a user with DBA authority.
2. In the left pane, double-click Procedures & Functions.
3. Select a procedure.
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.

Other 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 818.

Static variables are used in Java classes. They are appear on the Statics tab.

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. 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
   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:

\[
\text{CURRENT USER } = \text{'user-name'}
\]

The SQL special value CURRENT USER holds the user ID of the connection.

For more information, see “Editing breakpoint conditions” on page 864, and “CURRENT USER special value” [SQL Anywhere Server - SQL Reference].
Part IX. Glossary
CHAPTER 22

Glossary

Adaptive Server Anywhere (ASA)
The relational database server component of SQL Anywhere Studio, intended for use in mobile and embedded environments or as a server for small and medium-sized businesses. In version 10.0.0, Adaptive Server Anywhere was renamed SQL Anywhere Server, and SQL Anywhere Studio was renamed SQL Anywhere.

See also: “SQL Anywhere” on page 894.

agent ID

See also: “client message store ID” on page 873.

article

In MobiLink or SQL Remote, an article is a database object that represents a whole table, or a subset of the columns and rows in a table. Articles are grouped together in a publication.

See also:

● “replication” on page 892
● “publication” on page 889

atomic transaction

A transaction that is guaranteed to complete successfully or not at all. If an error prevents part of an atomic transaction from completing, the transaction is rolled back to prevent the database from being left in an inconsistent state.

base table

Permanent tables for data. Tables are sometimes called base tables to distinguish them from temporary tables and views.

See also:

● “temporary table” on page 896
● “view” on page 898

bit array

A bit array is a type of array data structure that is used for efficient storage of a sequence of bits. A bit array is similar to a character string, except that the individual pieces are 0s (zeros) and 1s (ones) instead of characters. Bit arrays are typically used to hold a string of Boolean values.
**business rule**

A guideline based on real-world requirements. Business rules are typically implemented through check constraints, user-defined data types, and the appropriate use of transactions.

See also:
- “constraint” on page 874
- “user-defined data type” on page 898

**carrier**

A MobiLink object, stored in MobiLink system tables or a Notifier properties file, that contains information about a public carrier for use by server-initiated synchronization.

See also: “server-initiated synchronization” on page 894.

**character set**

A character set is a set of symbols, including letters, digits, spaces, and other symbols. An example of a character set is ISO-8859-1, also known as Latin1.

See also:
- “code page” on page 873
- “encoding” on page 878
- “collation” on page 873

**check constraint**

A restriction that enforces specified conditions on a column or set of columns.

See also:
- “constraint” on page 874
- “foreign key constraint” on page 880
- “primary key constraint” on page 889
- “unique constraint” on page 897

**checkpoint**

The point at which all changes to the database are saved to the database file. At other times, committed changes are saved only to the transaction log.

**checksum**

The calculated number of bits of a database page that is recorded with the database page itself. The checksum allows the database management system to validate the integrity of the page by ensuring that the numbers match as the page is being written to disk. If the counts match, it's assumed that page was successfully written.

**client message store**

In QAnywhere, a SQL Anywhere database on the remote device that stores messages.
client message store ID

In QAnywhere, a MobiLink remote ID that uniquely identifies a client message store.

client/server

A software architecture where one application (the client) obtains information from and sends information to another application (the server). The two applications often reside on different computers connected by a network.

code page

A code page is an encoding that maps characters of a character set to numeric representations, typically an integer between 0 and 255. An example of a code page is Windows code page 1252. For the purposes of this documentation, code page and encoding are interchangeable terms.

See also:
• “character set” on page 872
• “encoding” on page 878
• “collation” on page 873

collation

A combination of a character set and a sort order that defines the properties of text in the database. For SQL Anywhere databases, the default collation is determined by the operating system and language on which the server is running; for example, the default collation on English Windows systems is 1252LATIN1. A collation, also called a collating sequence, is used for comparing and sorting strings.

See also:
• “character set” on page 872
• “code page” on page 873
• “encoding” on page 878

command file

A text file containing SQL statements. Command files can be built manually, or they can be built automatically by database utilities. The dbunload utility, for example, creates a command file consisting of the SQL statements necessary to recreate a given database.

communication stream

In MobiLink, the network protocol used for communication between the MobiLink client and the MobiLink server.

concurrency

The simultaneous execution of two or more independent, and possibly competing, processes. SQL Anywhere automatically uses locking to isolate transactions and ensure that each concurrent application sees a consistent set of data.
See also:
- “transaction” on page 896
- “isolation level” on page 882

**conflict resolution**

In MobiLink, conflict resolution is logic that specifies what to do when two users modify the same row on different remote databases.

**connection ID**

A unique number that identifies a given connection between a client application and the database. You can determine the current connection ID using the following SQL statement:

```
SELECT CONNECTION_PROPERTY( 'Number' );
```

**connection-initiated synchronization**

A form of MobiLink server-initiated synchronization in which synchronization is initiated when there are changes to connectivity.

See also: “server-initiated synchronization” on page 894.

**connection profile**

A set of parameters that are required to connect to a database, such as user name, password, and server name, that is stored and used as a convenience.

**consolidated database**

In distributed database environments, a database that stores the master copy of the data. In case of conflict or discrepancy, the consolidated database is considered to have the primary copy of the data.

See also:
- “synchronization” on page 896
- “replication” on page 892

**constraint**

A restriction on the values contained in a particular database object, such as a table or column. For example, a column may have a uniqueness constraint, which requires that all values in the column be different. A table may have a foreign key constraint, which specifies how the information in the table relates to data in some other table.

See also:
- “check constraint” on page 872
- “foreign key constraint” on page 880
- “primary key constraint” on page 889
- “unique constraint” on page 897
contention

The act of competing for resources. For example, in database terms, two or more users trying to edit the same row of a database contend for the rights to edit that row.

correlation name

The name of a table or view that is used in the FROM clause of a query—either its original name, or an alternate name, that is defined in the FROM clause.

creator ID

In UltraLite Palm OS applications, an ID that is assigned when the application is created.

cursor

A named linkage to a result set, used to access and update rows from a programming interface. In SQL Anywhere, cursors support forward and backward movement through the query results. Cursors consist of two parts: the cursor result set, typically defined by a SELECT statement; and the cursor position.

See also:
- “cursor result set” on page 875
- “cursor position” on page 875

cursor position

A pointer to one row within the cursor result set.

See also:
- “cursor” on page 875
- “cursor result set” on page 875

cursor result set

The set of rows resulting from a query that is associated with a cursor.

See also:
- “cursor” on page 875
- “cursor position” on page 875

data cube

A multi-dimensional result set with each dimension reflecting a different way to group and sort the same results. Data cubes provide complex information about data that would otherwise require self-join queries and correlated subqueries. Data cubes are a part of OLAP functionality.

data definition language (DDL)

The subset of SQL statements for defining the structure of data in the database. DDL statements create, modify, and remove database objects, such as tables and users.
**data manipulation language (DML)**

The subset of SQL statements for manipulating data in the database. DML statements retrieve, insert, update, and delete data in the database.

**data type**

The format of data, such as CHAR or NUMERIC. In the ANSI SQL standard, data types can also include a restriction on size, character set, and collation.

See also: “domain” on page 878.

**database**

A collection of tables that are related by primary and foreign keys. The tables hold the information in the database. The tables and keys together define the structure of the database. A database management system accesses this information.

See also:
- “foreign key” on page 879
- “primary key” on page 889
- “database management system (DBMS)” on page 876
- “relational database management system (RDBMS)” on page 891

**database administrator (DBA)**

The user with the permissions required to maintain the database. The DBA is generally responsible for all changes to a database schema, and for managing users and groups. The role of database administrator is automatically built into databases as user ID DBA with password sql.

**database connection**

A communication channel between a client application and the database. A valid user ID and password are required to establish a connection. The privileges granted to the user ID determine the actions that can be carried out during the connection.

**database file**

A database is held in one or more database files. There is an initial file, and subsequent files are called dbspaces. Each table, including its indexes, must be contained within a single database file.

See also: “dbspace” on page 877.

**database management system (DBMS)**

A collection of programs that allow you to create and use databases.

See also: “relational database management system (RDBMS)” on page 891.
database name

The name given to a database when it is loaded by a server. The default database name is the root of the initial database file.

See also: “database file” on page 876.

database object

A component of a database that contains or receives information. Tables, indexes, views, procedures, and triggers are database objects.

database owner (dbo)

A special user that owns the system objects not owned by SYS.

See also:

● “database administrator (DBA)” on page 876
● “SYS” on page 896

database server

A computer program that regulates all access to information in a database. SQL Anywhere provides two types of servers: network servers and personal servers.

DBA authority

The level of permission that enables a user to do administrative activity in the database. The DBA user has DBA authority by default.

See also: “database administrator (DBA)” on page 876.

dbspace

An additional database file that creates more space for data. A database can be held in up to 13 separate files (an initial file and 12 dbspaces). Each table, together with its indexes, must be contained in a single database file. The SQL command CREATE DBSPACE adds a new file to the database.

See also: “database file” on page 876.

deadlock

A state where a set of transactions arrives at a place where none can proceed.

device tracking

Functionality in MobiLink server-initiated synchronization that allows you to address messages using the MobiLink user name that identifies a remote device.

See also: “server-initiated synchronization” on page 894.
direct row handling

In MobiLink, a way to synchronize table data to sources other than the MobiLink-supported consolidated databases. You can implement both uploads and downloads with direct row handling.

See also:
- “consolidated database” on page 874
- “SQL-based synchronization” on page 894

domain

Aliases for built-in data types, including precision and scale values where applicable, and optionally including DEFAULT values and CHECK conditions. Some domains, such as the monetary data types, are pre-defined in SQL Anywhere. Also called user-defined data type.

See also: “data type” on page 876.

download

The stage in synchronization where data is transferred from the consolidated database to a remote database.

dynamic SQL

SQL that is generated programmatically by your program before it is executed. UltraLite dynamic SQL is a variant designed for small-footprint devices.

EBF

Express Bug Fix. An express bug fix is a subset of the software with one or more bug fixes. The bug fixes are listed in the release notes for the update. Bug fix updates may only be applied to installed software with the same version number. Some testing has been performed on the software, but the software has not undergone full testing. You should not distribute these files with your application unless you have verified the suitability of the software yourself.

embedded SQL

A programming interface for C programs. SQL Anywhere embedded SQL is an implementation of the ANSI and IBM standard.

encoding

Also known as character encoding, an encoding is a method by which each character in a character set is mapped onto one or more bytes of information, typically represented as a hexadecimal number. An example of an encoding is UTF-8.

See also:
- “character set” on page 872
- “code page” on page 873
- “collation” on page 873
event model

In MobiLink, the sequence of events that make up a synchronization, such as begin_synchronization and download_cursor. Events are invoked if a script is created for them.

external login

An alternate login name and password used when communicating with a remote server. By default, SQL Anywhere uses the names and passwords of its clients whenever it connects to a remote server on behalf of those clients. However, this default can be overridden by creating external logins. External logins are alternate login names and passwords used when communicating with a remote server.

extraction

In SQL Remote replication, the act of unloading the appropriate structure and data from the consolidated database. This information is used to initialize the remote database.

See also: “replication” on page 892.

failover

Switching to a redundant or standby server, system, or network on failure or unplanned termination of the active server, system, or network. Failover happens automatically.

FILE

In SQL Remote replication, a message system that uses shared files for exchanging replication messages. This is useful for testing and for installations without an explicit message-transport system.

See also: “replication” on page 892

file-based download

In MobiLink, a way to synchronize data in which downloads are distributed as files, allowing offline distribution of synchronization changes.

file-definition database

In MobiLink, a SQL Anywhere database that is used for creating download files.

See also: “file-based download” on page 879.

foreign key

One or more columns in a table that duplicate the primary key values in another table. Foreign keys establish relationships between tables.

See also:

● “primary key” on page 889
● “foreign table” on page 880
foreign key constraint

A restriction on a column or set of columns that specifies how the data in the table relates to the data in some other table. Imposing a foreign key constraint on a set of columns makes those columns the foreign key.

See also:
- “constraint” on page 874
- “check constraint” on page 872
- “primary key constraint” on page 889
- “unique constraint” on page 897

foreign table

The table containing the foreign key.

See also: “foreign key” on page 879.

full backup

A backup of the entire database, and optionally, the transaction log. A full backup contains all the information in the database and thus provides protection in the event of a system or media failure.

See also: “incremental backup” on page 881.

gateway

A MobiLink object, stored in MobiLink system tables or a Notifier properties file, that contains information about how to send messages for server-initiated synchronization.

See also: “server-initiated synchronization” on page 894.

generated join condition

A restriction on join results that is automatically generated. There are two types: key and natural. Key joins are generated when you specify KEY JOIN or when you specify the keyword JOIN but do not use the keywords CROSS, NATURAL, or ON. For a key join, the generated join condition is based on foreign key relationships between tables. Natural joins are generated when you specify NATURAL JOIN; the generated join condition is based on common column names in the two tables.

See also:
- “join” on page 883
- “join condition” on page 883

generation number

In MobiLink, a mechanism for forcing remote databases to upload data before applying any more download files.

See also: “file-based download” on page 879.
**global temporary table**

A type of temporary table for which data definitions are visible to all users until explicitly dropped. Global temporary tables let each user open their own identical instance of a table. By default, rows are deleted on commit, and rows are always deleted when the connection is ended.

See also:
- “temporary table” on page 896
- “local temporary table” on page 884

**grant option**

The level of permission that allows a user to grant permissions to other users.

**hash**

A hash is an index optimization that transforms index entries into keys. An index hash aims to avoid the expensive operation of finding, loading, and then unpacking the rows to determine the indexed value, by including enough of the actual row data with its row ID.

**histogram**

The most important component of column statistics, histograms are a representation of data distribution. SQL Anywhere maintains histograms to provide the optimizer with statistical information about the distribution of values in columns.

**iAnywhere JDBC driver**

The iAnywhere JDBC driver provides a JDBC driver that has some performance benefits and feature benefits compared to the pure Java jConnect JDBC driver, but which is not a pure-Java solution. The iAnywhere JDBC driver is recommended in most cases.

See also:
- “JDBC” on page 883
- “jConnect” on page 883

**identifier**

A string of characters used to reference a database object, such as a table or column. An identifier may contain any character from A through Z, a through z, 0 through 9, underscore (_), at sign (@), number sign (#), or dollar sign ($).

**incremental backup**

A backup of the transaction log only, typically used between full backups.

See also: “transaction log” on page 897.
index

A sorted set of keys and pointers associated with one or more columns in a base table. An index on one or more columns of a table can improve performance.

InfoMaker

A reporting and data maintenance tool that lets you create sophisticated forms, reports, graphs, cross-tabs, and tables, as well as applications that use these reports as building blocks.

inner join

A join in which rows appear in the result set only if both tables satisfy the join condition. Inner joins are the default.

See also:
- “join” on page 883
- “outer join” on page 887

integrated login

A login feature that allows the same single user ID and password to be used for operating system logins, network logins, and database connections.

integrity

Adherence to rules that ensure that data is correct and accurate, and that the relational structure of the database is intact.

See also: “referential integrity” on page 891.

Interactive SQL

A SQL Anywhere application that allows you to query and alter data in your database, and modify the structure of your database. Interactive SQL provides a pane for you to enter SQL statements, as well as panes that return information about how the query was processed and the result set.

isolation level

The degree to which operations in one transaction are visible to operations in other concurrent transactions. There are four isolation levels, numbered 0 through 3. Level 3 provides the highest level of isolation. Level 0 is the default setting. SQL Anywhere also supports three snapshot isolation levels: snapshot, statement-snapshot, and readonly-statement-snapshot.

See also: “snapshot isolation” on page 894.

JAR file

Java archive file. A compressed file format consisting of a collection of one or more packages used for Java applications. It includes all the resources necessary to install and run a Java program in a single compressed file.
Java class

The main structural unit of code in Java. It is a collection of procedures and variables grouped together because they all relate to a specific, identifiable category.

jConnect

A Java implementation of the JavaSoft JDBC standard. It provides Java developers with native database access in multi-tier and heterogeneous environments. However, the iAnywhere JDBC driver is the preferred JDBC driver for most cases.

See also:
- “JDBC” on page 883
- “iAnywhere JDBC driver” on page 881

JDBC

Java Database Connectivity. A SQL-language programming interface that allows Java applications to access relational data. The preferred JDBC driver is the iAnywhere JDBC driver.

See also:
- “jConnect” on page 883
- “iAnywhere JDBC driver” on page 881

join

A basic operation in a relational system that links the rows in two or more tables by comparing the values in specified columns.

join condition

A restriction that affects join results. You specify a join condition by inserting an ON clause or WHERE clause immediately after the join. In the case of natural and key joins, SQL Anywhere generates a join condition.

See also:
- “join” on page 883
- “generated join condition” on page 880

join type

SQL Anywhere provides four types of joins: cross join, key join, natural join, and joins using an ON clause.

See also: “join” on page 883.

Listener

A program, dbln, that is used by MobiLink server-initiated synchronization. Listeners are installed on remote devices and configured to initiate actions on the device when they receive information from a Notifier.
local temporary table

A type of temporary table that exists only for the duration of a compound statement or until the end of the connection. Local temporary tables are useful when you need to load a set of data only once. By default, rows are deleted on commit.

See also:
- “temporary table” on page 896
- “global temporary table” on page 881

lock

A concurrency control mechanism that protects the integrity of data during the simultaneous execution of multiple transactions. SQL Anywhere automatically applies locks to prevent two connections from changing the same data at the same time, and to prevent other connections from reading data that is in the process of being changed.

You control locking by setting the isolation level.

See also:
- “isolation level” on page 882
- “concurrency” on page 873
- “integrity” on page 882

log file

A log of transactions maintained by SQL Anywhere. The log file is used to ensure that the database is recoverable in the event of a system or media failure, to improve database performance, and to allow data replication using SQL Remote.

See also:
- “transaction log” on page 897
- “transaction log mirror” on page 897
- “full backup” on page 880

logical index

A reference (pointer) to a physical index. There is no indexing structure stored on disk for a logical index.

LTM

Log Transfer Manager (LTM) also called Replication Agent. Used with Replication Server, the LTM is the program that reads a database transaction log and sends committed changes to Sybase Replication Server.

See: “Replication Server” on page 892.
**maintenance release**

A maintenance release is a complete set of software that upgrades installed software from an older version with the same major version number (version number format is `major.minor.patch.build`). Bug fixes and other changes are listed in the release notes for the upgrade.

**materialized view**

A materialized view is a view that has been computed and stored on disk. Materialized views have characteristics of both views (they are defined using a query specification), and of tables (they allow most table operations to be performed on them).

See also:
- “base table” on page 871
- “view” on page 898

**message log**

A log where messages from an application such as a database server or MobiLink server can be stored. This information can also appear in a messages window or be logged to a file. The message log includes informational messages, errors, warnings, and messages from the MESSAGE statement.

**message store**

In QAnywhere, databases on the client and server device that store messages.

See also:
- “client message store” on page 872
- “server message store” on page 894

**message system**

In SQL Remote replication, a protocol for exchanging messages between the consolidated database and a remote database. SQL Anywhere includes support for the following message systems: FILE, FTP, and SMTP.

See also:
- “replication” on page 892
- “FILE” on page 879

**message type**

In SQL Remote replication, a database object that specifies how remote users communicate with the publisher of a consolidated database. A consolidated database may have several message types defined for it; this allows different remote users to communicate with it using different message systems.

See also:
- “replication” on page 892
- “consolidated database” on page 874
metadata

Data about data. Metadata describes the nature and content of other data.

See also: “schema” on page 893.

mirror log

See also: “transaction log mirror” on page 897.

MobiLink

A session-based synchronization technology designed to synchronize UltraLite and SQL Anywhere remote databases with a consolidated database.

See also:
- “consolidated database” on page 874
- “synchronization” on page 896
- “UltraLite” on page 897

MobiLink client

There are two kinds of MobiLink clients. For SQL Anywhere remote databases, the MobiLink client is the dbmlsync command line utility. For UltraLite remote databases, the MobiLink client is built in to the UltraLite runtime library.

MobiLink Monitor

A graphical tool for monitoring MobiLink synchronizations.

MobiLink server

The computer program that runs MobiLink synchronization, mlsrv11.

MobiLink system table

System tables that are required by MobiLink synchronization. They are installed by MobiLink setup scripts into the MobiLink consolidated database.

MobiLink user

A MobiLink user is used to connect to the MobiLink server. You create the MobiLink user on the remote database and register it in the consolidated database. MobiLink user names are entirely independent of database user names.

network protocol

The type of communication, such as TCP/IP or HTTP.
network server

A database server that accepts connections from computers sharing a common network.

See also: “personal server” on page 888.

normalization

The refinement of a database structure to eliminate redundancy and improve organization according to rules based on relational database theory.

Notifier

A program that is used by MobiLink server-initiated synchronization. Notifiers run on the same computer as the MobiLink server. They poll the consolidated database for push requests, and then send notifications to Listeners.

See also:
- “server-initiated synchronization” on page 894
- “Listener” on page 883

object tree

In Sybase Central, the hierarchy of database objects. The top level of the object tree shows all products that your version of Sybase Central supports. Each product expands to reveal its own sub-tree of objects.

See also: “Sybase Central” on page 896.

ODBC

Open Database Connectivity. A standard Windows interface to database management systems. ODBC is one of several interfaces supported by SQL Anywhere.

ODBC Administrator

A Microsoft program included with Windows operating systems for setting up ODBC data sources.

ODBC data source

A specification of the data a user wants to access via ODBC, and the information needed to get to that data.

outer join

A join that preserves all the rows in a table. SQL Anywhere supports left, right, and full outer joins. A left outer join preserves the rows in the table to the left of the join operator, and returns a null when a row in the right table does not satisfy the join condition. A full outer join preserves all the rows from both tables.

See also:
- “join” on page 883
- “inner join” on page 882
package

In Java, a collection of related classes.

parse tree

An algebraic representation of a query.

PDB

A Palm database file.

performance statistic

A value reflecting the performance of the database system. The CURRREAD statistic, for example, represents the number of file reads issued by the engine that have not yet completed.

personal server

A database server that runs on the same computer as the client application. A personal database server is typically used by a single user on a single computer, but it can support several concurrent connections from that user.

physical index

The actual indexing structure of an index, as it is stored on disk.

plug-in module

In Sybase Central, a way to access and administer a product. Plug-ins are usually installed and registered automatically with Sybase Central when you install the respective product. Typically, a plug-in appears as a top-level container, in the Sybase Central main window, using the name of the product itself; for example, SQL Anywhere.

See also: “Sybase Central” on page 896.

policy

In QAnywhere, the way you specify when message transmission should occur.

polling

In MobiLink server-initiated synchronization, the way the Notifier detects push requests on the consolidated database.

See also: “server-initiated synchronization” on page 894.

PowerDesigner

A database modeling application. PowerDesigner provides a structured approach to designing a database or data warehouse. SQL Anywhere includes the Physical Data Model component of PowerDesigner.
**PowerJ**

A Sybase product for developing Java applications.

**predicate**

A conditional expression that is optionally combined with the logical operators AND and OR to make up the set of conditions in a WHERE or HAVING clause. In SQL, a predicate that evaluates to UNKNOWN is interpreted as FALSE.

**primary key**

A column or list of columns whose values uniquely identify every row in the table.

See also: “foreign key” on page 879.

**primary key constraint**

A uniqueness constraint on the primary key columns. A table can have only one primary key constraint.

See also:
- “constraint” on page 874
- “check constraint” on page 872
- “foreign key constraint” on page 880
- “unique constraint” on page 897
- “integrity” on page 882

**primary table**

The table containing the primary key in a foreign key relationship.

**proxy table**

A local table containing metadata used to access a table on a remote database server as if it were a local table.

See also: “metadata” on page 886.

**publication**

In MobiLink or SQL Remote, a database object that identifies data that is to be synchronized. In MobiLink, publications exist only on the clients. A publication consists of articles. SQL Remote users can receive a publication by subscribing to it. MobiLink users can synchronize a publication by creating a synchronization subscription to it.

See also:
- “replication” on page 892
- “article” on page 871
- “publication update” on page 890
**publication update**

In SQL Remote replication, a list of changes made to one or more publications in one database. A publication update is sent periodically as part of a replication message to the remote database(s).

See also:
- “replication” on page 892
- “publication” on page 889

**publisher**

In SQL Remote replication, the single user in a database who can exchange replication messages with other replicating databases.

See also: “replication” on page 892.

**push notification**

In QAnywhere, a special message delivered from the server to a QAnywhere client that prompts the client to initiate a message transmission.

See also: “QAnywhere” on page 890.

**push request**

In MobiLink server-initiated synchronization, a row in a SQL result set or table on the consolidated database that contains a notification and information about how to send the notification.

See also: “server-initiated synchronization” on page 894.

**QAnywhere**

Application-to-application messaging, including mobile device to mobile device and mobile device to and from the enterprise, that permits communication between custom programs running on mobile or wireless devices and a centrally located server application.

**QAnywhere agent**

In QAnywhere, a process running on the client device that monitors the client message store and determines when message transmission should occur.

**query**

A SQL statement or group of SQL statements that access and/or manipulate data in a database.

See also: “SQL” on page 894.

**Redirector**

A web server plug-in that routes requests and responses between a client and the MobiLink server. This plug-in also implements load-balancing and failover mechanisms.
reference database

In MobiLink, a SQL Anywhere database used in the development of UltraLite clients. You can use a single SQL Anywhere database as both reference and consolidated database during development. Databases made with other products cannot be used as reference databases.

referencing object

An object, such as a view, whose definition directly references another object in the database, such as a table.

See also: “foreign key” on page 879.

referenced object

An object, such as a table, that is directly referenced in the definition of another object, such as a view.

See also: “primary key” on page 889.

referential integrity

Adherence to rules governing data consistency, specifically the relationships between the primary and foreign key values in different tables. To have referential integrity, the values in each foreign key must correspond to the primary key values of a row in the referenced table.

See also:
- “primary key” on page 889
- “foreign key” on page 879

regular expression

A regular expression is a sequence of characters, wildcards, and operators that defines a pattern to search for within a string.

relational database management system (RDBMS)

A type of database management system that stores data in the form of related tables.

See also: “database management system (DBMS)” on page 876.

remote database

In MobiLink or SQL Remote, a database that exchanges data with a consolidated database. Remote databases may share all or some of the data in the consolidated database.

See also:
- “synchronization” on page 896
- “consolidated database” on page 874
remote DBA authority

In SQL Remote, a level of permission required by the Message Agent. In MobiLink, a level of permission required by the SQL Anywhere synchronization client (dbmlsync). When the Message Agent or synchronization client connects as a user who has this authority, it has full DBA access. The user ID has no additional permissions when not connected through the Message Agent or synchronization client.

See also: “DBA authority” on page 877.

remote ID

A unique identifier in SQL Anywhere and UltraLite databases that is used by MobiLink. The remote ID is initially set to NULL and is set to a GUID during a database's first synchronization.

replication

The sharing of data among physically distinct databases. Sybase has three replication technologies: MobiLink, SQL Remote, and Replication Server.

Replication Agent

See: “LTM” on page 884.

replication frequency

In SQL Remote replication, a setting for each remote user that determines how often the publisher's message agent should send replication messages to that remote user.

See also: “replication” on page 892.

replication message

In SQL Remote or Replication Server, a communication sent between a publishing database and a subscribing database. Messages contain data, passthrough statements, and information required by the replication system.

See also:
- “replication” on page 892
- “publication update” on page 890

Replication Server

A Sybase connection-based replication technology that works with SQL Anywhere and Adaptive Server Enterprise. It is intended for near-real time replication between a small number of databases.

See also: “LTM” on page 884.

role

In conceptual database modeling, a verb or phrase that describes a relationship from one point of view. You can describe each relationship with two roles. Examples of roles are "contains" and "is a member of."
role name

The name of a foreign key. This is called a role name because it names the relationship between the foreign table and primary table. By default, the role name is the table name, unless another foreign key is already using that name, in which case the default role name is the table name followed by a three-digit unique number. You can also create the role name yourself.

See also: “foreign key” on page 879.

rollback log

A record of the changes made during each uncommitted transaction. In the event of a ROLLBACK request or a system failure, uncommitted transactions are reversed out of the database, returning the database to its former state. Each transaction has a separate rollback log, which is deleted when the transaction is complete.

See also: “transaction” on page 896.

row-level trigger

A trigger that executes once for each row that is changed.

See also:
  - “trigger” on page 897
  - “statement-level trigger” on page 895

schema

The structure of a database, including tables, columns, and indexes, and the relationships between them.

script

In MobiLink, code written to handle MobiLink events. Scripts programmatically control data exchange to meet business needs.

See also: “event model” on page 879.

script-based upload

In MobiLink, a way to customize the upload process as an alternative to using the log file.

script version

In MobiLink, a set of synchronization scripts that are applied together to create a synchronization.

secured feature

A feature specified by the -sf option when a database server is started, so it is not available for any database running on that database server.
server-initiated synchronization

A way to initiate MobiLink synchronization programmatically from the consolidated database.

server management request

A QAnywhere message that is formatted as XML and sent to the QAnywhere system queue as a way to administer the server message store or monitor QAnywhere applications.

server message store

In QAnywhere, a relational database on the server that temporarily stores messages until they are transmitted to a client message store or JMS system. Messages are exchanged between clients via the server message store.

service

In Windows operating systems, a way of running applications when the user ID running the application is not logged on.

session-based synchronization

A type of synchronization where synchronization results in consistent data representation across both the consolidated and remote databases. MobiLink is session-based.

snapshot isolation

A type of isolation level that returns a committed version of the data for transactions that issue read requests. SQL Anywhere provides three snapshot isolation levels: snapshot, statement-snapshot, and readonly-statement-snapshot. When using snapshot isolation, read operations do not block write operations.

See also: “isolation level” on page 882.

SQL

The language used to communicate with relational databases. ANSI has defined standards for SQL, the latest of which is SQL-2003. SQL stands, unofficially, for Structured Query Language.

SQL Anywhere

The relational database server component of SQL Anywhere that is intended for use in mobile and embedded environments or as a server for small and medium-sized businesses. SQL Anywhere is also the name of the package that contains the SQL Anywhere RDBMS, the UltraLite RDBMS, MobiLink synchronization software, and other components.

SQL-based synchronization

In MobiLink, a way to synchronize table data to MobiLink-supported consolidated databases using MobiLink events. For SQL-based synchronization, you can use SQL directly or you can return SQL using the MobiLink server APIs for Java and .NET.
SQL Remote

A message-based data replication technology for two-way replication between consolidated and remote databases. The consolidated and remote databases must be SQL Anywhere.

SQL statement

A string containing SQL keywords designed for passing instructions to a DBMS.

See also:
- “schema” on page 893
- “SQL” on page 894
- “database management system (DBMS)” on page 876

statement-level trigger

A trigger that executes after the entire triggering statement is completed.

See also:
- “trigger” on page 897
- “row-level trigger” on page 893

stored procedure

A program comprised of a sequence of SQL instructions, stored in the database and used to perform a particular task.

string literal

A string literal is a sequence of characters enclosed in single quotes.

subquery

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

There are two types of subquery: correlated and nested.

subscription

In MobiLink synchronization, a link in a client database between a publication and a MobiLink user, allowing the data described by the publication to be synchronized.

In SQL Remote replication, a link between a publication and a remote user, allowing the user to exchange updates on that publication with the consolidated database.

See also:
- “publication” on page 889
- “MobiLink user” on page 886
**Sybase Central**

A database management tool that provides SQL Anywhere database settings, properties, and utilities in a graphical user interface. Sybase Central can also be used for managing other Sybase products, including MobiLink.

**synchronization**

The process of replicating data between databases using MobiLink technology.

In SQL Remote, synchronization is used exclusively to denote the process of initializing a remote database with an initial set of data.

See also:
- “MobiLink” on page 886
- “SQL Remote” on page 895

**SYS**

A special user that owns most of the system objects. You cannot log in as SYS.

**system object**

Database objects owned by SYS or dbo.

**system table**

A table, owned by SYS or dbo, that holds metadata. System tables, also known as data dictionary tables, are created and maintained by the database server.

**system view**

A type of view, included in every database, that presents the information held in the system tables in an easily understood format.

**temporary table**

A table that is created for the temporary storage of data. There are two types: global and local.

See also:
- “local temporary table” on page 884
- “global temporary table” on page 881

**transaction**

A sequence of SQL statements that comprise a logical unit of work. A transaction is processed in its entirety or not at all. SQL Anywhere supports transaction processing, with locking features built in to allow concurrent transactions to access the database without corrupting the data. Transactions end either with a COMMIT statement, which makes the changes to the data permanent, or a ROLLBACK statement, which undoes all the changes made during the transaction.
**transaction log**

A file storing all changes made to a database, in the order in which they are made. It improves performance and allows data recovery in the event the database file is damaged.

**transaction log mirror**

An optional identical copy of the transaction log file, maintained simultaneously. Every time a database change is written to the transaction log file, it is also written to the transaction log mirror file.

A mirror file should be kept on a separate device from the transaction log, so that if either device fails, the other copy of the log keeps the data safe for recovery.

See also: “transaction log” on page 897.

**transactional integrity**

In MobiLink, the guaranteed maintenance of transactions across the synchronization system. Either a complete transaction is synchronized, or no part of the transaction is synchronized.

**transmission rule**

In QAnywhere, logic that determines when message transmission is to occur, which messages to transmit, and when messages should be deleted.

**trigger**

A special form of stored procedure that is executed automatically when a user runs a query that modifies the data.

See also:

- “row-level trigger” on page 893
- “statement-level trigger” on page 895
- “integrity” on page 882

**UltraLite**

A database optimized for small, mobile, and embedded devices. Intended platforms include cell phones, pagers, and personal organizers.

**UltraLite runtime**

An in-process relational database management system that includes a built-in MobiLink synchronization client. The UltraLite runtime is included in the libraries used by each of the UltraLite programming interfaces, as well as in the UltraLite engine.

**unique constraint**

A restriction on a column or set of columns requiring that all non-null values are different. A table can have multiple unique constraints.
unload

Unloading a database exports the structure and/or data of the database to text files (SQL command files for the structure, and ASCII comma-separated files for the data). You unload a database with the Unload utility. In addition, you can unload selected portions of your data using the UNLOAD statement.

upload

The stage in synchronization where data is transferred from a remote database to a consolidated database.

user-defined data type

See “domain” on page 878.

validate

To test for particular types of file corruption of a database, table, or index.

view

A SELECT statement that is stored in the database as an object. It allows users to see a subset of rows or columns from one or more tables. Each time a user uses a view of a particular table, or combination of tables, it is recomputed from the information stored in those tables. Views are useful for security purposes, and to tailor the appearance of database information to make data access straightforward.

window

The group of rows over which an analytic function is performed. A window may contain one, many, or all rows of data that has been partitioned according to the grouping specifications provided in the window definition. The window moves to include the number or range of rows needed to perform the calculations for the current row in the input. The main benefit of the window construct is that it allows additional opportunities for grouping and analysis of results, without having to perform additional queries.

Windows

The Microsoft Windows family of operating systems, such as Windows Vista, Windows XP, and Windows 200x.

Windows CE

See “Windows Mobile” on page 898.

Windows Mobile

A family of operating systems produced by Microsoft for mobile devices.
work table

An internal storage area for interim results during query optimization.
Index

Symbols
* (asterisk)
  SELECT statement, 295
*=
  Transact-SQL outer joins, 390
-im option
  performance improvement tips, 252
<
  comparison operator, 306
=* 
  Transact-SQL outer joins, 390
>
  comparison operator, 306
@@identity global variable
  IDENTITY column, 646

A
abbreviations used in execution plans
  about, 589
Access (see Microsoft Access)
access plans
  about, 541
  explanation of statistics, 595
accessing data on client computers
  about, 733
accessing remote data
  about, 753
  basic concepts, 754
  PowerBuilder DataWindows, 756
accessing tables
  table access algorithms, 568
acquire adequate hardware
  performance improvement tips, 239
actions
  CASCADE, 113
  RESTRICT, 112
  SET DEFAULT, 112
  SET NULL, 112
Adaptive Server Enterprise
  architecture, 636
  compatibility, 633
  compatibility in data import/export, 752
  data type conversions, 787
emulating, 641
  migrating to SQL Anywhere, 746
  server class, 786
Adaptive Server Enterprise compatibility
  about, 752
adding
  data to databases, 704
adding and removing statistics
  about, 225
adding data
  about, 513
  BLOBs, 516
  column data INSERT statement, 514
  constraints, 514
  defaults, 514
  into all columns, 513
  using INSERT, 513
  with SELECT, 515
adding new rows with SELECT
  about, 515
adding statistics
  Performance Monitor, 225
adding, changing, and deleting data
  about, 509
administrator role
  ASE, 638
adsodbc server class
  about, 789
advanced application profiling using diagnostic tracing
  about, 199
Advantage Database Server
  ODBC server class, 789
agent IDs
  glossary definition, 871
aggregate functions
  about, 350
  ALL keyword, 350
  applying to grouped data, 320
  data types, 352
  DISTINCT keyword, 353
  equivalent formulas for OLAP, 483
  GROUP BY clause, 358
  introduction, 319
  multiple levels, 423
  NULL, 353
  OLAP, 457
  order by and group by, 364
  outer references, 351
scalar aggregates, 351
vector aggregates, 355
windows (OLAP), 456
aggregates
item in execution plans, 599
algorithms
(see also query execution algorithms)
query execution, 565
aliases
about, 298
correlation names, 303
for calculated columns, 299
ALL
keyword and aggregate functions, 350
keyword and UNION clause, 365
subquery tests, 499
ALL operator
about, 499
notes, 499
subquery tests, 499
All-rows
choosing the optimizer goal, 240
All-rows optimization goal
performance, 260
allow_nulls_by_default option
setting for Transact-SQL compatibility, 642
allow_snapshot_isolation option
using, 128
alphabetical order
ORDER BY clause, 316
ALTER INDEX statement
unavailable with snapshot isolation, 126
ALTER SERVER statement
altering remote servers, 759
ALTER statement
automatic commit, 121
ALTER TABLE statement
CHECK constraints, 103
concurrency, 179
examples, 41
foreign keys, 47
primary keys, 44
unavailable with snapshot isolation, 126
ALTER TEXT CONFIGURATION statement
altering, 344
altering
CHECK constraints, 106
columns using SQL, 41
columns using Sybase Central, 40
computed column expressions, 49
procedures using Sybase Central, 809
regular views using SQL, 61
regular views using Sybase Central, 61
regular views with dependencies, 60
tables that have dependent materialized views, 55
tables using SQL, 41
tables using Sybase Central, 40
tables with view dependencies, 39
text indexes, 337
triggers using SQL, 823
triggers using Sybase Central, 822
altering and dropping CHECK constraints
about, 106
altering computed column expressions
about, 49
altering procedures
about, 809
altering regular views
about, 60
altering remote servers
about, 759
altering tables
about, 39
altering triggers
about, 822
always use a transaction log
performance improvement tips, 239
analyzing
procedure profiling results, 192
analyzing diagnostic tracing information
about, 212
analyzing procedure profiling results
about, 192
AND
using logical operators, 313
annotation phase
query processing, 526
ANSI
non-ANSI joins, 380
SQL standards and inconsistencies, 131
ANSI compliance
(see also SQL standards)
ANSI update constraints
exection plans, 596
ANY operator
about, 497
problems, 498
subquery tests, 497
apostrophes
character strings, 310
application profiling
about, 188
creating a tracing session, 210
detecting whether CPU is a limiting factor, 213
detecting whether I/O bandwidth is a limiting factor, 213
detecting whether memory is a limiting factor, 213
Index consultant, 194
procedure profiling, 190
production database, 199
request trace analysis, 214
tracing database, 199
tutorials, 263
application profiling mode
using, 188
application profiling wizard
about, 189
enabling and disabling automatic start, 189
starting, 189
tutorials, 263
apply
CROSS APPLY and OUTER APPLY joins, 399
apply expressions
about, 399
examples, 399
architectures
Adaptive Server, 636
arithmetic
expressions and operator precedence, 300
operations, 350
articles
glossary definition, 871
AS keyword
aliases, 298
ascending order
ORDER BY clause, 362
asejdbc server class
about, 800
aseodbc server class
about, 786
assigning
data types to columns, 107
domains to columns, 107
assumptions affecting optimization about, 544
asterisk (*)
SELECT statement, 295
AT clause
CREATE EXISTING TABLE statement, 767
atomic compound statements
about, 831
atomic transactions
about, 120
glossary definition, 871
attributes
choosing, 14
definition of, 5
obtaining query results as XML, 672
SQLCA.lock, 135
AUTO mode
using, 677
auto_commit option
grouping changes in Interactive SQL, 121
autocommit
performance, 250
transactions, 121
AUTOINCREMENT
default, 99
IDENTITY column, 645
negative numbers, 99
signed data types, 99
UltraLite applications, 99
when to use, 178
automatic commit
ALTER statement, 121
COMMENT statement, 121
data definition statements, 121
DROP statement, 121
automatic joins
foreign keys, 631
automatic performance tuning
about, 544
automatic translation of stored procedures
about, 653
automatic_timestamp option
setting for Transact-SQL compatibility, 642
automation
generating unique keys, 178
Avail IO statistic
description, 237
AVG function
equivalent mathematical formula, 483
usage, 457
AvgDiskReads
  estimate in access plans, 596
AvgDiskReadTime
  estimate in access plans, 596
AvgDiskWrites
  estimate in access plans, 596
AvgRowCount
  estimate in access plans, 596
AvgRunTime
  estimate in access plans, 596

B
B-link indexes
  about, 624
B-trees indexes
  about, 624
base tables
  creating, 38
  glossary definition, 871
  quick comparison with regular and materialized views, 54
baselining
  using procedure profiling, 284
baselining using a profiling log file
  about, 284
basic aggregate functions
  OLAP, 457
basic concepts to access remote data
  overview, 754
batch mode
  Interactive SQL, 750
batch operations
  Interactive SQL, 750
batches
  about, 827
  compared to stored procedures, 827
  compound statements, 828
  control statements, 827
  OUTPUT statement, 829
  SQL statements allowed, 855
  statements allowed, 855
  Transact-SQL overview, 652
  using SELECT statements, 855
  writing, 827
BCP format
  import/export with ASE, 752
BEGIN TRANSACTION statement
  remote data access, 777
  restrictions on transaction management, 777
benefits
  Index Consultant recommendations, 196
benefits of procedures and triggers
  about, 807
BETWEEN keyword
  range queries, 307
binary files
  importing, 721
binary large objects
  inserting, 516
bit arrays
  glossary definition, 871
bitmaps
  scanning, 616
bits
  item in execution plans, 599
BLOBs
  about, 25
  inserting, 516
  sharing, 25
  storing in the database, 25
blocking
  about, 138
  deadlock, 138
  example, 167
  transactions, 138
  troubleshooting, 139
blocking option
  using, 138
bloom filters (see Hash filters)
  hash filter algorithms, 586
book materialized views (see book views)
boolean searching
  full text search, 328
break conditions
  setting, 863
breakpoints
  about, 863
  conditions, 864
  counts, 864
  disabling, 864
  enabling, 864
  individual connections, 864
  individual users, 864
  setting, 863
status, 864
browsing
  regular views, 64
table data, 43
browsing data in views
  about, 64
browsing databases
  isolation levels, 157
browsing the data held in tables
  about, 43
buckets
  histograms, 542
bugs
  providing feedback, xv
build values
  item in execution plans, 599
bulk loading
  performance, 702
bulk operations
  about, 702
  issues for recovering data, 703
  performance impacts, 702
  performance improvement tips, 252
business rules
  glossary definition, 872
bypassing optimization
  simple queries, 527

C

  cache
    dynamic sizing, 256
    encrypted databases require larger cache, 241
    execution plans, 562
    increasing cache size to improve performance, 241
    initial, min, and max size, 255
    monitoring size, 258
    read-hit ratio, 604
    statement level caching, 562
    statements in stored procedures, 562
    statements that bypass query optimization, 562
    Unix, 257
    use the cache to improve performance, 254
    warming, 258
Cache Hits/sec statistic
  description, 227
Cache Pages Allocated Structures statistic
  description, 233

Copyright © 2008, iAnywhere Solutions, Inc. - SQL Anywhere 11.0.0
about, 258
cached plans
  optimizer bypass, 527
CacheHits
  Node Statistics field descriptions, 611
  statistic in access plans, 595
CacheRead
  Node Statistics field descriptions, 611
  statistic in access plans, 595
CacheReadIndLeaf
  statistic in access plans, 595
CacheReadTable
  statistic in access plans, 595
caching
  execution plans, 562
  statements that bypass query optimization, 562
  subqueries, 583
  user-defined functions, 584
call stack
debugger, 866
CALL statement
  about, 806
  control statements, 830
  examples, 810
  parameters, 834
calling procedures
  about, 810
calling user-defined functions
  about, 815
canceling changes
  about, 511
canceling requests
  remote data access, 782
candidate indexes
  about, 196
  Index Consultant, 196
cardinality
  item in execution plans, 598
  relationships and, 6
carriers
glossary definition, 872
Cartesian products
  about, 385
CASCADE action
  about, 113
case sensitivity
  creating ASE-compatible databases, 642
data, 643
databases, 643
domains, 644
identifiers, 644
passwords, 644
remote access, 781
sort order, 363
SQL, 293
table names, 293
Transact-SQL compatibility, 643
user IDs, 644
CASE statement
  control statements, 830
catalog
  ASE compatibility, 637
  finding dependency information, 57
  index information, 92
categorization of subqueries
  about, 486
cdata directive
  using, 688
changing data
  permissions, 510
  UPDATE statement, 517
  updating data using more than one table, 518
changing data using INSERT
  about, 519
changing data using UPDATE
  about, 517
changing diagnostic tracing configuration settings
  about, 209
changing diagnostic tracing settings while a tracing
  session is in progress
  about, 210
changing isolation levels within transactions
  about, 136
changing many-to-many relationships into entities
  about, 9
changing the isolation level
  about, 134
character data
  searching for, 310
character set conversion
  remote date access, 755
character sets
  glossary definition, 872
character strings
  quotes, 310
  select list using, 299
usage, 310
character strings and quotation marks
about, 310
character strings in query results
about, 298
CHECK conditions
  Transact-SQL, 637
CHECK constraints
  altering, 106
  choosing, 27
  columns, 103
  domains, 104
  dropping, 106
  glossary definition, 872
  tables, 104
  using in domains, 108
check for concurrency issues
  performance improvement tips, 239
check if you need to change the command delimiter
tips for writing procedures, 853
checking referential integrity at commit
about, 153
Checkpoint Flushed/sec statistic
description, 228
checkpoint log
  performance, 243
checkpoint log statistic
description, 228
checkpoint statistics
  list, 228
Checkpoint Urgency statistic
description, 228
checkpoints
  glossary definition, 872
Checkpoint/sec statistic
description, 228
checksums
  glossary definition, 872
ChkptLog Bitmap size statistic
description, 228
ChkptLog Commit to disk/sec statistic
description, 228
ChkptLog Log size statistic
description, 228
ChkptLog Page images saved/sec statistic
description, 228
ChkptLog Pages in use statistic
description, 228
ChkptLog Relocate pages statistic
  description, 228
ChkptLog Save preimage/sec statistic
  description, 228
ChkptLog Write pages/sec statistic
  description, 228
ChkptLog Writes to bitmap/sec statistic
  description, 228
ChkptLog Writes/sec statistic
  description, 228
choose the optimizer goal
  performance improvement tips, 240
choosing a diagnostic tracing level
about, 201
choosing a set of indexes
about, 86
choosing column names
about, 24
choosing constraints
about, 27
choosing data types for columns
about, 24
choosing isolation levels
about, 156
circumstances under which parallelism is used
  parallelism, 567
classes
  remote servers, 783
clauses
  about, 292
  COMPUTE, 649
  FOR BROWSE, 649
  FOR READ ONLY, 649
  FOR UPDATE, 649
  GROUP BY ALL, 649
  INTO, 837
  ON EXCEPTION RESUME, 657, 849
client files
  importing from, and exporting to, client computers, 733
client message store IDs
  glossary definition, 873
client message stores
  glossary definition, 872
client-side data
  preventing loss of data loaded from a client, 734
client-side loading
about, 702
client/server
glossary definition, 873
CLOSE statement
cursor management procedures, 841
clustered indexes
  implementing Index Consultant recommendations, 197
  Index Consultant recommendations, 196
  using, 87
ClusteredHashGroupBy algorithm (GrByHClust)
  about, 579
ClusteredHashGroupBy plan item
  abbreviations in the plan, 589
code pages
glossary definition, 873
collations
  glossary definition, 873
collect statistics on small tables
  performance improvement tips, 241
colons separate join strategies
  about, 600
column attributes
  AUTOINCREMENT, 178
  generating default values, 178
  NEWID, 178
column CHECK constraints from domains
  inheriting, 104
column compression
  about, 26
column constraints
  UNIQUE, 105
column defaults
  modifying and dropping, 98
column order
  results reflect order in select list, 297
column statistics (see histograms)
  about, 542
  updating, 543
columns
  allowing NULL values, 24
  altering using SQL, 41
  altering using Sybase Central, 40
  assigning data types and domains, 107
  calculated, 299
  CHECK constraints, 106
  compression, 26
  constraints, 27
  copying, 51
data types, 24
defaults, 97
GROUP BY clause, 355
IDENTITY, 645
managing column constraints, 105
naming, 24
properties, 24
select list, 296
SELECT statements, 297
timestamp, 644
Comm Bytes Received /sec statistic
description, 230
Comm Bytes Received Uncompressed/sec statistic
description, 230
Comm Bytes Sent Uncompressed/sec statistic
description, 230
Comm Bytes Sent/sec statistic
description, 230
Comm Free Buffers statistic
description, 230
Comm Multi-packets Received/sec statistic
description, 230
Comm Multi-packets Sent/sec statistic
description, 230
Comm Packets Received Uncompressed/sec statistic
description, 230
Comm Packets Received/sec statistic
description, 230
Comm Packets Sent Uncompressed/sec statistic
description, 230
Comm Packets Sent/sec statistic
description, 230
Comm Requests Received statistic
description, 230
Comm Send Fails/sec statistic
description, 230
Comm TotalBuffers statistic
description, 230
Comm Unique Client Addresses statistic
description, 230
command delimiter
  setting, 853
command files
  about, 750
building, 750
creating, 750
executing, 750
glossary definition, 873
opening in Interactive SQL, 750
overview, 750
running, 750
SQL Statements pane, 750
writing output, 751
commands
loading in Interactive SQL, 751
commas
star joins, 394
table expression lists, 385
when joining table expressions, 409
COMMENT statement
automatic commit, 121
comments
altering procedures using Sybase Central, 809
COMMIT statement
about, 510
compound statements, 831
procedures and triggers, 852
remote data access, 777
transactions, 121
verify referential integrity, 153
COMMIT TRANSACTION statement
restrictions on transaction management, 777
commits
wait_for_commit option, 153
common estimates used in the plan
about, 596
common statistics used in the plan
about, 595
common table expressions
about, 419
common applications, 423
data types in recursive, 431
exploring hierarchical data structures, 426
least distance problems, 432
multiple aggregation levels, 423
parts explosion problems, 428
restrictions on recursive, 427
storing constant sets, 424
where permitted, 422
communication streams
glossary definition, 873
communications statistics
list, 230
comparison operators
NULL values, 312
subqueries, 503
symbols, 306
comparison test
subqueries, 495
comparisons
introduction, 314
NULL values, 312
sort order, 306
trailing blanks, 306
compatibility
ASE, 633
case sensitivity, 643
GROUP BY clause, 371
import/export with ASE, 752
non-ANSI joins, 380
outputting NULLs, 730
servers and databases, 636
setting options for Transact-SQL compatibility, 642
compatibility of joins
about, 650
competing triggers
execution order, 824
complete passthrough of the statement
remote data access, 779
completing transactions
about, 121
complex outer joins
about, 388
compliance with SQL standards
(see also SQL standards)
composite indexes
about, 621
effect of column order, 621
ORDER BY clause, 622
compound statements
atomic, 831
declarations, 831
using, 831
compressed B-trees
indexes, 624
compressed columns
about, 26
compression
columns, 26
COMPUTE clause
CREATE TABLE, 48
Transact-SQL SELECT statement syntax
unsupported, 649
computed columns
  about, 48
  indexes, 89
  inserting and updating, 49
  limitations, 50
  making queries using sargable functions, 549
  recalculation, 50
  triggers, 49
computing values in the SELECT list
  about, 299
concatenating strings
  NULL, 313
conceptual data modeling
  about, 3
conceptual database models
  definition of, 5
concurrency
  about, 122
  benefits, 122
  consistency, 131
  DDL statements, 179
  glossary definition, 873
  how locking works, 142
  improving, 158
  improving using indexes, 159
  inconsistencies, 131
  ISO SQL standard, 131
  performance, 123
  primary keys, 178
concurrent transactions
  blocking, 138
  blocking example, 167
conditions
  connecting with logical operators, 313
  GROUP BY clause, 321
  pattern matching, 308
configuration notes for JDBC classes
  about, 799
configuring
  diagnostic tracing, 201
  diagnostic tracing settings, 209
configuring databases for Transact-SQL compatibility
  about, 641
configuring diagnostic tracing
  about, 201
configuring UNIQUE constraints
  about, 105
conflict resolution
  glossary definition, 874
conflicts
  cyclical blocking, 139
  locking, 149
  snapshot isolation, 131
  table locks, 146
  transaction blocking, 138
  transaction blocking example, 167
conformance with SQL standards
  (see also SQL standards)
connecting conditions with logical operators
  about, 313
  Connection Count statistic
    description, 237
  connection IDs
    glossary definition, 874
  connection locks
    duration, 142
  connection options
    impact on materialized views, 71
  connection profiles
    glossary definition, 874
  connection-initiated synchronization
    glossary definition, 874
  CONNECTION_PROPERTY function
    about, 223
connections
  debugger, 868
  debugging, 859
  loopback, 773
  materialized view candidacy, 551
  remote, 777
connectivity problems
  remote data access, 781
consistency
  about, 120
  assuring using locks, 142
  correctness and scheduling, 157
  dirty reads, 132
  dirty reads and locking, 150
  dirty reads tutorial, 160
  during transactions, 131
  effects of unserializable schedules, 157
  example of non-repeatable read, 165
  ISO SQL standard, 131
  isolation level 0, 150
  isolation levels, 124
  phantom rows, 132
<table>
<thead>
<tr>
<th>Using</th>
<th>Oracle data type conversions, 792</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE EXISTING TABLE statement</td>
<td>remote servers, 799</td>
</tr>
<tr>
<td>creating proxy tables for directory access servers, 762</td>
<td>create table check constraint wizard</td>
</tr>
<tr>
<td>specifying proxy table location, 767</td>
<td>accessing, 105</td>
</tr>
<tr>
<td>using, 768</td>
<td>CREATE TABLE statement</td>
</tr>
<tr>
<td>create external login wizard</td>
<td>about, 38, 53</td>
</tr>
<tr>
<td>using, 765</td>
<td>concurrency, 179</td>
</tr>
<tr>
<td>CREATE EXTERNLOGIN statement</td>
<td>creating proxy tables for directory access servers, 762</td>
</tr>
<tr>
<td>creating external logins for directory access servers, 762</td>
<td>creating Transact-SQL-compatible tables, 647</td>
</tr>
<tr>
<td>using, 765</td>
<td>foreign keys, 47</td>
</tr>
<tr>
<td>create foreign key wizard</td>
<td>primary keys, 44</td>
</tr>
<tr>
<td>using, 46</td>
<td>proxy tables, 769</td>
</tr>
<tr>
<td>CREATE FUNCTION statement</td>
<td>specifying proxy table location, 767</td>
</tr>
<tr>
<td>about, 814</td>
<td>create table wizard</td>
</tr>
<tr>
<td>create function wizard</td>
<td>accessing, 38</td>
</tr>
<tr>
<td>accessing, 814</td>
<td>create text configuration object wizard</td>
</tr>
<tr>
<td>create global temporary table wizard</td>
<td>using, 341</td>
</tr>
<tr>
<td>accessing, 53</td>
<td>CREATE TEXT CONFIGURATION statement</td>
</tr>
<tr>
<td>CREATE INDEX statement</td>
<td>using, 341</td>
</tr>
<tr>
<td>concurrency, 179</td>
<td>CREATE TEXT INDEX statement</td>
</tr>
<tr>
<td>unavailable with snapshot isolation, 126</td>
<td>using, 335</td>
</tr>
<tr>
<td>create index wizard</td>
<td>create text index wizard</td>
</tr>
<tr>
<td>using, 89</td>
<td>using, 335</td>
</tr>
<tr>
<td>create materialized view wizard</td>
<td>CREATE TRIGGER statement</td>
</tr>
<tr>
<td>accessing, 73</td>
<td>about, 819</td>
</tr>
<tr>
<td>CREATE PROCEDURE statement</td>
<td>create trigger wizard</td>
</tr>
<tr>
<td>examples, 808</td>
<td>using, 820</td>
</tr>
<tr>
<td>parameters, 833</td>
<td>create unique constraint wizard</td>
</tr>
<tr>
<td>using, 775</td>
<td>accessing, 105</td>
</tr>
<tr>
<td>create procedure wizard</td>
<td>CREATE VIEW statement</td>
</tr>
<tr>
<td>using, 808</td>
<td>WITH CHECK OPTION clause, 58</td>
</tr>
<tr>
<td>create proxy table wizard</td>
<td>create view wizard</td>
</tr>
<tr>
<td>using, 768</td>
<td>using, 60</td>
</tr>
<tr>
<td>create remote procedure wizard</td>
<td>creating</td>
</tr>
<tr>
<td>using, 775</td>
<td>column defaults, 97</td>
</tr>
<tr>
<td>create remote server wizard</td>
<td>data types using SQL, 108</td>
</tr>
<tr>
<td>using, 758</td>
<td>data types using Sybase Central, 107</td>
</tr>
<tr>
<td>CREATE RULE statement</td>
<td>databases tutorial, 28</td>
</tr>
<tr>
<td>unsupported, 637</td>
<td>diagnostic tracing session, 210</td>
</tr>
<tr>
<td>CREATE SERVER statement</td>
<td>domains using SQL, 108</td>
</tr>
<tr>
<td>creating directory access servers, 762</td>
<td>domains using Sybase Central, 107</td>
</tr>
<tr>
<td>creating remote servers, 757</td>
<td>external tracing database, 215</td>
</tr>
<tr>
<td>DB2 data type conversions, 790</td>
<td>indexes, 89</td>
</tr>
<tr>
<td>JDBC and Adaptive Server Enterprise, 800</td>
<td>materialized views, 73</td>
</tr>
<tr>
<td>Microsoft SQL Server data type conversions, 793</td>
<td>procedures, 808</td>
</tr>
<tr>
<td>ODBC and ASE data type conversions, 787</td>
<td>regular views, 60</td>
</tr>
</tbody>
</table>
| tables, 38 | **Table**
temporary procedures, 809
temporary tables, 53
text indexes, 333
triggers, 819
user-defined functions, 814
creating a diagnostic tracing session
about, 210
creating a proxy table with the CREATE TABLE
statement
about, 769
creating a Transact-SQL-compatible database
about, 641
creating an external tracing database
about, 215
creating compatible tables
about, 647
creating databases
Transact-SQL-compatible database, 641
creating directory access servers
about, 762
creating external logins
about, 765
creating indexes
about, 89
creating materialized views
about, 73
creating proxy tables in SQL
about, 768
creating proxy tables in Sybase Central
about, 768
creating regular views
about, 60
creating remote procedures
about, 775
creating remote servers
about, 757
creating tables
about, 38
creating temporary tables
about, 53
creator ID
glossary definition, 875
CROSS APPLY clause
about, 399
example, 399
cross joins
about, 385
cross products
about, 385
CUBE clause
about, 445
using as a shortcut to GROUPING SETS, 443
CUME_DIST function
determining the median of a set of values, 480
equivalent mathematical formula, 483
usage, 479
Current Active statistic
description, 233
current date and time defaults
about, 98
CurrentCacheSize
Optimizer Statistics field descriptions, 612
cursor instability
about, 133
cursor management
overview, 841
cursor positions
glossary definition, 875
cursor result sets
glossary definition, 875
cursor stability
about, 133
cursors
glossary definition, 875
instability, 133
LOOP statement, 841
procedures, 841
procedures and triggers, 841
SELECT statements, 841
stability, 133
updating in joins, 380
Cursors Open statistic
description, 236
Cursors statistic
description, 236
customizing
graphical plan appearance, 609
customizing graphical plans
about, 602
cyclical blocking conflict
about, 139
D
data
adding, changing, and deleting, 509
<table>
<thead>
<tr>
<th>Term</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>case sensitivity</td>
<td>643</td>
</tr>
<tr>
<td>consistency</td>
<td>131</td>
</tr>
<tr>
<td>editing in Interactive SQL</td>
<td>43</td>
</tr>
<tr>
<td>editing in Sybase Central</td>
<td>43</td>
</tr>
<tr>
<td>export tools</td>
<td>722</td>
</tr>
<tr>
<td>exporting</td>
<td>722</td>
</tr>
<tr>
<td>exporting as XML</td>
<td>664</td>
</tr>
<tr>
<td>importing</td>
<td>704</td>
</tr>
<tr>
<td>importing and exporting</td>
<td>702</td>
</tr>
<tr>
<td>integrity and correctness</td>
<td>157</td>
</tr>
<tr>
<td>invalid</td>
<td>94</td>
</tr>
<tr>
<td>permissions required to modify data</td>
<td>510</td>
</tr>
<tr>
<td>populating materialized views</td>
<td>74</td>
</tr>
<tr>
<td>refreshing manual views</td>
<td>75</td>
</tr>
<tr>
<td>searching</td>
<td>86</td>
</tr>
<tr>
<td>viewing</td>
<td>43</td>
</tr>
<tr>
<td>data consistency</td>
<td></td>
</tr>
<tr>
<td>assuring using locks</td>
<td>142</td>
</tr>
<tr>
<td>correctness</td>
<td>157</td>
</tr>
<tr>
<td>dirty reads</td>
<td>132</td>
</tr>
<tr>
<td>dirty reads and locking</td>
<td>150</td>
</tr>
<tr>
<td>dirty reads tutorial</td>
<td>160</td>
</tr>
<tr>
<td>ISO SQL standard</td>
<td>131</td>
</tr>
<tr>
<td>isolation level 0</td>
<td>150</td>
</tr>
<tr>
<td>phantom rows</td>
<td>132</td>
</tr>
<tr>
<td>phantom rows and locking</td>
<td>150</td>
</tr>
<tr>
<td>phantom rows tutorial</td>
<td>169</td>
</tr>
<tr>
<td>practical locking implications</td>
<td>174</td>
</tr>
<tr>
<td>repeatable reads</td>
<td>132</td>
</tr>
<tr>
<td>repeatable reads and locking</td>
<td>150</td>
</tr>
<tr>
<td>repeatable reads tutorial</td>
<td>164</td>
</tr>
<tr>
<td>snapshot isolation</td>
<td>151</td>
</tr>
<tr>
<td>data cube</td>
<td></td>
</tr>
<tr>
<td>glossary definition</td>
<td>875</td>
</tr>
<tr>
<td>data definition language (see DDL)</td>
<td></td>
</tr>
<tr>
<td>about</td>
<td>36</td>
</tr>
<tr>
<td>glossary definition</td>
<td>875</td>
</tr>
<tr>
<td>data definition statements (see DDL)</td>
<td></td>
</tr>
<tr>
<td>data entry</td>
<td></td>
</tr>
<tr>
<td>isolation levels</td>
<td>157</td>
</tr>
<tr>
<td>data integrity</td>
<td></td>
</tr>
<tr>
<td>about</td>
<td>94</td>
</tr>
<tr>
<td>checking</td>
<td>113</td>
</tr>
<tr>
<td>column constraints</td>
<td>27</td>
</tr>
<tr>
<td>column defaults</td>
<td>97</td>
</tr>
<tr>
<td>constraints</td>
<td>96, 103</td>
</tr>
<tr>
<td>effects of unserializable schedules on</td>
<td>157</td>
</tr>
<tr>
<td>enforcing</td>
<td>110</td>
</tr>
<tr>
<td>ensuring</td>
<td>93</td>
</tr>
<tr>
<td>information in the system tables</td>
<td>117</td>
</tr>
<tr>
<td>losing</td>
<td>112</td>
</tr>
<tr>
<td>tools</td>
<td>95</td>
</tr>
<tr>
<td>data manipulation language</td>
<td></td>
</tr>
<tr>
<td>glossary definition</td>
<td>876</td>
</tr>
<tr>
<td>data model normalization</td>
<td></td>
</tr>
<tr>
<td>about</td>
<td>15</td>
</tr>
<tr>
<td>data modeling</td>
<td></td>
</tr>
<tr>
<td>about</td>
<td>3</td>
</tr>
<tr>
<td>data modification statements</td>
<td></td>
</tr>
<tr>
<td>about</td>
<td>510</td>
</tr>
<tr>
<td>data organization</td>
<td></td>
</tr>
<tr>
<td>physical</td>
<td>615</td>
</tr>
<tr>
<td>data recovery</td>
<td></td>
</tr>
<tr>
<td>importing and exporting</td>
<td>703</td>
</tr>
<tr>
<td>transactions</td>
<td>511</td>
</tr>
<tr>
<td>data recovery issues for bulk operations</td>
<td></td>
</tr>
<tr>
<td>about</td>
<td>703</td>
</tr>
<tr>
<td>data sources</td>
<td></td>
</tr>
<tr>
<td>external servers</td>
<td>785</td>
</tr>
<tr>
<td>data tab</td>
<td></td>
</tr>
<tr>
<td>restrictions on remote tables</td>
<td>755</td>
</tr>
<tr>
<td>SQL Anywhere plug-in</td>
<td>43</td>
</tr>
<tr>
<td>data type conversions</td>
<td></td>
</tr>
<tr>
<td>DB2, 790</td>
<td></td>
</tr>
<tr>
<td>Microsoft SQL Server, 793</td>
<td></td>
</tr>
<tr>
<td>ODBC and ASE, 787</td>
<td></td>
</tr>
<tr>
<td>Oracle, 792</td>
<td></td>
</tr>
<tr>
<td>data types</td>
<td></td>
</tr>
<tr>
<td>aggregate functions</td>
<td>352</td>
</tr>
<tr>
<td>assigning columns</td>
<td>107</td>
</tr>
<tr>
<td>choosing</td>
<td>24</td>
</tr>
<tr>
<td>creating using SQL</td>
<td>108</td>
</tr>
<tr>
<td>creating using Sybase Central</td>
<td>107</td>
</tr>
<tr>
<td>dropping</td>
<td>109</td>
</tr>
<tr>
<td>EXCEPT statement</td>
<td>365</td>
</tr>
<tr>
<td>glossary definition</td>
<td>876</td>
</tr>
<tr>
<td>INTERSECT statement</td>
<td>365</td>
</tr>
<tr>
<td>recursive subqueries</td>
<td>431</td>
</tr>
<tr>
<td>remote procedures</td>
<td>776</td>
</tr>
<tr>
<td>Transact-SQL timestamp</td>
<td>644</td>
</tr>
<tr>
<td>UNION statement</td>
<td>365</td>
</tr>
<tr>
<td>user-defined</td>
<td>107</td>
</tr>
<tr>
<td>database administrator</td>
<td></td>
</tr>
<tr>
<td>glossary definition</td>
<td>876</td>
</tr>
<tr>
<td>roles</td>
<td>638</td>
</tr>
<tr>
<td>database connections</td>
<td></td>
</tr>
</tbody>
</table>
glossary definition, 876
database design concepts
about, 5
database files
file fragmentation, 246
fragmentation, 245
glossary definition, 876
performance, 243
database names
glossary definition, 877
database objects
direct references, 57
editing properties, 36
glossary definition, 877
indirect references, 57
database options
impact on materialized views, 71
Index Consultant, 197
setting for Transact-SQL compatibility, 642
text configuration object settings, 345
database owner
glossary definition, 877
database pages
Index Consultant recommendations, 196
database servers
glossary definition, 877
database threads
blocked, 139
database tracing wizard
tutorials, 263
using, 210
databases
case sensitivity, 643
case sensitivity in ASE-compatible databases, 642
design concepts, 5
designing, 3
displaying system objects, 36
exporting, 730
extracting for SQL Remote, 745
glossary definition, 876
importing XML, 665
joining tables from multiple, 773
migrating to SQL Anywhere, 746
normalizing, 15
rebuilding databases not involved in synchronization, 739
reloading, 743
storing XML, 663
Transact-SQL compatibility, 641
unloading, 730
unloading and reloading, 743
unloading and reloading databases involved in synchronization, 740
unloading and reloading databases not involved in synchronization, 739
upgrading database file format, 736
verifying design, 22
viewing and editing properties, 36
working with objects, 35
DataSet
using to export relational data as XML, 664
using to import XML, 670
DataWindows
remote data access, 756
date_format option
impact on text indexes, 345
dates
entry rules, 310
procedures and triggers, 853
search conditions introduction, 314
searching for, 310
DB2
data type conversions, 790
DB2 remote data access
about, 789
db2odbc server class
about, 789
DB_PROPERTY function
about, 223
DBA authority
glossary definition, 877
dbiqsql utility
rebuilding databases, 741
DBMS
glossary definition, 876
dbo user
ASE, 638
dbspaces
glossary definition, 877
managing, 636
dbunload utility
exporting data, 726
rebuild tools, 741
using, 730
dbxtract utility
extracting data, 745
index

DDL
about, 36
automatic commit, 121
concurrency, 179
glossary definition, 875
statements disallowed in snapshot isolation transactions, 126
deadlock reporting
about, 139
Deadlock system event
using, 139
deadlocks
about, 138
Application Profiling tutorial, 265
diagnosing, 139
glossary definition, 877
reasons for, 139
reporting, 139
transaction blocking, 138
tutorial: diagnosing deadlocks, 265
debug mode
using, 858
debugger
about, 857
connecting, 859
debugging stored procedures, 860
examining variables, 866
features, 858
getting started, 859
requirements, 858
SOAP functions, 858
starting, 859
tutorial, 859
working with breakpoints, 863
working with connections, 868
debugger_tutorial procedure
about, 861
debugging
about, 857
permissions, 858
requirements, 858
SOAP procedures, 858
stored procedures, 860
tutorial, 860
using the SQL Anywhere debugger, 857
debugging application logic
about, 214
debugging logic in the database
about, 857
deciding when to use materialized views
about, 67
decision support
isolation levels, 157
declarations in compound statements
about, 831
declare constraints
performance improvement tips, 241
DECLARE statement
compound statements, 831
cursor management procedures, 841
procedures, 845
declaring parameters for procedures
about, 833
DecodePostings (DP)
about, 584
decrypting
materialized views, 78
default error handling in procedures and triggers
about, 844
default handling of warnings in procedures and triggers
about, 847
default_char
text configuration objects, 339
default_nchar
text configuration objects, 339
defaults
AUTOINCREMENT, 99
column, 97
constant expressions, 102
creating, 97
creating in Sybase Central, 98
current date and time, 98
GLOBAL AUTOINCREMENT, 100
INSERT statement and, 514
introduction, 95
NEWID, 101
NULL, 102
string and number, 102
Transact-SQL, 637
user ID, 99
using in domains, 108
with transactions and locks, 178
defining the merge behavior
about, 713
defragmenting
about, 245
all tables in a database, 246
hard disk, 246
individual tables in a database, 246
delayed_commits option
performance improvement tips, 252
delaying commits
performance improvement tips, 252
delaying referential integrity checks
about, 153
DELETE statement
errors, 114
locking during, 154
referential integrity check on DELETE, 114
using, 520
deleting
directory access servers, 763
indexes, 91
materialized views, 84
procedures, 811
remote servers, 759
tables, 42
triggers, 823
views, 62
deleting all rows from a table
about, 521
deleting data
DELETE statement, 520
TRUNCATE TABLE statement, 521
deleting directory access servers
about, 763
deleting remote servers
about, 759
delimit statements within your procedure
tips for writing procedures, 853
demo.db file
schema, 375
DENSE_RANK function
equivalent mathematical formula, 483
usage, 477
dependencies
view dependencies, 55
dependencies and schema-altering changes
about, 55
depth
item in execution plans, 598
derived tables
about, 303
DerivedTable algorithm, 584
joins, 398
key joins, 414
natural joins, 405
outer joins, 389
DerivedTable algorithm (DT)
about, 584
DerivedTable plan item
abbreviations in the plan, 589
descending order
ORDER BY clause, 362
design process
about, 11
designing
databases, 3
designing databases
about, 3
concepts, 5
procedure, 11
designing the database table properties
about, 24
detecting slow statements
tutorial: diagnosing slow statements, 270
detecting when hardware resources are a limiting factor
about, 213
determine the degree of table fragmentation
about, 246
determining current diagnostic tracing settings
about, 208
determining which logical indexes share a physical index
about, 618
deterministic functions
defined, 584
side-effects, 584
developer community
newsgroups, xv
device tracking
glossary definition, 877
devices
managing, 636
diagnostic tracing
about, 199
changing tracing settings during a tracing session, 210
configure tracing settings, 209
configuring, 201
creating a tracing session, 210
creating an external tracing database, 215
database properties related to tracing, 201
determining tracing settings, 208
interpreting information, 212
production database, 199
sa_diagnostic_tracing_level table, 201
sa_save_trace_data system procedure, 201
sa_set_tracing_level system procedure, 201
tracing conditions, 207
tracing database, 199
tracing levels, 202
tracing scopes, 203
tracing types, 204
diagnostic tracing conditions
about, 207
diagnostic tracing levels
about, 202
deciding which diagnostic tracing level to use, 201
setting, 209
diagnostic tracing scopes
about, 203
descriptions of, 203
diagnostic tracing session
creating, 210
diagnostic tracing types
about, 204
OPTIMIZATION_LOGGING, 204
OPTIMIZATION_LOGGING_WITH_PLANS, 204
difference between FALSE and UNKNOWN
about, 313
differences from other SQL dialects
overview, 627
direct references
database objects, 57
direct row handling
glossary definition, 878
directed graphs
about, 432
direction
item in execution plans, 598
directory access servers
about, 762
altering, 763
creating, 762
deleting, 763
deleting proxy tables, 763
dirty reads
inconsistencies, 131
locking during queries, 150
tutorial, 160
versus isolation levels, 132
disabled status, materialized views
about, 68
disabling
materialized views, 80
procedure profiling in Sybase Central, 191
views, 63
disabling breakpoints
about, 864
enabling, 864
disabling procedure profiling
about, 191
disabling profiling using sa_server_option
about, 220
discovery of exploitable conditions through predicate inference
about, 536
disk access cost models
about, 541
Disk Active I/Os statistic
description, 231
disk allocation for inserted rows
about, 615
disk I/O statistics
list, 231
Disk Maximum I/Os statistic
description, 231
disk read statistics
list, 232
Disk Reads Active statistic
description, 232
Disk Reads Index interior/sec statistic
description, 232
Disk Reads Index leaf/sec statistic
description, 232
Disk Reads Maximum Active statistic
description, 232
Disk Reads Table/sec statistic
description, 232
Disk Reads Total Pages/sec statistic
description, 232
Disk Reads Work Table statistics
description, 232
DISK statements
unsupported, 636
disk write statistics
list, 232
Disk Writes Active statistic
description, 232
Disk Writes Commit Files/sec statistic
description, 232
Disk Writes Database Extends/sec statistic
description, 232
Disk Writes Maximum Active statistic
description, 232
Disk Writes Pages/sec statistic
description, 232
Disk Writes Temp Extends/sec statistic
description, 232
Disk Writes Transaction Log/sec statistic
description, 232
DiskRead
Node Statistics field descriptions, 611
statistic in access plans, 595
DiskReadIndInt
statistic in access plans, 595
DiskReadIndLeaf
statistic in access plans, 595
DiskReadTable
statistic in access plans, 595
DiskReadTime
Node Statistics field descriptions, 611
DiskWrite
Node Statistics field descriptions, 611
statistic in access plans, 595
DiskWriteTime
Node Statistics field descriptions, 611
displaying system objects in a database
about, 36
DistH plan item
abbreviations in the plan, 589
DISTINCT clause
eliminating duplicate results, 302
unnecessary distinct elimination, 529
distinct elimination
about, 529
DISTINCT keyword
aggregate functions, 353
distinct list
item in execution plans, 599
DistO plan item
abbreviations in the plan, 589
DML
about, 510
glossary definition, 876
permissions, 510
documentation
conventions, xi
SQL Anywhere, x
SQL Anywhere 11.0.0, x
domains
assigning columns, 107
case sensitivity, 644
CHECK constraints, 104
creating using SQL, 108
creating using Sybase Central, 107
dropping, 109
elements of use, 107
glossary definition, 878
using, 107
double quotes
character strings, 310
downloads
glossary definition, 878
DROP DATABASE statement
ASE, 636
DROP EXTERNLOGIN statement
using, 765
DROP INDEX statement
unavailable with snapshot isolation, 126
DROP PROCEDURE statement
using, 776
DROP SERVER statement
deleting directory access servers, 763
deleting remote servers, 759
DROP statement
automatic commit, 121
concurrency, 179
dropping
CHECK constraints, 106
column defaults, 98
data types, 109
directory access servers, 763
domains, 109
indexes, 91
materialized views, 84
procedures, 811
remote servers, 759
tables, 42
triggers, 823
user-defined data types, 109
views, 62
dropping connections
remote data access, 782
dropping external logins
about, 765
dropping indexes
indexes, 91
dropping materialized views
about, 84
dropping remote procedures
about, 776
dropping tables
about, 42
dropping views
about, 62
DT plan item
abbreviations in the plan, 589
DUMP DATABASE statement
unsupported, 636
DUMP TRANSACTION statement
unsupported, 636
duplicate correlation names in joins (star joins)
about, 394
duplicate elimination
query execution algorithms, 577
duplicate elimination algorithms
HashDistinct, 577
OrderedDistinct, 578
duplicate results
eliminating, 302
duplicate rows
removing with UNION, 365
duration
locks, 142
dynamic cache sizing
about Unix, 257
performance improvement tips, 256
Windows, 256
Windows Mobile, 256
dynamic SQL
glossary definition, 878

E
EAH plan item
abbreviations in the plan, 589
EAM plan item
abbreviations in the plan, 589
early release of locks
exceptions, 155
transactions, 158
EBFs
glossary definition, 878
editing
properties of database objects, 36
table data, 43
editing breakpoint conditions
about, 864
editing data
Sybase Central, 43
effects of
transaction scheduling, 157
unserializable transaction scheduling, 157
efficiency
improving and locks, 159
saving time when importing data, 705
EH plan item
abbreviations in the plan, 589
element directive
using, 685
elements
generating XML from relational data, 664
obtaining query results as XML, 672
storing XML in databases, 663
eliminating duplicate query results
about, 302
elimination of unnecessary case translation
about, 537
elimination of unnecessary inner and outer joins
about, 534
EM plan item
abbreviations in the plan, 589
embedded SQL
glossary definition, 878
emulating Adaptive Server Enterprise
about, 641
enabled status, materialized views
about, 68
enabling
materialized views, 80
procedure profiling in Sybase Central, 190
views, 63
enabling and disabling materialized views
about, 80
enabling and disabling optimizer use of a materialized view
about, 82
enabling and disabling views
about, 63
enabling breakpoints
about, 864
enabling procedure profiling
about, 190
enabling profiling using sa_server_option
about, 219
enabling snapshot isolation
about, 128
encoding
glossary definition, 878
encoding illegal XML names
about, 674
encrypting
materialized views, 78
encryption
cache size, 241
hiding objects, 856
materialized views, 78
ending transactions
about, 121
enforcing column uniqueness
about, 623
enforcing entity and referential integrity
about, 110
enforcing referential integrity
about, 111
ensuring compatible object names
about, 644
ensuring data integrity
about, 93
entities
about, 5
attributes, 5
choosing, 11
definition of, 5
forcing integrity, 110
entity integrity
breached by client application, 110
introduction, 96
primary keys, 631
entity-relationship diagrams
about, 5
reading, 7
enumeration phase
query processing, 526
equals operator
comparison operator, 306
equijoins
about, 383
erasing
CHECK constraints, 106
tables, 42
error handling
ON EXCEPTION RESUME, 845
procedures and triggers, 844
error handling in Transact-SQL procedures
about, 656
errors
conversion, 718
procedures and triggers, 844
Transact-SQL, 656, 657
errors and warnings in procedures and triggers
about, 844
errors on DELETE or UPDATE
about, 114
EstCpuTime
estimate in access plans, 596
EstDiskReads
estimate in access plans, 596
EstDiskReadTime
estimate in access plans, 596
EstDiskWrites
estimate in access plans, 596
Estimated Active statistic
description, 233
Estimated Cache Pages
Optimizer Statistics field descriptions, 612
estimated leaf pages
item in execution plans, 598
estimated pages
item in execution plans, 598
estimated row size
item in execution plans, 598
estimated rows
item in execution plans, 598
EstRowCount
estimate in access plans, 596
EstRunTime
     estimate in access plans, 596
event model
     glossary definition, 879
events
     generating and reviewing profiling results, 190
     statements allowed, 855
examining variables
     debugger, 866
examples
     dirty reads, 160
     implications of locking, 174
     non-repeatable reads, 164
     phantom locks, 174
     phantom rows, 169
Excel
     importing data into a SQL Anywhere database, 709
     remote data access, 797
Except algorithm
     supported by hash join algorithm, 573
     supported by merge join algorithm, 573
Except algorithms (EAH, EAM, EH, EM)
     about, 580
EXCEPT statement
     combining queries, 365
     NULL, 368
     rules, 366
     using, 366
exception handlers
     nested compound statements, 849
     procedures and triggers, 848
exceptions
     declaring, 845
Exchange algorithm (Exchange)
     about, 585
Exchange plan item
     abbreviations in the plan, 589
exclusive locks
     about, 144, 145
exclusive table locks
     about, 147
EXECUTE IMMEDIATE statement
     procedures, 851
executing
     queries more than once, 318
executing triggers
     about, 822
execution phase
query processing, 527
execution plans
     abbreviations, 589
     caching, 562
     context sensitive help, 605
     customizing the appearance, 609
     graphical plans, 602
     long text plans, 601
     printing, 609
     reading, 589
     short text plans, 600
     view matching outcomes, 596
     viewing without executing a query, 589
existence test
     about, 500
     negation of, 501
EXISTS operator
     about, 500
EXISTS predicates
     rewriting subqueries as, 537
explicit join conditions
     about, 377
     explicit join conditions (the ON clause)
     about, 381
EXPLICIT mode
     syntax, 680
     using, 680
     using the cdata directive, 688
     using the element directive, 685
     using the hide directive, 686
     using the xml directive, 687
     writing queries, 682
export tools
     about, 722
     dbunload utility, 726
     Interactive SQL wizard, 723
     OUTPUT statement, 724
     UNLOAD statement, 725
     UNLOAD TABLE statement, 724
export wizard
     using, 723
exporting
     ASE compatibility, 752
     introduction, 722
     NULL values, 730
     query results, 728
     relational data as XML, 664
     schemas, 742
extra available statistic
description, 233
extract database wizard
SQL Remote, 745
extracting
databases for SQL Remote, 745
extraction
glossary definition, 879
failover
glossary definition, 879
false conditions
NULL, 313
fan-out
indexes, 621
fastfirstrow
choosing the optimizer goal, 240
fastfirstrow table hint
NestedLoopsJoin algorithm, 576
features not supported for remote data
about, 781
Federal Information Processing Standard Publication
compliance
(see also SQL standards)
feedback
documentation, xv
providing, xv
reporting an error, xv
requesting an update, xv
fetch statement
cursor management procedures, 841
fetchtst
about, 242
file

glossary definition, 879
file fragmentation
about, 246
file manipulation statements
unsupported Transact-SQL statements, 636
file message type
glossary definition, 879
file-based downloads
glossary definition, 879
file-definition database
glossary definition, 879
files
fragmentation, 245
graphical plan, 602
Filter algorithms (Filter, PreFilter)
about, 585
Filter plan item
abbreviations in the plan, 589
filters
filter algorithms, 585
hash filter algorithms, 586
finding out more and requesting technical assistance
technical support, xv
finishing transactions
about, 121
FIPS compliance
(see also SQL standards)
FIRST clause
about, 363
first normal form
testing, 17
First-row
choosing the optimizer goal, 240
First-Row optimization goal
NestedLoopsJoin algorithm, 576
FIRST_VALUE function
eexamples, 466
usage, 457
FirstRowRunTime
Node Statistics field descriptions, 611
FOR BROWSE clause
Transact-SQL SELECT statement syntax
unsupported, 649
FOR clause
obtaining query results as XML, 672
using FOR XML AUTO, 677
using FOR XML EXPLICIT, 680
using FOR XML RAW, 676
FOR OLAP WORKLOAD option
ClusteredHashGroupBy algorithm, 579
FOR READ ONLY clause
ignored, 649
FOR statement
control statements, 830
FOR UPDATE clause
Transact-SQL SELECT statement syntax
unsupported, 649
FOR XML AUTO
using, 677
FOR XML clause
BINARY data type, 673
EXPLICIT mode syntax, 680
IMAGE data type, 673
LONG BINARY data type, 673
obtaining query results as XML, 672
restrictions, 673
usage, 673
using AUTO mode, 677
using EXPLICIT mode, 680
using RAW mode, 676
VARBINARY data type, 673
FOR XML EXPLICIT
syntax, 680
using, 680
using the cdata directive, 688
using the element directive, 685
using the hide directive, 686
using the xml directive, 687
FOR XML RAW
using, 676
foreign key constraints
glossary definition, 880
foreign keys
creating in Sybase Central, 46
creating using SQL, 47
deleting in Sybase Central, 46
displaying in Sybase Central, 46
generated indexes, 619
glossary definition, 879
inserts, 114
integrity, 631
key joins, 406
managing, 45
mandatory/optional, 111
modifying using SQL, 47
performance, 254
referential integrity, 111
role name, 407
foreign tables
glossary definition, 880
formulas
OLAP aggregate functions, 483
FORWARD TO statement
native statements, 774
sending native statements to remote servers, 774
FoxPro
remote data access, 797
fragmentation
about, 245
file, 246
indexes, 247
indexes, application profiling tutorial, 275
of files, tables, and indexes, 245
tables, 246
tables, application profiling tutorial, 279
FROM clause
derived tables in, 303
explanation of joins, 376
introduction, 303
isolation levels, 124
stored procedures in, 304
full backups
glossary definition, 880
full compares
about, 620
statistic in access plans, 595
full outer joins
about, 386
full text search
about, 323
altering text indexes, 337
boolean searching, 328
DecodePostings algorithm, 584
forbidden keywords and wildcards, 325
forming a full text query, 323
full example, 346
grouping terms and subexpressions, 325
impact of database options on text indexes, 345
listing text configuration objects, 339
listing text indexes, 332
maximum term length, 341
minimum term length, 341
obtaining scores for search results, 331
phrase searching, 327
prefix searching, 327
proximity searching, 327
searches for unindexed terms, 330
searching multiple columns, 325
stoplists, 341
term and phrase searching, 325
term breaker algorithm, 341
text configuration objects, 339
text indexes, 332
types of full text searches, 325
FullCompare
statistic in access plans, 595
FullOuterHashJoin plan item
abbreviations in the plan, 589
functions
caching, 584
create function wizard, 814
generating and reviewing profiling results, 190
idempotent or deterministic, 584
SOUNDEX function, 315
TRACEBACK, 845
tsequal, 645
user-defined, 814
window, 450
window ranking (OLAP), 474
fuzzy matching
using the FUZZY operator in full text searches, 329
G
gateways
glossary definition, 880
general guidelines for writing portable SQL
about, 647
general problems with queries
remote data access, 782
generated join conditions
about, 377
glossary definition, 880
generated joins and the ON clause
about, 382
generating
physical data models, 19
unique keys, 178
generation numbers
glossary definition, 880
getting help
technical support, xv
GLOBAL AUTOINCREMENT
default, 100
global autoincrement
compared to GUIDs and UUIDs, 101
global temporary tables
about, 52
glossary definition, 881
merging table structures, 720
sharing, 52
global variables
debugger, 866
glossary
  list of SQL Anywhere terminology, 871
go
  batch statement delimiter, 827
  Transact-SQL batches, 652
grant options
  glossary definition, 881
GRANT statement
  concurrency, 179
  Transact-SQL, 640
graphical plan with statistics
  about, 602, 603
graphical plans
  abbreviations, 589
  about, 602
  accessing using SQL functions, 610
  bypassing optimization, 605
  context sensitive help, 605
  customizing appearance, 609
  monitoring and improving performance, 602
  Node Statistics field descriptions, 611
  Optimizer Statistics field descriptions, 612
  predicate, 607
  printing, 603, 609
  reading execution plans, 602
  simple queries, 605
  viewing detailed node information, 605
  viewing in Interactive SQL, 610
  viewing without executing a query, 589
graphing
  using the Performance Monitor, 224
GrByH plan item
  abbreviations in the plan, 589
GrByHClust plan item
  abbreviations in the plan, 589
GrByHSets plan item
  abbreviations in the plan, 589
GrByO plan item
  abbreviations in the plan, 589
GrByOSets plan item
  abbreviations in the plan, 589
GrByS plan item
  abbreviations in the plan, 589
GrBySSets plan item
  abbreviations in the plan, 589
greater than
  comparison operator, 306
  range specification, 307
greater than or equal to
  comparison operator, 306
GROUP BY ALL clause
  Transact-SQL SELECT statement syntax unsupported, 649
GROUP BY clause
  about, 355
  aggregate functions, 358
  applying aggregate functions to grouped data, 320
  errors, 320
  execution, 355
  extensions, 440
  order by and, 364
  SQL standard compliance, 371
  SQL/2003 standard, 371
  using with the WHERE and HAVING clauses, 355
  WHERE clause, 358
group by multiple columns
  about, 357
group reads
  tables, 616
group-by list
  item in execution plans, 599
grouped data
  about, 319
grouping
  full text searching, 329
  using multiple columns, 357
grouping algorithms
  ClusteredHashGroupBy, 579
  HashGroupBy, 578
  HashGroupBySets, 579
  OrderedGroupBy, 579
  OrderedGroupBySets, 580
  query execution algorithms, 578
  SingleRowGroupBy, 580
  SortedGroupBySets, 580
grouping changes into transactions
  about, 120
GROUPING function
  detecting NULL placeholders, 447
  used with a CUBE clause (OLAP), 445
  used with a ROLLUP clause (OLAP), 444
groups
  ASE, 639
GUIDs
  compared to global autoincrement, 101
default column value, 101
generating, 178

handling conversion errors during import
about, 718

hash
glossary definition, 881
Hash filter algorithms (HF, HFP)
about, 586
hash filters
hash filter algorithms, 586
hash joins
HashAntisemijoin algorithm, 575
HashJoin algorithms, 573
HashSemijoin algorithm, 574
RecursiveHashJoin algorithm, 574
RecursiveLeftOuterHashJoin algorithm, 574
hash maps
hash filter algorithms, 586
HashAntisemijoin algorithm (JHA)
about, 575
HashAntisemijoin plan item
abbreviations in the plan, 589
HashDistinct algorithm (DistH)
about, 577
HashDistinct plan item
abbreviations in the plan, 589
HashExcept algorithm
Windows Mobile, 580
HashExcept plan item
abbreviations in the plan, 589
HashExceptAll plan item
abbreviations in the plan, 589
HashFilter plan item
abbreviations in the plan, 589
HashGroupBy algorithm (GrByH)
about, 578
HashGroupBy plan item
abbreviations in the plan, 589
HashGroupBySets algorithm (GrByHSets)
about, 579
HashGroupBySets plan item
abbreviations in the plan, 589
HashIntersect plan item
abbreviations in the plan, 589
HashIntersectAll plan item
abbreviations in the plan, 589
HashJoin algorithms (JH, JHSP, JHFO, JHAP, JHO, JHPO)
about, 573
HashJoin plan item
abbreviations in the plan, 589
HashSemijoin algorithm (JHS)
about, 574
HashSemijoin plan item
abbreviations in the plan, 589
HashTableScan method (HTS)
about, 571
HashTableScan plan item
abbreviations in the plan, 589
HAVING clause
logical operators, 361
performance, 548
selecting groups of data, 360
subqueries, 493
using with GROUP BY clause, 360
using with the GROUP BY clause, 355
WHERE clause and, 321
with and without aggregates, 360
Heaps Carver statistic
description, 233
Heaps Query Processing statistic
description, 233
Heaps Relocatable Locked statistic
description, 233
Heaps Relocatable statistic
description, 233
help
technical support, xv
heuristics
query optimization, 542
HF plan item
abbreviations in the plan, 589
HFP plan item
abbreviations in the plan, 589
hide directive
using, 686
hiding materialized views
about, 84
hierarchical data structures
exploring hierarchical data structures, 426
parts explosion problems, 428
hinting
about index hints, 87
index hints, 86
histograms
   about, 542
   glossary definition, 881
   updating, 543
HOLDLOCK keyword
   Transact-SQL, 649
host variables
   in batches, 827
how data can become invalid
   about, 94
how database contents change
   about, 95
how database options affect text index creation
   about, 345
how joins work
   about, 376
how NULL affects Transact-SQL outer joins
   about, 392
how queries with group by are executed
   about, 355
HTS plan item
   abbreviations in the plan, 589
I
I/O
   scanning bitmaps, 616
IAH plan item
   abbreviations in the plan, 589
IAM plan item
   abbreviations in the plan, 589
iAnywhere developer community
   newsgroups, xv
iAnywhere JDBC driver
   glossary definition, 881
IBM DB2
   migrating to SQL Anywhere, 746
   remote data access to DB2, 789
icons
   used in this Help, xiv
id
   metaproperty name, 668
idempotent functions
   defined, 584
identifiers
   case sensitivity, 644
   glossary definition, 881
   qualifying, 293
   uniqueness, 644
   using in domains, 108
   identifying entities and relationships
      about, 11
IDENTITY column
   retrieving values, 646
   special IDENTITY, 645
Idle Actives/sec statistic
   description, 228
Idle Checkpoint Time statistic
   description, 228
Idle Checkpoints/sec statistic
   description, 228
Idle Writes/sec statistic
   description, 228
if a client application breaches entity integrity
   about, 110
if a client application breaches referential integrity
   about, 112
IF statement
   control statements, 830
IGNORE NULLS clause
   usage in LAST_VALUE function, 467
IH plan item
   abbreviations in the plan, 589
IM plan item
   abbreviations in the plan, 589
images
   inserting, 516
immediate materialized views (see immediate views)
immediate views
   about, 66
   changing to a manual view, 77
   creating, 76
   immediate refresh type, 66
   only impacted rows are refreshed, 66
   restrictions when creating, 72
implementing Index Consultant recommendations
   about, 197
import tools
   about, 705
   INPUT statement, 707
   INSERT statement, 711
   Interactive SQL import wizard, 705
   LOAD TABLE statement, 709
   MERGE statement, 712
   proxy tables, 718
import wizard
statistic in access plans, 595
Index Adds/sec statistic
description, 233
Index Consultant
about, 194
assessing recommendations, 197
connecting to a version 9 database server, 194
connection state, 197
DBA or PROFILE authority required to run, 194
implementing recommendations, 197
introduction, 86
obtaining recommendations for a query, 194
server state, 197
understanding recommendations, 195
using, 194
index fan-out
about, 621
index fragmentation
about, 247
application profiling tutorial, 275
Index Full Compares/sec statistic
description, 233
index functions
row numbering, 482
index hints
using, 87
index information in the catalog
about, 92
Index Lookups/sec statistic
description, 233
index name
item in execution plans, 598
index only retrieval
IndexOnlyScan method, 569
index scans
IndexOnlyScan, 569
IndexScan, 568
MultipleIndexScan method, 569
ParallelIndexScan method, 570
index selectivity
about, 620
index sharing using logical indexes
about, 617
index-only retrieval
about, 87
indexes
about, 617
B-link, 624
benefits and locking, 159
candidate, 196
classified, 87
classified and unclassified, 623
composite, 621
computed columns, 89
correlations between, 197
costs and benefits, 194
creating, 89
deleting, 91
determining shared physical indexes, 618
effect of column order, 621
fan-out and page sizes, 617
fragmentation, 247
frequently-searched columns, 86
generated, 619
glossary definition, 882
HAVING clause performance, 548
improving performance, 620
Index Consultant, 86
index hints, 87
index-only retrievals, 569
inspecting using system views, 92
introduction, 318
leaf pages, 621
logical, 617
optimization, 617
performance, 243
physical, 617
predicate analysis, 548
querying a view using a text index, 330
rebuilding, 90
recommended page sizes, 617
restrictions and considerations, 617
sargable predicates, 548
skew, 247
statistics list, 233
structure, 621
temporary tables, 620
text indexes, 332
Transact-SQL, 644
types of, 623
understanding Index Consultant recommendations, 195
unused, 197
used to satisfy a predicate, 546
using the Index Consultant, 194
validating, 90
<table>
<thead>
<tr>
<th>Virtual, 196</th>
<th>Inputting data</th>
</tr>
</thead>
<tbody>
<tr>
<td>When to use, 86</td>
<td>Importing data, 704</td>
</tr>
<tr>
<td>WHERE clause performance, 548</td>
<td>Importing tables, 719</td>
</tr>
<tr>
<td>Working with, 86</td>
<td>Inputting data</td>
</tr>
<tr>
<td>IndexOnlyScan method (IO)</td>
<td>Importing and exporting, 702</td>
</tr>
<tr>
<td>About, 569</td>
<td>INPUT statement, 707</td>
</tr>
<tr>
<td>IndexOnlyScan plan item</td>
<td>INSERT statement, 711</td>
</tr>
<tr>
<td>Abbreviations in the plan, 589</td>
<td>LOAD TABLE statement, 709</td>
</tr>
<tr>
<td>IndexScan method</td>
<td>MERGE statement, 712</td>
</tr>
<tr>
<td>About, 568</td>
<td>Insert locks</td>
</tr>
<tr>
<td>IndexScan plan item</td>
<td>About, 148</td>
</tr>
<tr>
<td>Abbreviations in the plan, 589</td>
<td></td>
</tr>
<tr>
<td>Indirect references</td>
<td>INSERT statement</td>
</tr>
<tr>
<td>Database objects, 57</td>
<td>About, 513, 711</td>
</tr>
<tr>
<td>IndLookup</td>
<td>Considerations for materialized views, 708</td>
</tr>
<tr>
<td>Statistic in access plans, 595</td>
<td>Considerations for text indexes, 708</td>
</tr>
<tr>
<td>Inequalities</td>
<td>Inputting data</td>
</tr>
<tr>
<td>Testing for inequality, 314</td>
<td>Importing data, 704</td>
</tr>
<tr>
<td>InfoMaker</td>
<td>Importing tables, 719</td>
</tr>
<tr>
<td>Glossary definition, 882</td>
<td>Inputting data</td>
</tr>
<tr>
<td>Inheriting column CHECK constraints from domains</td>
<td></td>
</tr>
<tr>
<td>About, 104</td>
<td>Considerations for materialized views, 711</td>
</tr>
<tr>
<td>Initial cache size</td>
<td>Considerations for text indexes, 712</td>
</tr>
<tr>
<td>About, 255</td>
<td>Duplicate data, 113</td>
</tr>
<tr>
<td>Initializing materialized views</td>
<td>Locking during, 152</td>
</tr>
<tr>
<td>About, 74</td>
<td>Referential integrity check on INSERT, 113</td>
</tr>
<tr>
<td>Inlining</td>
<td>Select, 513</td>
</tr>
<tr>
<td>Simple system procedures, 540</td>
<td>Using, 711</td>
</tr>
<tr>
<td>User-defined functions, 539</td>
<td>Using to change data, 519</td>
</tr>
<tr>
<td>Inlining simple procedures</td>
<td>Insert triggers</td>
</tr>
<tr>
<td>About, 540</td>
<td>Fire as a result of INPUT statements, 702</td>
</tr>
<tr>
<td>Inlining user-defined function</td>
<td>Inserting</td>
</tr>
<tr>
<td>About, 539</td>
<td>NULLs, behavior for unspecified columns, 514</td>
</tr>
<tr>
<td>InList algorithm (IN)</td>
<td>Inserting data</td>
</tr>
<tr>
<td>About, 586</td>
<td>Behavior for unspecified columns, 514</td>
</tr>
<tr>
<td>InList plan item</td>
<td>BLOBs, 516</td>
</tr>
<tr>
<td>Abbreviations in the plan, 589</td>
<td>Column data INSERT statement, 514</td>
</tr>
<tr>
<td>Inner and outer joins</td>
<td>Constraints, 514</td>
</tr>
<tr>
<td>About, 386</td>
<td>Defaults, 514</td>
</tr>
<tr>
<td>Inner joins</td>
<td>INPUT statement, 707</td>
</tr>
<tr>
<td>About, 386</td>
<td>INSERT statement, 711</td>
</tr>
<tr>
<td>Converting from outer joins, 533</td>
<td>Into all columns, 513</td>
</tr>
<tr>
<td>Glossary definition, 882</td>
<td>MERGE statement, 712</td>
</tr>
<tr>
<td>Join elimination rewrite optimization, 534</td>
<td>Using INSERT, 513</td>
</tr>
<tr>
<td>Inserting documents and images</td>
<td>With SELECT, 515</td>
</tr>
<tr>
<td>About, 516</td>
<td>Inserting data</td>
</tr>
<tr>
<td>Inserting values into all columns of a row</td>
<td>About, 513</td>
</tr>
<tr>
<td>About, 514</td>
<td>Inserting values into specific columns</td>
</tr>
</tbody>
</table>

Copyright © 2008, iAnywhere Solutions, Inc. - SQL Anywhere 11.0.0  931
install-dir
documentation usage, xiii
INSTEAD OF triggers
about, 824
recursion, 824
using to update views, 825
instest
about, 242
integrated logins
glossary definition, 882
integrity
about, 94
checking, 113
column defaults, 97
constraints, 103
enforcing, 110
glossary definition, 882
implementing integrity constraints, 96
information in the system tables, 117
losing, 112
tools, 95
using triggers to maintain, 96
integrity constraints belong in the database
about, 94
integrity rules in the system tables
about, 117
intent locks
about, 145
snapshot isolation, 146
intent to write table locks
about, 147
inter-query parallelism
(see also intra-query parallelism)
about, 566
intra- vs. inter-query parallelism, 566
Interactive SQL
batch mode, 750
batch operations, 750
command delimiter, 853
command files, 750
default editor for .sql files, 750
displaying a list of tables, 374
effects of exiting, 121
exporting query results, 728
exporting relational data as XML, 664
glossary definition, 882
grouping changes into transactions, 121
Index Consultant, 194
loading commands, 751
rebuilding databases, 741
running scripts, 750
viewing graphical plans, 610
Interactive SQL import wizard
about, 705
interference between transactions
about, 138
element example, 167
 interleaving transactions
about, 157
internal loading
about, 702
internal operations
remote data access, 778
Intersect algorithm
supported by hash join algorithm, 573
supported by merge join algorithm, 573
intersect algorithms
Windows Mobile, 581
Intersect algorithms (IH, IM, IAH, IAM)
about, 581
INTERSECT statement
combining queries, 365
NULL, 368
rules, 366
using, 366
INTO clause
using, 837
intra-query parallelism
(see also inter-query parallelism)
about, 566
exchange algorithm, 566
intra- vs. inter-query parallelism, 566
uses the exchange algorithm, 585
invalid data
about, 94
investigating deadlocks
about, 265
Invocations
Node Statistics field descriptions, 611
invocations
statistic in access plans, 591
invoking the SQL Flagger
about, 629
IO plan item
abbreviations in the plan, 589
IS NULL keyword
about, 313
ISNULL function
  about, 313
ISO compliance
  (see also SQL standards)
ISO SQL standards
  concurrency, 131
  typical inconsistencies, 131
isolation level 0
  about, 124
  example, 160
  SELECT statement locking, 150
isolation level 1
  about, 124
  example, 164
  SELECT statement locking, 150
isolation level 2
  about, 124
  example, 169, 174
  SELECT statement locking, 150
isolation level 3
  about, 124
  example, 171
  SELECT statement locking, 150
  sequential table scans, 570
isolation level read committed
  about, 124
isolation level read uncommitted
  about, 124
isolation level readonly-statement-snapshot
  about, 124
isolation level repeatable read
  about, 124
isolation level serializable
  about, 124
isolation level snapshot
  about, 124
isolation level statement-snapshot
  about, 124
isolation levels
  about, 124
  changing within a transaction, 136
  choosing, 156
  choosing a snapshot isolation level, 156
  choosing types of locking tutorial, 167
  glossary definition, 882
  implementation at level 0, 150
  implementation at level 1, 150
  implementation at level 2, 150
  implementation at level 3, 150
  improving concurrency at levels 2 and 3, 158
  ODBC, 135
  setting, 134
  tutorials, 160
  typical transactions for each, 158
  versus typical inconsistencies, 132, 169, 174
  versus typical transactions, 157
  viewing, 137
isolation levels and consistency
  about, 124
isolation_level
  Optimizer Statistics field descriptions, 612
ISYSFK
  system table usage, 92
ISYSIDX
  system table usage, 92, 618
ISYSIDX system table
ISYSIDXCOL
  system table usage, 92
ISYSPHYSIDX
  system table usage, 92, 618
ISYSPHYSIDX system table

J
JAR files
  glossary definition, 882
Java classes
  glossary definition, 883
jConnect
  glossary definition, 883
JDBC
  glossary definition, 883
  materialized view candidacy, 551
JDBC classes
  configuration notes, 799
  limitations, 799
JDBC-based server classes
  about, 799
JH plan item
  abbreviations in the plan, 589
JHA plan item
  abbreviations in the plan, 589
JHAP plan item
  abbreviations in the plan, 589
JHFO plan item
abbreviations in the plan, 589
JHO plan item
abbreviations in the plan, 589
JHPO plan item
abbreviations in the plan, 589
JHR plan item
abbreviations in the plan, 589
JHRO plan item
abbreviations in the plan, 589
JHS plan item
abbreviations in the plan, 589
JHSP plan item
abbreviations in the plan, 589
JM plan item
abbreviations in the plan, 589
JMFO plan item
abbreviations in the plan, 589
JMO plan item
abbreviations in the plan, 589
JNL plan item
abbreviations in the plan, 589
JNLA plan item
abbreviations in the plan, 589
JNLFO plan item
abbreviations in the plan, 589
JNLO plan item
abbreviations in the plan, 589
JNLS plan item
abbreviations in the plan, 589
join algorithms
about, 572
hash, merge, and nested loops join variants, 572
HashAntisemijoin, 575
HashJoin, 573
HashSemijoin, 574
MergeJoin, 576
NestedLoopsAntisemijoin, 577
NestedLoopsJoin, 576
NestedLoopsSemijoin, 576
RecursiveHashJoin, 574
RecursiveLeftOuterHashJoin, 574
join compatible data types
about, 380
join conditions
glossary definition, 883
types, 383
join operators
Transact-SQL, 650
join types
glossary definition, 883
joining more than two tables
about, 379
joining remote tables
about, 771
joining tables from multiple local databases
about, 773
joining two tables
about, 379
joins
about, 376
automatic, 631
Cartesian product, 385
commas, 385
conversion of outer joins to inner joins, 533
converting subqueries into, 502
converting subqueries to joins, 502
CROSS APPLY and OUTER APPLY joins, 399
cross joins, 385
data type conversion, 380
default is KEY JOIN, 378
delete, update and insert statements, 380
derived tables, 398
duplicate correlation names, 394
equijoins, 383
FROM clause, 376
glossary definition, 883
HashAntisemijoin algorithm, 575
HashJoin algorithm, 573
HashSemijoin algorithm, 574
how an inner join is computed, 379
inner, 386
inner and outer, 386
introduction, 374
join conditions, 377
join elimination rewrite optimization, 534
joined tables, 378
key, 631
key joins, 406
MergeJoin algorithm, 576
more than two tables, 379
natural, 631
natural joins, 402
NestedLoopsAntisemijoin algorithm, 577
NestedLoopsJoin algorithm, 576
NestedLoopsSemijoin algorithm, 576
nesting, 379
non-ANSI joins, 380
null-supplying tables, 386
of two tables, 379
ON clause, 381
or subqueries, 490
outer, 386
preserved tables, 386
query execution algorithms, 572
RecursiveHashJoin algorithm, 574
RecursiveLeftOuterHashJoin algorithm, 574
remote tables, 771
resulting from apply expressions, 399
search conditions, 383
self-joins, 393
star joins, 394
table expressions, 379
tables in different databases, 773
Transact-SQL, 650
Transact-SQL outer and NULL values, 392
Transact-SQL outer and views, 392
Transact-SQL restrictions on outer, 391
updating cursors, 380
WHERE clause, 383
joins resulting from apply expressions
about, 399
Joins: Retrieving Data from Several Tables
about, 373

K
key joins
about, 406
glossary definition, 880
if more than one foreign key, 407
lists and table expressions that do not contain
commas, 412
ON clause, 382
rules, 416
table expression lists, 411
table expressions, 409
table expressions that do not contain commas, 410
views and derived tables, 414
with an ON clause, 406
key type
item in execution plans, 598
keys
assigning, 16
performance, 254
keywords
HOLDLOCK, 649
NOHOLDLOCK, 649
remote servers, 781

L
LAST.VALUE function
examples, 466
usage, 457
leaf pages
about, 621
least distance problems
about, 432
LEAVE statement
control statements, 830
left outer joins
about, 386
LeftOuterHashJoin plan item
abbreviations in the plan, 589
less than
comparison operator, 306
range specification, 307
less than or equal to
comparison operator, 306
LIKE search condition
introduction, 308
optimizations, 533
wildcards, 309
limitations
JDBC classes, 799
remote data access character set conversion, 755
limiting rows
FIRST clause, 363
TOP clause, 363
limiting the memory used by the cache
performance improvement tips, 255
line breaks
SQL, 293
linear regression functions
OLAP, 472
Listeners
glossary definition, 883
listing remote server capabilities
about, 761
listing the columns on a remote table
about, 770
listing the remote tables on a server
about, 760
LOAD DATABASE statement
unsupported, 636
LOAD statement
about, 709
using, 709
LOAD TABLE statement
considerations for materialized views, 710
considerations for text indexes, 710
using, 719
LOAD TRANSACTION statement
unsupported, 636
loading
commands in Interactive SQL, 751
considerations for database recovery, 710
considerations for mirroring, 711
considerations for synchronization, 710
loading data
conversion errors, 718
local temporary tables
about, 52
glossary definition, 884
local variables
debugger, 866
localname
glossary definition, 884
Lock Count statistic
description, 236
lockable objects
about, 142
locked tables
item in access plans, 597
locking
(see also locks)
about, 142
conflicts, 149
duration, 142
during deletes, 154
during inserts, 152
during queries, 150
during updates, 153
exclusive table locks, 147
insert locks, 148
intent locks, 145
intent to write table locks, 147
phantom locks, 148
position table locks, 147
reducing through indexes, 623
shared table locks, 146
tables, 146
locks
(see also locking)
about, 142
blocking, 138
blocking example, 167
choosing isolation levels tutorial, 167
conflict handling, 138, 167
conflicting types, 146
conflicts, 149
deadlock, 138
duration, 142
eyear release of, 158
eyear release of read locks, 155
exclusive, 145
exclusive schema, 144
exclusive table, 147
glossary definition, 884
implementation at isolation level 0, 150
implementation at isolation level 1, 150
implementation at isolation level 2, 150
implementation at isolation level 3, 150
inconsistencies versus typical isolation levels, 132
insert, 148
intent, 145
intent to write table, 147
isolation levels, 124
objects that can be locked, 142
orphans and referential integrity, 153
phantom, 148
phantom rows versus isolation levels, 169, 174
position table, 147
procedure for deletes, 154
procedure for inserts, 152
procedure for updates, 153
read, 144
reducing the impact through indexes, 159
row, 144
schema, 144
shared schema, 144
shared table, 146
table, 146
transaction blocking and deadlock, 138
typical transactions versus isolation levels, 157
viewing in Sybase Central, 143
viewing information, 143
viewing using the sa_locks system procedure, 143

936 Copyright © 2008, iAnywhere Solutions, Inc. - SQL Anywhere 11.0.0
abbreviations in the plan, 589
merging
  behavior with triggers, 714
merging different table structures
  about, 720
message log
  glossary definition, 885
MESSAGE statement
  procedures, 845
message stores
  glossary definition, 885
message systems
  glossary definition, 885
message types
  glossary definition, 885
metadata
  glossary definition, 886
metaproperty names
  id, 668
  localname, 668
Microsoft Access
  migrating to SQL Anywhere, 746
  remote data access, 795
Microsoft Excel
  importing data into a SQL Anywhere database, 709
  remote data access, 797
Microsoft FoxPro
  remote data access, 797
Microsoft SQL Server
  migrating to SQL Anywhere, 746
migrate database wizard
  about, 746
migrating databases
  about, 746
  migrate database wizard, 746
  using sa_migrate system procedures, 747
migrating databases to SQL Anywhere
  about, 746
MIN function
  equivalent mathematical formula, 483
  rewrite optimization, 562
minimal administration work
  optimizer, 544
minimize cascading referential actions
  about, 242
minimizing downtime during rebuilding
  about, 743
minimum cache size
  about, 255
mirror logs
  glossary definition, 886
miscellaneous statistics
  list, 237
MobiLink
  glossary definition, 886
  rebuilding databases, 740
MobiLink clients
  glossary definition, 886
MobiLink Monitor
  glossary definition, 886
MobiLink server
  glossary definition, 886
MobiLink system tables
  glossary definition, 886
MobiLink users
  glossary definition, 886
modifying
  column defaults, 98
  regular views with dependencies, 60
  tables with view dependencies, 39
  modifying and dropping column defaults
  about, 98
monitor
  opening the Performance Monitor, 225
  Performance Monitor overview, 224
monitor query performance
  performance improvement tips, 242
monitoring and improving performance
  about, 185
monitoring cache size
  about, 258
monitoring database performance
  about, 223
monitoring database statistics from Sybase Central
  about, 224
monitoring database statistics using Windows
  Performance Monitor
  about, 226
monitoring performance
  abbreviations used in execution plans, 589
  Performance Monitor statistics, 227
  reading execution plans, 589
  tools to measure queries, 242
monitoring statistics using SQL functions
  about, 223
moving data
about, 701
exporting, 722
importing, 704
importing and exporting, 702
performance, 702
msaccessodbc server class
about, 795
msodbc server class
about, 793
Multi-Page Allocations statistic
description, 233
MultIdx plan item
abbreviations in the plan, 589
multiple databases
joins, 773
multiple row subqueries
about, 486
multiple transactions
concurrency, 122
MultipleIndexScan method (MultIdx)
about, 569
MultipleIndexScan plan item
abbreviations in the plan, 589
MySQL
ODBC server class, 794
mysqlodbc server class
about, 794

N
n-grams
defined, 341
name spaces
indexes, 644
triggers, 644
naming savepoints
about, 123
nesting
derived tables in joins, 398
joins, 379
outer joins, 388
network protocols
glossary definition, 886
network server
glossary definition, 887
NEWID function
default column value, 101
when to use, 178
newsgroups
technical support, xv
ngrams
defined, 341
NOHOLDLOCK keyword
ignored, 649
non-ANSI joins
about, 380
non-deterministic functions
side-effects, 584
non-dirty reads
tutorial, 160
non-replicable reads
about, 132
example, 165
isolation levels, 132

nested loops
NestedLoopsJoin algorithm, 576
NestedLoopsSemijoin algorithm, 576
nested loops joins
NestedLoopsAntisemijoin algorithm, 577
nested subqueries
about, 489
NestedLoopsAntisemijoin algorithm (JNLA)
about, 577
NestedLoopsAntisemijoin plan item
abbreviations in the plan, 589
NestedLoopsJoin algorithms (JNL, JNLO, JNLO)
about, 576
NestedLoopsJoin plan item
abbreviations in the plan, 589
NestedLoopsSemijoin algorithm (JNLS)
about, 576
NestedLoopsSemijoin plan item
abbreviations in the plan, 589
nesting
derived tables in joins, 398
joins, 379
outer joins, 388
nesting savepoints
about, 123
n-grams
defined, 341
NEXID function
default column value, 101
when to use, 178
newsgroups
technical support, xv
ngrams
defined, 341
NOHOLDLOCK keyword
ignored, 649
non-ANSI joins
about, 380
non-deterministic functions
side-effects, 584
non-dirty reads
tutorial, 160
non-replicable reads
about, 132
example, 165
isolation levels, 132
tutorial, 164
normal forms
  first normal form, 17
  normalizing database designs, 15
  second normal form, 18
  third normal form, 19
normalization
  about, 15
  glossary definition, 887
  performance benefits, 243
normalize your table structure
  performance improvement tips, 243
NOT
  using logical operators, 313
NOT BETWEEN keyword
  range queries, 307
not equal to
  comparison operator, 306
not greater than
  comparison operator, 306
NOT keyword
  example, 306
not less than
  comparison operator, 306
Notes and remote access
  about, 798
Notifiers
  glossary definition, 887
NULL
  about, 311
  aggregate functions, 353
  allowing in columns, 24
  as different from zeros or blanks, 311
  column default, 642
  column definition, 313
  comparing, 312
  default, 102
  default parameters, 312
  DISTINCT clause, 302
  EXCEPT statement, 368
  INTERSECT statement, 368
  output, 730
  placeholder in OLAP, 447
  properties, 313
  results in UNKNOWN when used in comparison, 312
  set operators, 368
  sort order, 363
Transact-SQL compatibility, 647
Transact-SQL outer joins, 392
UNION statement, 368
NULL values
  ignoring conversion errors, 718
  importing data, 720
  inserting, 514
null-supplying tables
  in outer joins, 386
Number of Grant Fails statistic
  description, 233
Number of Grant Requests statistic
  description, 233
Number of Grant Waits statistic
  description, 233
O
  object trees
    glossary definition, 887
objects
  hiding, 856
  lockable objects, 142
objects that can be locked
  about, 142
obtaining database statistics from a client application
  about, 223
obtaining information about locks
  about, 143
obtaining query results as XML
  about, 672
ODBC
  applications, and locking, 135
  external servers, 785
  glossary definition, 887
  materialized view candidacy, 551
  setting isolation levels, 135
ODBC Administrator
  glossary definition, 887
ODBC data sources
  glossary definition, 887
ODBC server classes
  about, 785, 796
  Adaptive Server Enterprise, 786
  Advantage Database Server, 789
  DB2, 789
  Lotus Notes SQL 2.0, 798
  Microsoft Access, 795
Index

Microsoft Excel, 797
Microsoft FoxPro, 797
MySQL, 794
Oracle, 791
SQL Anywhere, 786
SQL Server, 793
UltraLite, 786
odbcfet
about, 242
OLAP
about, 437
basic aggregate functions, 457
correlation functions, 472
CUBE clause, 445
GROUP BY clause extensions, 440
improving OLAP performance, 438
introduction, 438
linear regression functions, 472
ROLLUP clause, 444
row numbering functions, 482
standard deviation functions, 468
variance functions, 468
window aggregate functions, 456
window functions, 450
window ranking functions, 474
WITH CUBE clause, 447
WITH ROLLUP clause, 445
OLAP functions
formulas, 483
OMNI (see remote data access)
ON clause
introduction, 381
joins, 381
ON EXCEPTION RESUME clause
about, 845
not with exception handling, 849
stored procedures, 844
Transact-SQL, 657
one-to-many relationships
definition of, 7
resolving, 20
one-to-one relationships
definition of, 6
resolving, 20
online analytical processing (see OLAP)
online books
PDF, x
OPEN statement
cursor management procedures, 841
opening the Performance Monitor
about, 225
OpenString algorithm (OpenString)
about, 587
OPENSTRING clause
OpenString clause
OpenString plan item
abbreviations in the plan, 589
openxml system procedure
using, 665
using with xp_read_file, 668
operators
arithmetic, 300
connecting conditions, 313
NOT keyword, 306
precedence, 300
optimization
about, 541
cost based, 541
reading execution plans, 589
optimization for MIN and MAX functions
about, 562
optimization goal
execution plans, 596
optimization of LIKE predicates
about, 533
optimization of OR and IN-list predicates
about, 532
optimization of queries
about, 541
assumptions, 544
phases of, 526
reading execution plans, 589
rewriting subqueries as EXISTS predicates, 537
optimization phase
query processing, 526
Optimization Time
Optimizer Statistics field descriptions, 612
optimization_goal
Optimizer Statistics field descriptions, 612
optimization_goal_option
using, 545
optimization_level
Optimizer Statistics field descriptions, 612
optimization_workload
Optimizer Statistics field descriptions, 612
optimization_workload_option
ClusteredHashGroupBy algorithm, 579
using, 545
optimize for first rows or for entire result set
about, 545
optimize for mixed or OLAP workload
about, 545
optimizer
about, 541
assumptions, 544
bypass, 527
minimal administration work, 544
phases of query processing, 526
predicate analysis, 548
semantic subquery transformations, 528
semantic transformations, 528
using materialized views, 82
optimizer estimates
about, 542
optional foreign keys
about, 111
optional relationships
about, 8
options
blocking, 138
DEFAULTS, 720
isolation_level, 134
OR
using logical operators, 313
Oracle
data type conversions, 792
migrating to SQL Anywhere, 746
Oracle and remote access
about, 791
oraodbc server class
about, 791
ORDER BY and GROUP BY
about, 364
ORDER BY clause
composite indexes, 622
examples, 316
GROUP BY, 364
limiting results, 363
materialized view definitions, 72
performance, 249
regular view definition restrictions, 57
required to ensure rows always appear in same order,
318
sorting query results, 362
using indexes to improve performance, 318
order-by
item in execution plans, 599
OrderedDistinct algorithm (DistO)
about, 578
OrderedDistinct plan item
abbreviations in the plan, 589
OrderedGroupBy algorithm (GrByO)
about, 579
OrderedGroupBy plan item
abbreviations in the plan, 589
OrderedGroupBySets algorithm (GrByOSets)
about, 580
OrderedGroupBySets plan item
abbreviations in the plan, 589
ordering of transactions
about, 157
organization
of data, physical, 615
organizing query results into groups
about, 355
orphan and referential integrity
about, 153
other diagnostic tools and techniques
about, 217
other uses for indexes
about, 623
OUT parameters
defined, 833
OUTER APPLY clause
about, 399
element, 399
outer joins
about, 386
and join conditions, 388
complex, 388
converting to inner joins, 533
glossary definition, 887
join elimination rewrite optimization, 534
restrictions, 391
star join example, 396
Transact-SQL, 390, 650
Transact-SQL and views, 392
Transact-SQL restrictions on, 391
views and derived tables, 389
outer references
about, 489
aggregate functions, 351
defined, 489
HAVING clause, 493
output redirection
about, 728
OUTPUT statement
about, 724
exporting query results, 728
using to export data as XML, 664
outputting
export tools, 722
exporting data, 722
exporting databases, 730
exporting query results, 728
exporting tables, 731
NULLs, 730
outputting data
importing and exporting, 702
saving to a file, 751
to a file, 725
outputting NULLs
about, 730
overview of OLAP functionality
about, 438

P
packages
glossary definition, 888
page maps
item in execution plans, 598
scanning, 616
page sizes
about, 616
and indexes, 617
considerations for Windows Mobile, 616
disk allocation for inserted rows, 615
performance, 250
performance considerations, 616
pages
disk allocation for inserted rows, 615
item in execution plans, 599
Pages Granted statistic
description, 233
parallel table scans
about, 571
ParallelHashAntisemijoin plan item
abbreviations in the plan, 589
ParallelHashFilter plan item
abbreviations in the plan, 589
ParallelHashSemijoin plan item
abbreviations in the plan, 589
ParallelIndexScan method
about, 570
ParallelIndexScan plan item
abbreviations in the plan, 589
parallelism
about, 566
circumstances under which parallelism is used, 567
ParallelLeftOuterHashJoin plan item
abbreviations in the plan, 589
ParallelTableScan method
about, 571
ParallelTableScan plan item
abbreviations in the plan, 589
parameters
to functions, 319
parentheses
in arithmetic statements, 300
UNION operators, 365
parse trees
glossary definition, 888
query processing, 526
partial index scan
about, 623
partial passthrough of the statement
remote data access, 779
particular concurrency issues
about, 178
parts explosion problems
about, 428
passing parameters to functions
about, 834
passing parameters to procedures
about, 834
passwords
case sensitivity, 644
Lotus Notes, 798
pattern matching
introduction, 308
PC plan item
abbreviations in the plan, 589
PCTFREE setting
reducing table fragmentation, 246
PDB
glossary definition, 888
PDF
PERCENT_RANK function

equivalent mathematical formula, 483
usage, 480

PercentTotalCost

Node Statistics field descriptions, 611
performance
about, 239
advanced application profiling, 199
All-rows optimization goal, 260
application profiling, 188
automatic tuning, 544
bulk loading, 702
cache read-hit ratio, 604
comparing optimizer estimates and actual statistics, 603
estimate source, 604
file fragmentation, 246
improving, 86, 87
improving versus locks, 159
indexes, 86, 253
keys, 254
list of improvement tips, 239
measuring query speed, 242
minimize cascading referential actions, 242
monitoring, 223, 227
monitoring using the Performance Monitor, 224
monitoring using Windows Performance Monitor, 226
optimizer workload, 240
page sizes, 250
predicate analysis, 548
reading execution plans, 589
rebuild your database, 245
recommended page sizes, 616
runtime actual and estimated, 604
scattered reads, 251
selectivity, 603
statistics in Windows Performance Monitor, 226
table and page sizes, 616
tools for monitoring and improving performance, 185
WITH EXPRESS CHECK, 260
work tables, 260
performance impacts of bulk operations
about, 702
performance improvement tips
about, 239

acquire adequate hardware, 239
bulk operations, 252
cache, 241
checking for concurrency issues, 239
choosing the optimizer goal, 240
consider collecting statistics on small tables, 241
declare collecting statistics, 241
delayed commits, 252
dynamic cache sizing, 256
in memory mode, 252
limiting the memory used by the cache, 255
monitor query performance, 242
normalize your table structure, 243
place different files on different devices, 243
reduce expensive user-defined functions, 249
reduce fragmentation, 245
reduce primary key width, 248
reduce requests between client and server, 249
reduce table widths, 248
reducing index fragmentation and skew, 247
reducing table fragmentation, 246
replace expensive triggers, 249
review the order of columns in tables, 243
separate primary and foreign key indexes, 254
specify the correct cursor type, 249
strategic sorting of query results, 249
supply explicit selectivity estimates sparingly, 250
transaction log, 239
turn off autocommit mode, 250
use an appropriate page size, 250
use appropriate data types, 251
use AUTOINCREMENT to create primary keys, 252
use indexes effectively, 253
use keys effectively, 254
use the cache to improve performance, 254
use the compression features, 259
use the WITH EXPRESS CHECK option when validating tables, 260
use work tables in query processing, 260

Performance Monitor
about, 224
adding and removing statistics, 225
list of supported statistics, 227
opening in Sybase Central, 225
overview, 224
Sybase Central, 224
Windows Performance Monitor, 226
Performance Monitor statistics
   about, 227
performance statistics
   glossary definition, 888
performance tips for importing data
   about, 705
performance tools
   graphical plans, 602
   procedure profiling system procedures, 219
   timing utilities, 222
PerformanceFetch
   about, 242
PerformanceInsert
   about, 242
PerformanceTraceTime
   about, 242
PerformanceTransaction
   about, 242
performing request trace analysis
   about, 214
permissions
   ASE, 639
   data modification, 510
   debugging, 858
   procedure result sets, 837
   triggers, 824
   user-defined functions, 816
personal server
   glossary definition, 888
phantom locks
   about, 148, 174
phantom rows
   data inconsistencies, 132
   preventing with isolation level 2, 150
   tutorial, 169
   versus isolation levels, 132, 174
phrase searching
   full text search, 327
phrases, full text searching
   about, 325
physical data models
   generating, 19
physical data organization and access
   about, 615
physical indexes
   about, 617
   determining shared physical indexes, 618
   glossary definition, 888
place different files on different devices
   performance improvement tips, 243
plan building phase
   query processing, 527
plan caching
   about, 562
plan viewer
   accessing, 610
   Node Statistics field descriptions, 611
   Optimizer Statistics field descriptions, 612
planning for capacity
   about, 213
planning for recovery when loading client-side data
   about, 734
plans
   abbreviations used in, 589
   caching, 562
   context sensitive help, 605
   customizing graphical plans, 609
   graphical plans, 602
   long text plans, 601
   printing, 609
   reading, 589
   short text plans, 600
   viewing without executing a query, 589
plug-in modules
   glossary definition, 888
plus operator
   NULL values, 313
policies
   glossary definition, 888
polling
   glossary definition, 888
portable SQL
   writing, 647
position locks
   about, 142
   duration, 142
position table locks
   about, 147
   insert locks, 148
   phantom locks, 148
PowerBuilder
   remote data access, 756
PowerDesigner
   glossary definition, 888
PowerJ
   glossary definition, 889
practical locking implications tutorial
  about, 174
pre-optimization phase
  query processing, 526
predicate
  item in execution plans, 598
predicate analysis
  about, 548
predicate push-down in UNION or GROUPed views and derived tables
  about, 531
predicates
  glossary definition, 889
  optimizer, 548
  optimizing IN-lists, 532
  optimizing LIKE, 533
  performance, 548
  reading in execution plans, 607
  usage, 305
PreFilter plan item
  abbreviations in the plan, 589
prefix searching
  full text search, 327
prefixes, full text searching
  about, 325
PREPARE statement
  remote data access, 777
PREPARE TRANSACTION statement
  remote data access, 777
preserved tables
  in outer joins, 386
primary key constraints
  glossary definition, 889
primary keys
  AUTOINCREMENT, 99
  concurrency, 178
  creating in Sybase Central, 44
  creating using SQL, 44
  entity integrity, 110
  generated indexes, 619
  generation, 178
  GLOBAL AUTOINCREMENT, 100
  glossary definition, 889
  integrity, 631
  managing, 43
  modifying in Sybase Central, 44
  modifying using SQL, 44
  performance, 254
  using NEWID to create UUIDs, 101
primary keys enforce entity integrity
  about, 110
primary tables
  glossary definition, 889
  probe values
    item in execution plans, 599
ProcCall algorithm (PC)
  about, 587
  ProcCall plan item
    abbreviations in the plan, 589
procedure calls
  ProcCall algorithm, 587
procedure language
  overview, 651
procedure profiling
  baselining, 284
  disabling, 191
  enabling, 190
  in Sybase Central, 190
  objects you can profile, 192
  performing using system procedures, 219
  resetting, 191
  understanding profiling results, 192
  using sa_server_option to disable, 220
  using sa_server_option to reset profiling, 220
  using sa_server_option to set profiling filters, 219
  using system procedures to retrieve profiling data, 221
procedure profiling in application profiling mode
  about, 190
procedure profiling using system procedures
  about, 219
procedures
  about, 805
  altering using Sybase Central, 809
  benefits of, 807
  caching statements, 562
  calling, 810
  command delimiter, 853
  considerations when referencing temporary tables, 53
  copying, 811
  create procedure wizard, 808
  creating, 808
  creating remote procedures, 775
  cursors, 841
  dates, 853
default error handling, 844
deleting, 811
deleting remote procedures, 776
error handling, 656, 657, 844
exception handlers, 848
EXECUTE IMMEDIATE statement, 851
generating and reviewing profiling results, 190
multiple result sets from, 839
overview, 806
parameters, 833, 834
permissions for result sets, 837
ProcCall algorithm (PC), 587
result sets, 812, 837
return values, 656
returning results, 836
returning results from, 811
savepoints, 852
security, 807
statements allowed, 855
statistics, 543
structure, 833
table names, 853
times, 853
tips for writing, 853
Transact-SQL, 653
Transact-SQL overview, 651
translation, 653
using, 808
using cursors in, 841
using in the FROM clause, 304
variable result sets from, 839
verifying input, 854
warnings, 847
production database
about, 199
profiling applications
about, 188
profiling database
creating internally vs. externally, 200
program variables
common table expression, 423
projections
about, 296
properties
setting all database object properties, 36
properties of NULL
about, 313
PROPERTY function
about, 223
proximity searching
full text search, 327
proxy tables
about, 767
creating, 754, 768, 769
creating from Sybase Central, 768
deleting from directory access servers, 763
glossary definition, 889
importing data, 718
location, 767
publication updates
glossary definition, 890
publications
glossary definition, 889
publisher
glossary definition, 889
push notifications
glossary definition, 890
push requests
glossary definition, 890
Q
QAnywhere
glossary definition, 890
QAnywhere Agent
glossary definition, 890
QOG
query processing, 526
qualifications
about, 305
qualified names
database objects, 293
quantified comparison test
about, 504
subqueries, 495
queries
about, 292
circumstances under which parallelism is used, 567
common table expressions, 420
elimination of unnecessary case translation, 537
elimination of unnecessary inner and outer joins, 534
execution plans, 589
exporting, 728
glossary definition, 890
optimization, 541
query memory
  about, 547
query normalization
  remote data access, 778
query optimization
  (see also optimizer)
  IN-list predicates, 532
  LIKE predicates, 533
  optimizer bypass, 527
query optimization and execution
  about, 525
query optimizer
  (see also optimizer)
  about, 541
query parsing
  remote data access, 778
query performance
  cache reads and hits, 604
  estimate sources, 603
  identifying data fragmentation problems, 605
  lack of effective indexes, 604
  predicate selectivity, 603
  reading execution plans, 589
  RowsReturned statistic, 603
  selectivity statistics, 603
Query Plan cache pages statistic
  description, 237
query preprocessing
  remote data access, 778
query processing
  phases of, 526
query results
  exporting, 728
query rewrite phase
  query processing, 526
Query Rows materialized/sec statistic
  description, 237
query transformations
  inlining of simple system procedures, 540
  inlining of user-defined functions, 539
QueryMemActiveEst
  Optimizer Statistics field descriptions, 612
QueryMemActiveMax
  Optimizer Statistics field descriptions, 612
QueryMemLikelyGrant
  Optimizer Statistics field descriptions, 612
QueryMemMaxUseful
  Node Statistics field descriptions, 611
  Optimizer Statistics field descriptions, 612
QueryMemNeedsGrant
  Optimizer Statistics field descriptions, 612
QueryMemPages
  Optimizer Statistics field descriptions, 612
quotation marks
  Adaptive Server Enterprise, 310
  character strings, 310
  quoted_identifier option
    about, 310
    setting for Transact-SQL compatibility, 642
R
RAISERROR action
  using for a merge operation, 717
RAISERROR statement
  ON EXCEPTION RESUME, 657
    Transact-SQL, 657
RANGE clause
  using, 453
range queries
  about, 307
RANK function
  equivalent mathematical formula, 483
  usage, 474
rank functions
  finding top and bottom percentiles, 481
ranking
  using with aggregation, 477
ranking functions
  examples, 474
ranking functions with windows
  about, 474
RAW mode
  using, 676
RDBMS
  glossary definition, 891
read committed
  introduction, 124
  SELECT statements, 150
  setting for ODBC, 135
  types of inconsistency, 132
read locks
  about, 144
  long-term, 145
READ statement
  executing command files, 750
read uncommitted
  introduction, 124
  SELECT statements, 150
  setting for ODBC, 135
  types of inconsistency, 132
READ_CLIENT_FILE function
  importing from, and exporting to, client computers, 733
READCLIENTFILE authority
  importing from, and exporting to, client computers, 733
READCOMMITTED table hint
  (see also read committed)
reading entity-relationship diagrams
  about, 7
reading execution plans
  about, 589
reading the execution plan
  key statistics, 603
readonly-statement-snapshot isolation level
  SELECT statement locking, 151
  using, 156
READUNCOMMITTED table hint
  (see also read uncommitted)
reasons to rebuild databases
  about, 737
rebuild and reload tools
  about, 741
rebuild tools
  about, 741
  dbisql utility, 741
  dbunload utility, 741
  UNLOAD TABLE statement, 742
rebuild your database
  performance improvement tips, 245
rebuilding databases
  about, 736
  indexes, 90
  minimizing downtime, 743
  purpose, 736
  tools, 741
rebuilding databases
  about, 736
  command line, 743
  compared to exporting, 736
  considerations, 738
  MobiLink, 740
  non-replicating databases, 739
  reasons, 737
  reducing table fragmentation, 246
  replicating databases, 740
  tools, 741
  UNLOAD TABLE statement, 742
  using dbunload for databases involved in synchronization, 740
rebuilding databases involved in synchronization
  about, 740
rebuilding databases not involved in replication
  about, 739
rebuilding databases not involved in synchronization
  about, 739
rebuilding indexes
  about, 90
recalculating
  computed columns, 50
recalculating computed columns
  about, 50
ReceivingTracingFrom
  tracing configuration, 201
recommended indexes tab
  Index Consultant recommendations, 196
recovery
  import/export, 703
  loading client-side data, 734
Recovery I/O Estimate statistic
  description, 228
recovery statistics
  list, 228
Recovery Urgency statistic
  description, 228
recursion
  max_recursive_iterations option, 427
recursive queries
  restrictions, 427
recursive subqueries
  about, 419
  data type declarations in, 431
  least distance problems, 432
  multiple aggregation levels, 423
  parts explosion problems, 428
RecursiveHashJoin algorithm (JHR)
  about, 574
RecursiveHashJoin plan item
  abbreviations in the plan, 589
RecursiveLeftOuterHashJoin algorithm (JHRO)
  about, 574
RecursiveLeftOuterHashJoin plan item abbreviations in the plan, 589
RecursiveTable algorithm (RT) about, 582
RecursiveTable plan item abbreviations in the plan, 589
RecursiveUnion algorithm (RU) about, 582
RecursiveUnion plan item abbreviations in the plan, 589
redirecting output to files, 728
Redirector glossary definition, 890
reduce expensive user-defined functions performance improvement tips, 249
reduce fragmentation performance improvement tips, 245
reduce primary key width performance improvement tips, 248
reduce requests between client and server performance improvement tips, 249
reduce table widths performance improvement tips, 248
reducing file fragmentation about, 246
reducing table fragmentation performance improvement tips, 246
reducing the impact of locking about, 159
reference databases glossary definition, 891
referenced object glossary definition, 891
references displaying references from other tables, 46
referencing object glossary definition, 891
referential integrity about, 94 actions, 112 breached by client application, 112 check performed during DELETE, 114 check performed during INSERT, 113 checking, 113 column defaults, 97 constraints, 96, 103 enforcing, 110, 111 foreign keys, 111 glossary definition, 891 information in the system tables, 117 introduction, 96 losing, 112 orphans, 153 primary keys, 631 system triggers, 112 tools, 95 verification at commit, 153 referential integrity actions implemented by system triggers, 113 reflexive relationships about, 8 REFRESH MATERIALIZED VIEW statement unavailable with snapshot isolation, 126 REFRESH TEXT INDEX statement using, 336 refresh type changing for a materialized view, 76 refresh types manual and immediate views, 66 refreshing choosing a type for refreshing text indexes, 333 refreshing manual views about, 75 REGR_AVGX function equivalent mathematical formula, 483 formula, 472 REGR_AVGY function equivalent mathematical formula, 483 formula, 472 REGR_COUNT function equivalent mathematical formula, 483 formula, 472 REGR_INTERCEPT function equivalent mathematical formula, 483 formula, 472 REGR_R2 function equivalent mathematical formula, 483 formula, 472 REGR_SLOPE function equivalent mathematical formula, 483 formula, 472 REGR_SXX function equivalent mathematical formula, 483 formula, 472 REGR_SXY function
equivalent mathematical formula, 483
formula, 472
REGR_SYY function
equivalent mathematical formula, 483
formula, 472
regular expressions
glossary definition, 891
regular views
quick comparison with materialized views and base
tables, 54
relational data
exporting as XML, 664
relationships
about, 6
cardinality of, 6
changing to an entity, 9
choosing, 11
definition of, 5
mandatory versus optional, 8
many-to-many, 7
one-to-many, 7
one-to-one, 6
reflexive, 8
resolving, 19
roles, 7
relative benefit
Index Consultant recommendations, 196
reload.sql
about, 736
exporting table data, 742
exporting tables, 731
rebuilding databases, 736
rebuilding remote databases, 736
reloading databases, 743
reloading databases
(see also rebuilding databases)
about, 736
command line, 743
remember to delimit statements within your procedure
tips for writing procedures, 853
remote data
location, 767
remote data access
about, 753
case sensitivity, 781
character set conversion limitation, 755
complete passthrough of the statement, 779
connection names, 782
connectivity problems, 781
general problems with queries, 782
internal operations, 778
introduction, 754
Lotus Notes SQL 2.0, 798
Microsoft Access, 795
Microsoft Excel, 797
Microsoft FoxPro, 797
partial passthrough of the statement, 779
passthrough mode, 774
performance limitations, 754
PowerBuilder DataWindows, 756
queries blocked on themselves, 782
query normalization, 778
query parsing, 778
query preprocessing, 778
remote servers, 757
server capabilities, 778
SQL Remote unsupported, 781
troubleshooting, 781
unsupported features, 781
remote databases
glossary definition, 891
REMOTE DBA authority
glossary definition, 892
remote IDs
glossary definition, 892
remote procedure calls
about, 775
remote procedures
calls, 775
creating, 775
data types, 776
deleting, 776
remote servers
about, 757
Advantage Database Server, 789
altering, 759
ASE JDBC, 800
ASE ODBC, 786
classes, 783, 784
creating, 757
creating from Sybase Central, 758
DB2, 789
deleting, 759
external logins, 765
JDBC, 799
JDBC limitations, 799
Index

listing properties, 761
Lotus Notes SQL 2.0, 798
Microsoft Access, 795
Microsoft Excel, 797
Microsoft FoxPro, 797
MySQL, 794
ODBC, 796
Oracle, 791
sending native statements, 774
SQL Anywhere JDBC, 799
SQL Anywhere ODBC, 786
SQL Server, 793
transaction management, 777
UltraLite, 786
remote table mappings
about, 754
remote tables
about, 754
accessing, 753
listing, 760
listing columns, 770
remote transaction management
overview, 777
removing statistics
Performance Monitor, 225
renaming columns in query results
about, 298
REORGANIZE TABLE statement
unavailable with snapshot isolation, 126
reorganizing tables
reducing table fragmentation, 246
repeatable reads
improving concurrency, 158
introduction, 124
SELECT statements, 150
setting for ODBC, 135
tutorial, 164
types of inconsistency, 132
REPEATABLEREAD table hint
(see also repeatable reads)
replace expensive triggers
performance improvement tips, 249
replication
concurrency issues, 178
glossary definition, 892
rebuilding databases, 740
rebuilding databases involved in synchronization, 740
Replication Agent
glossary definition, 892
replication frequency
glossary definition, 892
replication messages
glossary definition, 892
Replication Server
glossary definition, 892
request log
about, 217
security, 217
request logging
about, 217
request statistics
list, 236
request trace analysis
about, 214
performing, 214
request-level logging (see request logging)
requests
reducing number of, 249
Requests Active statistic
description, 236
Requests Exchange statistic
description, 236
Requests GET DATA/sec statistic
description, 237
Requests statistic
description, 236
requests tab
Index Consultant recommendations, 197
Requests Unscheduled statistic
description, 236
Requests Waiting statistic
description, 233
requirements
SQL Anywhere debugger, 858
reserved words
remote servers, 781
resetting
procedure profiling in Sybase Central, 191
resetting procedure profiling
about, 191
resetting profiling using sa_server_option
about, 220
RESIGNAL statement
about, 849
resolving
relationships, 19
resolving relationships
database design, 19
RESTRICT action
about, 112
restrictions
about, 296
changing manual views to immediate views, 72
remote data access, 781
restrictions on transaction management
about, 777
restrictions when managing materialized views
about, 71
result sets
executing a query more than once, 318
limiting the number of rows, 363
multiple, 839
permissions, 837
procedures, 812, 837
remote procedures, 775
saving to a file, 751
Transact-SQL, 654
troubleshooting, 318
variable, 839
retrieving
index-only retrievals, 569
retrieving information about materialized views
about, 66
retrieving profiling information using system procedures
about, 221
retrieving the first few rows of a query
about, 363
RETURN statement
about, 836
return values
procedures, 656
returning a value using the RETURN statement
about, 836
returning multiple result sets from procedures
about, 839
returning procedure results in parameters
about, 811
returning procedure results in result sets
about, 812
returning result sets from procedures
about, 837
returning result sets from Transact-SQL procedures
about, 654
returning results as procedure parameters
about, 836
returning results from procedures
about, 836
returning variable result sets from procedures
about, 839
review the order of columns in tables
performance improvement tips, 243
REVOKE statement
concurrency, 179
Transact-SQL, 640
rewrite optimization
about, 528
rewriting subqueries as EXISTS predicates
about, 537
right outer joins
about, 386
RL plan item
abbreviations in the plan, 589
glossary definition, 892
role names
about, 407
glossary definition, 893
roles
ASE, 638
defined, 7
glossary definition, 892
rollback logs
data recovery, 703
glossary definition, 893
savepoints, 123
ROLLBACK statement
about, 511
compound statements, 831
procedures and triggers, 852
transactions, 121
triggers, 652
rolling back
transactions, 121
ROLLUP clause
about, 444
using as a shortcut to GROUPING SETS, 443
ROLLUP operation
understanding GROUP BY, 356
row limit count
item in execution plans, 600
row locks
about, 142, 144
exclusive, 145
intent, 145
read, 144
write, 145
row numbering functions with windows
about, 474, 482
row versions
about, 127
row-level triggers
glossary definition, 893
ROW_NUMBER function
usage, 482
RowConstructor algorithm (ROWS)
about, 588
RowConstructor plan item
abbreviations in the plan, 589
ROWID function
used by the RowIdScan method, 571
ROWID plan item
abbreviations in the plan, 589
RowIdScan method (ROWID)
about, 571
RowIdScan plan item
abbreviations in the plan, 589
RowLimit algorithm (RL)
about, 588
RowLimit plan item
abbreviations in the plan, 589
RowReplicate algorithm (RR)
about, 582
RowReplicate plan item
abbreviations in the plan, 589
rows
copying with INSERT, 515
deleting, 520
impact of deleting, 616
intent locks, 145
locks, 144
selecting, 305
ROWS clause
using, 453
ROWS plan item
abbreviations in the plan, 589
RowsReturned
Node Statistics field descriptions, 611
statistic in access plans, 595
RR plan item
abbreviations in the plan, 589
RT plan item
abbreviations in the plan, 589
RU plan item
abbreviations in the plan, 589
rules
Transact-SQL, 637
rules describing the operation of key joins
about, 416
running
command files, 750
SQL scripts, 750
running SQL command files
about, 750
RunTime
Node Statistics field descriptions, 611
statistic in access plans, 595
S
sa ANSI standard packages system procedure
SQL Flagger usage, 629
SA_DEBUG group
debugger, 858
sa dependent views system procedure
using, 57
sa locks system procedure
using, 143
sa migrate system procedure
using, 747
sa migrate create fks system procedure
using, 748
sa migrate create remote fks list system procedure
using, 748
sa migrate create remote table list system procedure
using, 748
sa migrate create tables system procedure
using, 748
sa migrate data system procedure
using, 748
sa migrate drop proxy tables system procedure
using, 748
sa procedure profile system procedure
obtaining in-depth profiling information, 221
sa procedure profile summary system procedure
obtaining summary profiling information, 221
sa report deadlocks system procedure
using, 139
sa server option system procedure
disabling procedure profiling, 220
enabling procedure profiling, 219
resetting procedure profiling, 220
setting filters on procedure profiling, 219
SA_SQL_TXN_READONLY_STATEMENT_SNAPSHOT
    isolation level, 135
SA_SQL_TXN_SNAPSHOT
    isolation level, 135
SA_SQL_TXN_STATEMENT_SNAPSHOT
    isolation level, 135
sajdbc server class
    about, 799
sample database
    schema for demo.db, 375
samples-dir
    documentation usage, xiii
saodbc server class
    about, 786
saplan files
    about, 602
sargable predicates
    about, 548
SATMP environment variable
    temporary file location, 244
savepoints
    naming, 123
    nesting, 123
    procedures and triggers, 852
    within transactions, 123
saving
    result sets, 751
    transaction results, 121
saving transaction results
    about, 121
scalar aggregate functions
    defined, 358
scalar aggregates
    about, 351
scattered reads
    performance, 251
schedules
    effects of serializability, 157
    effects of unserializable, 157
    serializable, 157
    serializable versus early release of locks, 155
    scheduling of transactions
    about, 157
schema
    locks, 144
schema locks
    about, 144
    exclusive, 144
    shared, 144
schemas
    exporting, 742
    glossary definition, 893
scopes
    diagnostic tracing, 203
scoring
    full text search, 331
script versions
    glossary definition, 893
script-based uploads
    glossary definition, 893
scripts
    about creating command files, 750
    creating command files, 750
    glossary definition, 893
    loading command files, 751
    running in Interactive SQL, 750
search conditions
    date comparisons, 314
    example with NOT keyword, 306
    GROUP BY clause, 321
    pattern matching, 308
    subqueries, 486
    usage, 305
searching
    full text search, 323
searching for unindexed terms
    about, 330
second normal form
    testing, 18
secured features
    glossary definition, 893
security
    hiding objects, 856
    importing from, and exporting to, client computers, 734
    procedures, 807
    request log, 217
select list
    about, 295
    aliases, 298
    calculated columns, 299
<table>
<thead>
<tr>
<th>Column Order Impacts Order in Results, 297</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXCEPT Statements, 365</td>
</tr>
<tr>
<td>Execution Plans, 596</td>
</tr>
<tr>
<td>INTERSECT Statements, 365</td>
</tr>
<tr>
<td>UNION Statement, 366</td>
</tr>
<tr>
<td>UNION Statements, 365</td>
</tr>
<tr>
<td>SELECT Statement</td>
</tr>
<tr>
<td>About, 292</td>
</tr>
<tr>
<td>Aliases, 298</td>
</tr>
<tr>
<td>Character Data, 310</td>
</tr>
<tr>
<td>Column Headings, 298</td>
</tr>
<tr>
<td>Column Order, 297</td>
</tr>
<tr>
<td>Cursors, 841</td>
</tr>
<tr>
<td>INSERT FROM, 513</td>
</tr>
<tr>
<td>INTO Clause, 837</td>
</tr>
<tr>
<td>Keys and Query Access, 254</td>
</tr>
<tr>
<td>Restrictions in Regular Views, 57</td>
</tr>
<tr>
<td>Specifying Rows, 305</td>
</tr>
<tr>
<td>Strings in Display, 298</td>
</tr>
<tr>
<td>Subqueries, 486</td>
</tr>
<tr>
<td>Transact-SQL Compatibility, 648</td>
</tr>
<tr>
<td>Variables, 649</td>
</tr>
<tr>
<td>Select-List (See Select List)</td>
</tr>
<tr>
<td>About, 295</td>
</tr>
<tr>
<td>Selecting All Columns from a Table</td>
</tr>
<tr>
<td>About, 295</td>
</tr>
<tr>
<td>Selecting Data Using Subqueries</td>
</tr>
<tr>
<td>Overview, 485</td>
</tr>
<tr>
<td>Selecting Specific Columns from a Table</td>
</tr>
<tr>
<td>About, 296</td>
</tr>
<tr>
<td>Selectivity</td>
</tr>
<tr>
<td>Item in Execution Plans, 598</td>
</tr>
<tr>
<td>Reading in Execution Plans, 607</td>
</tr>
<tr>
<td>Reading the Execution Plan, 603</td>
</tr>
<tr>
<td>Selectivity Estimates</td>
</tr>
<tr>
<td>Reading in Execution Plans, 607</td>
</tr>
<tr>
<td>Using a Partial Index Scan, 623</td>
</tr>
<tr>
<td>Selectivity in the Plan</td>
</tr>
<tr>
<td>About, 607</td>
</tr>
<tr>
<td>Selectivity Statistics</td>
</tr>
<tr>
<td>About, 603</td>
</tr>
<tr>
<td>Self-Joins</td>
</tr>
<tr>
<td>About, 393</td>
</tr>
<tr>
<td>Self_Recursion Option</td>
</tr>
<tr>
<td>Adaptive Server Enterprise, 652</td>
</tr>
<tr>
<td>Semantic Query Transformations</td>
</tr>
<tr>
<td>About, 528</td>
</tr>
<tr>
<td>Semantic Transformations</td>
</tr>
<tr>
<td>About, 528</td>
</tr>
<tr>
<td>Semicolons</td>
</tr>
<tr>
<td>Command Delimiter, 853</td>
</tr>
<tr>
<td>Sending Native Statements to Remote Servers</td>
</tr>
<tr>
<td>About, 774</td>
</tr>
<tr>
<td>SendingTracingTo</td>
</tr>
<tr>
<td>Tracing Configuration, 201</td>
</tr>
<tr>
<td>Separate Primary and Foreign Key Indexes</td>
</tr>
<tr>
<td>Performance Improvement Tips, 254</td>
</tr>
<tr>
<td>Seq Plan Item</td>
</tr>
<tr>
<td>Abbreviations in the Plan, 589</td>
</tr>
<tr>
<td>Sequential Scans</td>
</tr>
<tr>
<td>Disk Allocation and Performance, 615</td>
</tr>
<tr>
<td>Sequential Table Scans</td>
</tr>
<tr>
<td>About, 570</td>
</tr>
<tr>
<td>Disk Allocation and Performance, 615</td>
</tr>
<tr>
<td>Serializable</td>
</tr>
<tr>
<td>Improving Concurrency, 158</td>
</tr>
<tr>
<td>Introduction, 124</td>
</tr>
<tr>
<td>Schedules, 157</td>
</tr>
<tr>
<td>SELECT Statements, 150</td>
</tr>
<tr>
<td>Setting for ODBC, 135</td>
</tr>
<tr>
<td>Types of Inconsistency, 132</td>
</tr>
<tr>
<td>Serializable Schedules</td>
</tr>
<tr>
<td>About, 157</td>
</tr>
<tr>
<td>Effect Of, 157</td>
</tr>
<tr>
<td>Versus Early Release of Locks, 155</td>
</tr>
<tr>
<td>SERIALIZABLE Table Hint</td>
</tr>
<tr>
<td>(See Also Serializable)</td>
</tr>
<tr>
<td>Server Capabilities</td>
</tr>
<tr>
<td>Remote Data Access, 778</td>
</tr>
<tr>
<td>Server Classes</td>
</tr>
<tr>
<td>About, 755</td>
</tr>
<tr>
<td>Advantage Database Server, 789</td>
</tr>
<tr>
<td>ASEJDBC, 800</td>
</tr>
<tr>
<td>ASEODBC, 786</td>
</tr>
<tr>
<td>Db2odbc, 789, 795</td>
</tr>
<tr>
<td>Defining, 754</td>
</tr>
<tr>
<td>MSODBC, 793</td>
</tr>
<tr>
<td>MySQL, 794</td>
</tr>
<tr>
<td>ODBC, 785, 796</td>
</tr>
<tr>
<td>ORAODBC, 791</td>
</tr>
<tr>
<td>Sajdbc, 799</td>
</tr>
<tr>
<td>Saodbc, 786</td>
</tr>
<tr>
<td>Ulodbc, 786</td>
</tr>
<tr>
<td>Server Classes for Remote Data Access</td>
</tr>
<tr>
<td>About, 783</td>
</tr>
<tr>
<td>Server Management Requests</td>
</tr>
</tbody>
</table>
SOAP services
  debugging, 858
sort
  Sort algorithm, 582
Sort algorithm (Sort)
  about, 582
sort order
  comparisons, 306
  ORDER BY clause, 362
Sort plan item
  abbreviations in the plan, 589
SortedGroupBySets algorithm (GrBySSets)
  about, 580
SortedGroupBySets plan item
  abbreviations in the plan, 589
sorting
  query execution algorithms, 582
  query results, 316
Sort algorithm, 582
SortTopN algorithm, 583
  with an index, 249
sorting algorithms
  Sort, 582
  SortTopN, 583
SortTopN algorithm (SrtN)
  about, 583
SortTopN plan item
  abbreviations in the plan, 589
SOUNDEX function
  about, 315
source code
  setting breakpoints, 863
sp_addgroup system procedure
  Transact-SQL, 639
sp_addlogin system procedure
  support, 636
  Transact-SQL, 639
sp_adduser system procedure
  Transact-SQL, 639
sp_bindefault procedure
  Transact-SQL, 637
sp_bindrule procedure
  Transact-SQL, 637
sp_changegroup system procedure
  Transact-SQL, 639
sp_dboption system procedure
  Transact-SQL, 642
sp_dropgroup system procedure
  Transact-SQL, 639
sp_droplogin system procedure
  Transact-SQL, 639
sp_dropuser system procedure
  Transact-SQL, 639
sp_remote_columns system procedure
  using, 770
sp_remote_tables system procedure
  using, 760
sp_servercaps system procedure
  using, 761
special IDENTITY column
  about, 645
special Transact-SQL timestamp column and data type
  about, 644
specifying proxy table locations
  about, 767
SQL
  differences from other SQL dialects, 631
  entering, 293
  glossary definition, 894
SQL Anywhere
  DB2 data type conversions, 790
  differences from other SQL dialects, 631
  documentation, x
  glossary definition, 894
  Microsoft SQL Server data type conversions, 793
  ODBC and ASE data type conversions, 787
  Oracle data type conversions, 792
  server class, 786
SQL Anywhere debugger (see debugger)
SQL Anywhere SQL features
  about, 631
SQL command files
  about, 750
  creating, 750
  executing, 750
  opening in Interactive SQL, 750
  running, 750
  writing output, 751
SQL Flagger
  about, 629
invoking, 629
standards and compatibility, 630
testing SQL compliance with UltraLite SQL, 629
SQL preprocessor
  SQL Flagger usage, 629
SQL queries
  about, 293
SQL Remote
  glossary definition, 895
  remote data access, 781
SQL Server
  data type conversions, 793
  remote access, 793
SQL standards
  about, 630
  compliance, 627
  GROUP BY clause, 371
  non-ANSI joins, 380
SQL statements
  disallowed in snapshot isolation transactions, 126
  executing in Interactive SQL, 750
  glossary definition, 895
  writing compatible SQL statements, 647
SQL statements for implementing integrity constraints
  about, 96
SQL-based synchronization
  glossary definition, 894
SQL/1999
  testing compliance of SQL statements, 629
SQL/2003
  testing compliance of SQL statements, 629
SQL/2003 compliance
  (see also SQL standards)
SQL/XML
  about, 672
sql_flagger_error_level option
  SQL Flagger usage, 629
sql_flagger_warning_level option
  SQL Flagger usage, 629
SQL_TXN_ISOLATION
  about, 135
SQL_TXN_READ_COMMITTED
  isolation level, 135
SQL_TXN_READ_UNCOMMITTED
  isolation level, 135
SQL_TXN_REPEATABLE_READ
  isolation level, 135
SQL_TXN_SERIALIZABLE
  isolation level, 135
SQLCA.lock
  selecting isolation levels, 135
  versus isolation levels, 132
SQLCODE variable
  introduction, 844
SQLFLAGGER function
  SQL Flagger usage, 629
SQLSetConnectOption
  about, 135
SQLSTATE variable
  introduction, 844
SQLX (see SQL/XML)
SrtN plan item
  abbreviations in the plan, 589
  stale data
    refreshing in manual views, 75
  stale manual views
    about, 66
staleness and refreshing text indexes
  about, 333
standard deviation functions
  OLAP, 468
standard output
  redirecting to files, 728
standards
  (see also SQL standards)
  standards and compatibility
    (see also SQL standards)
star joins
  about, 394
starting
  transactions, 121
Statement Cache Hits statistic
  description, 236
Statement Cache Misses statistic
  description, 236
Statement Prepares statistic
  description, 236
statement-level triggers
  about, 651
  glossary definition, 895
statement-snapshot isolation level
  SELECT statement isolation level
  using, 156
statements
  compound, 831
  detecting slow statements, 270
Index Lookups/sec, 233  
list, 227  
Lock Count, 236  
Main Heap Bytes, 237  
Map physical memory/sec, 233  
Mem Pages Carver, 233  
Mem Pages Lock Table, 235  
Mem Pages Locked Heap, 235  
Mem Pages Main Heap, 235  
Mem Pages Map Pages, 235  
Mem Pages Pinned Cursor, 233  
Mem Pages Procedure Definitions, 235  
Mem Pages Query Processing, 233  
Mem Pages Relocatable, 235  
Mem Pages Relocations/sec, 235  
Mem Pages Rollback Log, 235  
Mem Pages Trigger Definitions, 235  
Mem Pages View Definitions, 235  
memory pages, 235  
miscellaneous, 237  
monitoring, 223  
monitoring performance, 227  
monitoring using the Performance Monitor, 224  
Multi-Page Allocations, 233  
Number of Grant Fails, 233  
Number of Grant Requests, 233  
Number of Grant Waits, 233  
Pages Granted, 233  
ProcCall algorithm, 587  
procedures, 587  
Query Low memory strategies, 237  
Query Plan cache pages, 237  
Query Rows materialized/sec, 237  
Recovery I/O Estimate, 228  
Recovery Urgency, 228  
removing from the Performance Monitor, 225  
Requests, 236  
Requests Active, 236  
Requests Exchange, 236  
Requests GET DATA/sec, 237  
Requests Unscheduled, 236  
Requests Waiting, 233  
Snapshot Count, 236  
Statement Cache Hits, 236  
Statement Cache Misses, 236  
Statement Prepares, 236  
Statements, 236  
Temporary Table Pages, 237  
Transaction Commits, 236  
Transaction Rollbacks, 236  
Translog Group Commits, 232  
updating column statistics, 543  
Version Store Pages, 237  
statistics are present and correct  
about, 545  
STDDEV function  
equivalent mathematical formula, 483  
see STDDEV_SAMP function, 469  
STDDEV_POP function  
equivalent mathematical formula, 483  
ext, 469  
usage, 469  
STDDEV_SAMP function  
equivalent mathematical formula, 483  
ext, 471  
usage, 471  
steps in optimization  
about, 526  
stoplists  
about, 341  
behavior when searching for stoplist terms, 330  
full text search, 341  
stored procedure language  
overview, 651  
stored procedures  
caching statements, 562  
common table expressions in, 423  
compared to batches, 827  
debugging, 860  
generating and reviewing profiling results, 190  
glossary definition, 895  
Transact-SQL stored procedure overview, 651  
using in the FROM clause, 304  
storing BLOBs in the database  
about, 25  
storing values  
common table expressions, 424  
storing XML documents in relational databases  
about, 663  
strategic sorting of query results  
performance improvement tips, 249  
string and number defaults  
about, 102  
string literal  
glossary definition, 895  
strings
| matching | 308 |
| quotation marks | 310, 311 |
| searching the database using full text search | 323 |
| structure of procedures and triggers | 833 |
| subqueries | 485, 486 |
| ALL test | 499 |
| ANY operator | 498 |
| ANY test | 497 |
| caching of | 583 |
| categorization of | 486 |
| comparison operators | 503 |
| comparison test | 495 |
| converting to joins | 502 |
| correlated | 489 |
| correlated subqueries | 489 |
| existence test | 495, 500 |
| glossary definition | 895 |
| GROUP BY | 493 |
| HAVING clause | 493 |
| IN keyword | 308 |
| introduction | 486 |
| multiple row subqueries | 486 |
| nested | 489 |
| or joins | 490 |
| outer references | 493 |
| quantified comparison test | 495 |
| rewriting as EXISTS predicates | 537 |
| rewriting as joins | 502 |
| row group selection | 493 |
| row selection | 492 |
| set membership test | 495, 496 |
| single row subqueries | 486 |
| types of operators | 495 |
| un-nesting | 530 |
| WHERE clause | 492, 502 |
| subqueries and joins | 502 |
| subqueries in the HAVING clause | 493 |
| subquery and function caching | 583 |
| subquery comparison test | 495 |
| subquery set membership test | 496 |
| subquery tests | 495 |

**Subquery Transformations during Optimization**

- about, 528

**Subscriptions**

- glossary definition, 895

**Substituting a Value for NULL**

- about, 313

**Subtotaling Results**

- CUBE clause, 445
- ROLLUP clause, 444
- WITH CUBE clause, 447
- WITH ROLLUP clause, 445

**Subtransactions**

- procedures and triggers, 852
- savepoints, 123

**SUM Function**

- equivalent mathematical formula, 483
- usage, 457

**Summarizing Data**

- about, 319

**Summarizing Query Results Using Aggregate Functions**

- about, 350

**Summarizing, Grouping and Sorting Query Results**

- about, 349

**Summary Tab**

- Index Consultant recommendations, 196

**Summary Values**

- about, 350, 355

**Supply Explicit Selectivity Estimates Sparingly**

- performance improvement tips, 250

**Support**

- newsgroups, xv

**Surrogate Rows**

- about, 153

**Swap Space**

- database cache, 257

**Sybase Central**

- altering tables, 40
- altering text configuration objects, 344
- altering text indexes, 338
- altering views, 61
- column constraints, 105
- column defaults, 98
- copying tables, 51
- creating indexes, 89
- creating regular views, 60
- creating tables, 38
- creating temporary tables, 53
creating text configuration objects, 341
creating text indexes, 335
disabling materialized views, 80
disabling optimizer use of a materialized view, 82
disabling views, 63
displaying system object contents, 36
displaying system objects, 36
dropping materialized views, 85
dropping tables, 42
dropping views, 62
enabling materialized views, 81
enabling optimizer use of a materialized view, 82
enabling views, 64
glossary definition, 896
managing foreign keys, 46
managing primary keys, 44
profiling applications, 188
rebuilding databases, 741
refreshing text indexes, 336
table constraints, 105
translating procedures, 653
unloading databases, 726
using the application profiling wizard, 189
validating indexes, 90
symbols
  string comparisons, 309
synchronization
  glossary definition, 896
  rebuilding databases, 740
syntax-independent optimization
  about, 541
SYS
  glossary definition, 896
SYSCOLSTAT
  system view, updating column statistics, 543
SYSCOLUMNS
  Transact-SQL name conflicts, 642
SYSINDEXES
  Transact-SQL name conflicts, 642
SYSSERVER
  system view, remote servers, 757
SYSTAB
  compatibility view information, 64
system administrator
  ASE, 638
system catalog
  ASE, 637
system failures
  transactions, 511
system functions
  tsequal, 645
system objects
  glossary definition, 896
  querying for a list of system objects by owner, 37
  viewing contents, 36
system procedures
  generating and reviewing profiling results, 190
  inlining as part of query transformation, 540
  procedure profiling using system procedures, 219
system security officer
  ASE, 638
system tables
  ASE, 637
  glossary definition, 896
  information about referential integrity, 117
  owner, 638
  querying for a list of system tables by owner, 37
  Transact-SQL name conflicts, 642
  viewing contents, 36
views, 64
system triggers
  enforcing referential integrity, 112
  generating and reviewing profiling results, 190
  implementing referential integrity actions, 113
system views
  glossary definition, 896
  indexes, 92
  information about referential integrity, 117
  querying for a list of views tables by owner, 37
SYSVIEWS
  consolidated view information, 64

T
  table access algorithms
    about, 568
table access methods
  HashTableScan, 571
  IndexOnlyScan, 569
  IndexScan, 568
  MultipleIndexScan, 569
  ParallelIndexScan, 570
  ParallelTableScan, 571
  RowIdScan, 571
  TableScan, 570
table and page sizes
about, 570
TableScan plan item
abbreviations in the plan, 589
technical support
newsgroups, xv
TEMP environment variable
temporary file location, 244
temporary files
work tables, 243
temporary procedures
creating, 809
Temporary Table Pages statistic
description, 237
temporary tables
about, 52
benefits of non-transactional, 52
considerations when referencing from within procedures, 53
creating, 38, 53
glossary definition, 896
importing data, 720
indexes, 620
local and global, 52
making non-transactional, 52
merging table structures, 720
Transact-SQL compatibility, 648
work tables in query processing, 260
term and phrase searching
full text search, 325
term breaker
full text search, 341
term breaker algorithm
full text search, 341
term length
setting term lengths for text indexes, 341
term lengths
full text search, 341
TermBreaker algorithm (Termbreak)
about, 588
terms
searching the database using full text search, 323
terms, full text searching
about, 325
testing
database normalization, 15
database verification, 22
testing a column for NULL
about, 312
testing set membership with IN conditions
about, 496
text configuration objects
about, 339
creating, 340
default_char, 339
default_nchar, 339
determining if used by text indexes, 339
text indexes
about, 332
altering, 337
cannot change the text configuration object, 337
choosing a refresh type for a text index, 333
creating, 333
determining the text configuration object used, 332
full text search, 332
impact of database options on creating and refreshing, 345
not allowed on views or temporary tables, 332
querying views, 330
refreshing, 333
renaming, 338
require storage space, 323
staleness and refreshing, 333
text plans
reading execution plans, 600
third normal form
testing, 19
thread safety
user-defined functions, 814
threads
deadlock when none available, 139
time-saving strategies
importing data, 705
time_format option
impact on text indexes, 345
times
procedures and triggers, 853
TIMESTAMP data type
Transact-SQL, 644
timestamp_format option
impact on text indexes, 345
timing utilities
about, 222
tips
improving performance, 239
tips for writing procedures
about, 853
check if you need to change the command delimiter, 853
remember to delimit statements within your procedure, 853
specifying dates and times in procedures, 853
use fully-qualified names for tables in procedures, 853
verifying that procedure input arguments are passed correctly, 854
TMP environment variable
temporary file location, 244
TMPDIR environment variable
temporary file location, 244
TO CLIENT FILE clause
importing from, and exporting to, client computers, 733
TO VARIABLE clause
importing from, and exporting to, client computers, 734
tools
exporting data, 722
importing data, 705
rebuilding databases, 741
reloading data, 741
unloading data, 722
TOP clause
about, 363
top performance tips
list of, 239
topics
graphic icons, xiv
total benefits
Index Consultant recommendations, 196
total cost benefit
Index Consultant recommendations, 197
TRACEBACK function
about, 845
tracing
(see also diagnostic tracing)
about, 199
application profiling using database tracing, 199
tracking database, 199
tracing data
about, 199
not unloaded as part of an unload operation, 199
tracking database
about, 199
creating internally vs. externally, 200
not unloaded as part of an unload operation, 199
tracing session
about, 199
data, 199
tracing session data
about, 199
trailing blanks
comparisons, 306
creating databases, 642
Transact-SQL, 642
Transact-SQL
about, 633
batches, 652
creating databases, 641
IDENTITY column, 645
joins, 650
NULL, 647
NULL values and joins, 392
outer join limitations, 391
outer joins, 390
outer joins and views, 392
overview, 633
procedures, 651
result sets, 654
timestamp column, 644
trailing blanks, 642
triggers, 651
using WITH ROLLUP, 445
variables, 655
writing compatible SQL statements, 647
writing portable SQL, 647
Transact-SQL batch overview
about, 652
Transact-SQL compatibility
about, 633
databases, 643
SELECT statement, 648, 649
setting database options, 642
Transact-SQL procedure language overview
about, 651
Transact-SQL stored procedure overview
about, 651
Transact-SQL trigger overview
about, 651
transaction blocking
about, 138
Transaction Commits statistic
description, 236
transaction locks
duration, 142

transaction log
data recovery, 703
dbmlsync, 740
glossary definition, 897
performance improvement tips, 252
performance tip, 239
replication, 740

transaction log mirror
glossary definition, 897

transaction management and remote data
about, 777

transaction processing
data recovery, 511
effects of scheduling, 157
performance, 123
scheduling, 157
serializable scheduling, 157

Transaction Rollbacks statistic
description, 236

transaction scheduling
effects of, 157

transactional integrity
glossary definition, 897

transactions
about, 120
beginning in snapshot isolation, 128
blocking, 138
blocking and deadlock, 138
blocking example, 167
changing isolation levels, 136
completing, 121
concurrency, 122
data modification, 510
data recovery, 511
deadlock, 138
glossary definition, 896
interference between, 138, 167
multiple, 122
procedures and triggers, 852
remote data access, 777
restrictions on management, 777
savepoints, 123
starting, 121
sub-transactions and savepoints, 123
typical isolation levels, 157
using, 120

transactions and data modification
about, 510

transactions and data recovery
about, 511

transactions and isolation levels
about, 119

transactions and savepoints in procedures and triggers
about, 852

transactions processing
blocking, 138
blocking example, 167

transferring data
about, 701
conversion errors, 718
import wizard, 705
importing, 704
importing and exporting, 702
INPUT statement, 707
INSERT statement, 711
LOAD TABLE statement, 709
MERGE statement, 712
OUTPUT statement, 724
performance, 702
performance tips, 705
UNLOAD statement, 725
UNLOAD TABLE statement, 724

transferring data into and out of databases
about, 702

transformations
rewrite optimization, 528

Translog Group Commits statistic
description, 232

transmission rules
glossary definition, 897

tranest
about, 242

trigger conditions
order in which triggers fire, 824

triggers
about, 805
AFTER triggers, 818
altering using SQL, 823
altering using Sybase Central, 822
BEFORE triggers, 818
benefits of, 807
command delimiter, 853
create trigger wizard, 820
creating, 819
understanding group by
about, 355
understanding Index Consultant recommendations
about, 195
union list
item in execution plans, 599
UNION statement
combining queries, 365
NULL, 368
rules, 366
UnionAll algorithm (UA)
about, 582
UnionAll plan item
abbreviations in the plan, 589
unions
query execution algorithms, 582
unique constraints
about, 105
generated indexes, 619
glossary definition, 897
unique identifiers
tables, 43
unique keys
generating and concurrency, 178
unique results
limiting, 302
uniqueness
enforcing with an index, 623
Unix
initial cache size, 255
maximum cache size, 255
minimum cache size, 255
UNKNOWN
NULL, 313
unknown values
about, 311
unload
glossary definition, 898
unload data window
using, 728
unload database wizard
using, 726
UNLOAD statement
about, 725
UNLOAD TABLE statement
about, 724
unload tools
about, 722
unload data window, 728
unload database wizard, 726
unloading
about, 736
unloading and reloading
databases, 743
databases involved in synchronization, 740
databases not involved in synchronization, 739
unloading databases
about, 730, 736
from Sybase Central, 726
in comma-delimited format, 741
unnecessary distinct elimination
about, 529
unserializable transaction scheduling
effects of, 157
unused indexes tab
Index Consultant recommendations, 197
UPDATE conflicts
snapshot isolation, 131
UPDATE statement
errors, 115
examples, 115
locking during, 153
using, 517
updates tab
Index Consultant recommendations, 197
updating column statistics
about, 543
updating the database
overview, 509
upgrading
database file format, 737
upgrading databases
about, 737
uploads
glossary definition, 898
use an appropriate page size
performance improvement tips, 250
use appropriate data types
performance improvement tips, 251
use AUTOINCREMENT to create primary keys
performance improvement tips, 252
use fully-qualified names for tables in procedures
tips for writing procedures, 853
use indexes effectively
performance improvement tips, 253
use indexes for frequently-searched columns
about, 86
use keys effectively
performance improvement tips, 254
use the cache to improve performance
performance improvement tips, 254
use the compression features
performance improvement tips, 259
use the WITH EXPRESS CHECK option when
validating tables
performance improvement tips, 260
use work tables in query processing
performance improvement tips, 260
user defined functions (see user-defined functions)
see user-defined functions, 814
user IDs
ASE, 639
case sensitivity, 644
default, 99
user-defined data types
about, 107
CHECK constraints, 104
creating, 107
creating using SQL, 108
dropping, 109
glossary definition, 898
user-defined functions
about, 814
caching, 584
calling, 815
creating, 814
dropping, 816
execution permissions, 816
generating and reviewing profiling results, 190
inlining as part of query transformation, 539
parameters, 834
thread safety, 814
user_estimates
Optimizer Statistics field descriptions, 612
using a directory access server
about, 762
using a text index to query a view
about, 330
using aggregate functions with distinct
about, 353
using cache warming
about, 258
using CHECK constraints on columns
about, 103
using CHECK constraints on tables
about, 104
USING CLIENT FILE clause
importing from, and exporting to, client computers, 733
using clustered indexes
about, 87
using column defaults
about, 97
using comparison operators in the WHERE clause
about, 306
using compound statements
about, 831
using count (*)
about, 352
using CUBE
about, 445
using cursors in procedures and triggers
about, 841
using cursors on SELECT statements in procedures
about, 841
using domains
about, 107
using exception handlers in procedures and triggers
about, 848
using FOR XML AUTO
about, 677
using FOR XML EXPLICIT
about, 680
using FOR XML RAW
about, 676
using group by with aggregate functions
about, 358
using GROUP BY with multiple columns
about, 357
using Interactive SQL to export data
about, 723
using joins in delete, update, and insert statements
about, 380
using lists in the WHERE clause
about, 308
using OLAP
about, 437
USING parameter in the CREATE SERVER statement
about, 799
using procedures, triggers, and batches
about, 805
using proxy tables to import data
about, 718
using ranges in the WHERE clause
about, 307
using regular views
about, 57
using remote procedure calls
about, 775
using ROLLUP
about, 444
using SELECT statements in batches
about, 855
using snapshot isolation to avoid dirty reads
about, 162
using snapshot isolation to avoid non-repeatable reads
about, 168
using SQL command files
about, 750
using subqueries
about, 485
using subqueries in the WHERE clause
about, 492
using Sybase Central to translate stored procedures
about, 653
using table and column constraints
about, 103
using the dbunload utility to export data
about, 726
using the dbunload utility to rebuild databases
about, 741
using the EXECUTE IMMEDIATE statement in procedures
about, 851
using the Index Consultant
about, 194
using the INPUT statement to import data
about, 707
using the INSERT statement to import data
about, 711
using the Interactive SQL import wizard
about, 705
using the LOAD TABLE statement to import data
about, 709
using the MERGE statement
about, 712
using the migrate database wizard
about, 746
using the OUTPUT statement to export data
about, 724
using the RAISERROR action
about, 717
using the RAISERROR statement in procedures
about, 657
using the sa_migrate system procedures
about, 747
using the Unload Data window
about, 728
using the UNLOAD statement
about, 725
using the UNLOAD TABLE statement
about, 724
using the UNLOAD TABLE statement to rebuild databases
about, 742
using the WHERE clause for join conditions
about, 383
using the WITH CHECK OPTION clause
about, 58
using transaction and isolation levels
about, 119
using transactions
about, 120
USING VALUE clause
importing from, and exporting to, client computers,
733
using views with Transact-SQL outer joins
about, 392
using WITH CUBE
about, 447
using WITH ROLLUP
about, 445
using XML in the database
about, 661
UUIDs
compared to global autoincrement, 101
default column value, 101
generating, 178
V
validate
glossary definition, 898
validating
indexes, 90
tables using WITH EXPRESS CHECK, 260
XML, 688
validating indexes
about, 90
validation
column constraints, 27
XML, 688
ValuePtr parameter
about, 135
VAR_POP function
equivalent mathematical formula, 483
example, 471
usage, 471
VAR_SAMP function
equivalent mathematical formula, 483
example, 472
usage, 472
variables
assigning, 649
local, 649
SELECT statement, 649
SET statement, 649
Transact-SQL, 655
VARIANCE function
equivalent mathematical formula, 483
see VAR_SAMP function, 471
variance functions
OLAP, 468
vector aggregate functions
about, 355
defined, 358
verifying
database design, 22
verifying that procedure input arguments are passed correctly
tips for writing procedures, 854
Version Store Pages statistic
description, 237
VersionStorePages property
using, 127
view dependencies
about, 55
information in the catalog, 57
regular view status, 59
schema changes, 55
view dependency information in the catalog
about, 57
view matching
about, 553
execution plan outcomes, 596
materialized views with OUTER JOIN, 560
query execution, 550
using with snapshot isolation, 127
view matching algorithm, about, 553
view matching algorithm, execution plan outcomes, 596
view status
determining, 59
disabled, 59
invalid, 59
regular views, 59
understanding, 59
valid, 59
viewing
procedure profiling results, 192
regular view data, 64
table data, 43
viewing and accessing graphical plans
about, 610
Node Statistics field descriptions, 611
Optimizer Statistics field descriptions, 612
viewing system table data using regular views
about, 64
viewing the isolation level
about, 137
views
altering and view dependencies, 61
altering regular views using SQL, 61
altering regular views using Sybase Central, 61
altering regular views, considerations, 60
browsing data in regular views, 64
check option and regular views, 58
common table expressions, 420
copying regular views, 58
creating regular views, 60
dependencies, 55
DISABLED status for regular views, 59
disabling, 63
dropping, 62
enabling, 63
exporting, 724
FROM clause, 303
glossary definition, 898
INVALID status for regular views, 59
key joins, 414
natural joins, 405
outer joins, 389
querying using a text index, 330
regular view status, 59
SELECT statement restrictions for regular views, 57
updating regular views, 58
updating using INSTEAD OF triggers, 825
using regular views, 57
VALID status for regular views, 59
working with, 54
virtual indexes
  about, 196
  Index Consultant, 196
virtual memory
  scarce resource, 546

W
wait_for_commit option
  using, 153
waiting
  to access locked rows, 167
  to verify referential integrity, 153
waiting to access locked rows
deadlock, 138
warming
  cache, 258
warnings
  procedures and triggers, 847
Watcom-SQL
  about, 633
  writing compatible SQL statements, 647
when to create an index
  about, 619
when to use indexes
  about, 86
WHERE clause
  about, 305
  compared to HAVING, 360
  date comparisons introduction, 314
GROUP BY clause, 358
HAVING clause and, 321
joins, 383
  modifying rows in a table, 517
NULL values, 312
  pattern matching, 308
performance, 249, 548
string comparisons, 309
subqueries, 492
UPDATE statement, 518
  using with the GROUP BY clause, 355
where you can use aggregate functions
  about, 351
WHILE statement
  control statements, 830
wildcards
  LIKE search condition, 309
  pattern matching, 308
  string comparisons, 309
window (OLAP)
  glossary definition, 898
window aggregate functions
  about, 456
  list of supported functions, 451
OLAP, 456
Window algorithm (Window)
  about, 588
WINDOW clause
  using in the SELECT statement, 452
window functions
  about, 450
  aggregate, list of, 456
  ranking, list of, 474
  row numbering, 482
Window plan item
  abbreviations in the plan, 589
window ranking functions
  list of supported functions, 451
Windows
  glossary definition, 898
  initial cache size, 255
  maximum cache size, 255
  minimum cache size, 255
windows (OLAP)
  defining inline windows, 452
  order of evaluation of clauses, 452
  size, 453
  sizing using RANGE clause, 453
  sizing using ROWS clause, 453
  WINDOW clause of the SELECT statement, 452
Windows Mobile
  cache and page size considerations, 616
  glossary definition, 898
  hashexcept algorithm, 580
  intersect algorithms, 581
Windows Performance Monitor
  about, 226
  running multiple copies, 226
  starting, 226
WITH CHECK OPTION clause
about, 58
WITH clause
common table expressions, 420
RecursiveTable algorithm, 582
RecursiveUnion algorithm, 582
WITH CUBE clause
about, 447
WITH EXPRESS CHECK
performance, 260
WITH ROLLUP clause
about, 445
work tables
about, 260
glossary definition, 899
performance tips, 243
query processing, 260
working with breakpoints
about, 863
working with column defaults in Sybase Central
about, 98
working with computed columns
about, 48
working with database objects
about, 35
working with external logins
about, 765
working with indexes
about, 86
working with materialized views
about, 65
working with proxy tables
about, 767
working with remote servers
about, 757
working with table and column constraints in Sybase Central
about, 105
working with tables
about, 38
working with temporary tables
about, 52
working with views
about, 54
write locks
about, 145
WRITE_CLIENT_FILE function
importing from, and exporting to, client computers, 733
WRITECLIENTFILE authority
importing from, and exporting to, client computers, 733
writing an EXPLICIT mode query
about, 682
writing compatible queries
about, 648
writing compatible SQL statements
about, 647
writing database output to a file
about, 751
X
XML
defined, 662
exporting data as from Interactive SQL, 664
exporting data as using the DataSet object, 664
exporting relational data as, 664
importing as relational data, 665
importing using openxml, 665
importing using the DataSet object, 670
obtaining query results as XML, 672
obtaining query results as XML from relational data, 672
storing in relational databases, 663
using FOR XML AUTO, 677
using FOR XML EXPLICIT, 680
using FOR XML RAW, 676
using in SQL Anywhere databases, 661
XML and SQL Anywhere
about, 662
XML data type
using, 663
xml directive
using, 687
XMLAGG function
using, 690
XMLCONCAT function
using, 691
XMLELEMENT function
using, 691
XMLFOREST function
using, 694
XMLGEN function
using, 694
xp_read_file system procedure
    importing XML, 668
XPath
    using, 665