SQL Anywhere® Server
SQL Usage

Published: October 2006

Copyright and trademarks

Contents

About This Manual

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 manual is for all users of SQL Anywhere.

Before you begin

This manual 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 10 - Introduction* before reading this manual.

SQL Anywhere documentation

This book is part of the SQL Anywhere documentation set. This section describes the books in the documentation set and how you can use them.

The SQL Anywhere documentation

The complete SQL Anywhere documentation is available in two forms: an online form that combines all books, and as separate PDF files for each book. Both forms of the documentation contain identical information and consist of the following books:

- ♦ **SQL Anywhere 10 - Introduction** This book introduces SQL Anywhere 10—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 10 - Changes and Upgrading** This book describes new features in SQL Anywhere 10 and in previous versions of the software.
- ♦ **SQL Anywhere Server - Database Administration** This book covers material related to running, managing, and configuring SQL Anywhere databases. It describes database connections, the database server, database files, security, backup procedures, security, and replication with Replication Server, as well as administration utilities and options.
- ♦ **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.
- ♦ **SQL Anywhere Server - SQL Reference** This book provides a complete reference for the SQL language used by SQL Anywhere. It also describes the SQL Anywhere system views and procedures.
- ♦ **SQL Anywhere Server - Programming** This book describes how to build and deploy database applications using the C, C++, and Java programming languages, as well as Visual Studio .NET. Users of tools such as Visual Basic and PowerBuilder can use the programming interfaces provided by those tools.
- ♦ **SQL Anywhere 10 - Error Messages** This book provides a complete listing of SQL Anywhere error messages together with diagnostic information.
- ♦ **MobiLink - Getting Started** This manual introduces MobiLink, a session-based relational-database synchronization system. MobiLink technology allows two-way replication and is well suited to mobile computing environments.
- ♦ **MobiLink - Server Administration** This manual describes how to set up and administer MobiLink applications.
- ♦ **MobiLink - Client Administration** This manual describes how to set up, configure, and synchronize MobiLink clients. MobiLink clients can be SQL Anywhere or UltraLite databases.
- ♦ **MobiLink - Server-Initiated Synchronization** This manual 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 manual describes QAnywhere, which defines 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.
- **SQL Anywhere 10 - Context-Sensitive Help** This manual provides context-sensitive help for the Connect dialog, the Query Editor, the MobiLink Monitor, the SQL Anywhere Console utility, the Index Consultant, and Interactive SQL.
- **UltraLite - Database Management and Reference** This manual introduces the UltraLite database system for small devices.
- **UltraLite - AppForge Programming** This manual describes UltraLite for AppForge. With UltraLite for AppForge you can develop and deploy database applications to handheld, mobile, or embedded devices, running Palm OS, Symbian OS, or Windows CE.
- **UltraLite - .NET Programming** This manual describes UltraLite.NET. With UltraLite.NET you can develop and deploy database applications to computers, or handheld, mobile, or embedded devices.
- **UltraLite - M-Business Anywhere Programming** This manual 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 CE, or Windows XP.
- **UltraLite - C and C++ Programming** This manual describes UltraLite C and C++ programming interfaces. With UltraLite you can develop and deploy database applications to handheld, mobile, or embedded devices.

Documentation formats

SQL Anywhere provides documentation in the following formats:

- **Online documentation** The online documentation contains the complete SQL Anywhere documentation, including the books and the context-sensitive help for SQL Anywhere tools. The online documentation is updated with each maintenance release of the product, and is the most complete and up-to-date source of documentation.

  To access the online documentation on Windows operating systems, choose Start ► Programs ► SQL Anywhere 10 ► Online Books. You can navigate the online documentation using the HTML Help table of contents, index, and search facility in the left pane, as well as using the links and menus in the right pane.

  To access the online documentation on Unix operating systems, see the HTML documentation under your SQL Anywhere installation or on your installation CD.
- **PDF files** The complete set of SQL Anywhere books is provided as a set of Adobe Portable Document Format (pdf) files, viewable with Adobe Reader.

  On Windows, the PDF books are accessible from the online books via the PDF link at the top of each page, or from the Windows Start menu (Start ► Programs ► SQL Anywhere 10 ► Online Books - PDF Format).

On Unix, the PDF books are accessible on your installation CD.

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:

  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:

  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:

  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.

  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.

  [**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.

  [**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.

File name conventions

The documentation generally adopts Windows conventions when describing operating-system dependent tasks and features such as paths and file names. In most cases, there is a simple transformation to the syntax used on other operating systems.

- **Directories and path names** The documentation typically lists directory paths using Windows conventions, including colons for drives and backslashes as a directory separator. For example,

  ```
  MobiLink\redirector
  ```

  On Unix, Linux, and Mac OS X, you should use forward slashes instead. For example,

  ```
  MobiLink/redirector
  ```

- **Executable files** The documentation shows executable file names using Windows conventions, with the suffix *.exe*. On Unix, Linux, and Mac OS X, executable file names have no suffix. On NetWare, executable file names use the suffix *.nlm*.

  For example, on Windows, the network database server is *dbsrv10.exe*. On Unix, Linux, and Mac OS X, it is *dbsrv10*. On NetWare, it is *dbsrv10.nlm*.

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

  After installation is complete, the environment variable SQLANY10 specifies the location of the installation directory containing the SQL Anywhere components (*install-dir*). SQLANYSH10 specifies the location of the directory containing components shared by SQL Anywhere with other Sybase applications.

  For more information on the default location of *install-dir*, by operating system, see "File Locations and Installation Settings" [*SQL Anywhere Server - Database Administration*].

- **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 SQLANYSAMP10 specifies the location of the directory containing the samples (*samples-dir*). From the Windows Start menu, choosing Programs ► SQL Anywhere 10 ► Sample Applications and Projects opens a Windows Explorer window in this directory.

  For more information on the default location of *samples-dir*, by operating system, see "The samples directory" [*SQL Anywhere Server - Database Administration*].

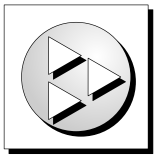
- **Environment variables** The documentation refers to setting environment variables. On Windows, environment variables are referred to using the syntax *%envvar%*. On Unix, Linux, and Mac OS X, environment variables are referred to using the syntax *$envvar* or *${envvar}*.

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

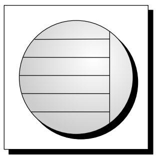
Graphic icons

The following icons are used in this documentation.

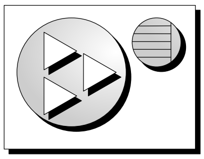
- A client application.



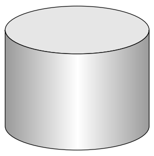
- A database server, such as SQL Anywhere.



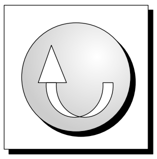
- An UltraLite application.



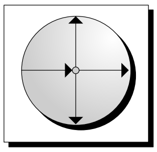
- A database. In some high-level diagrams, the icon may be used to represent both the database and the database server that manages it.



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



- A Sybase Replication Server



- A programming interface.

Finding out more and providing feedback

Finding out more

Additional information and resources, including a code exchange, are available at the iAnywhere Developer Network at http://www.ianywhere.com/developer/.

If you have questions or need help, you can post messages to the iAnywhere Solutions 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 entering **dbeng10 -v** at a command prompt.

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 Solutions 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.

Feedback

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

You can email comments and suggestions to the SQL Anywhere documentation team at iasdoc@ianywhere.com. Although we do not reply to emails sent to that address, we read all suggestions with interest.

In addition, you can provide feedback on the documentation and the software through the newsgroups listed above.

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 Your Database

Contents

About this chapter

This chapter introduces the basic concepts of relational database design and gives you step-by-step suggestions for designing your own databases. It uses the expedient technique known as conceptual data modeling, which focuses on entities and the relationships between them.

Introduction

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.

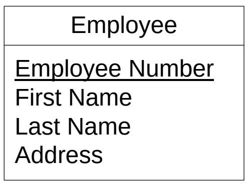
Entities

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.

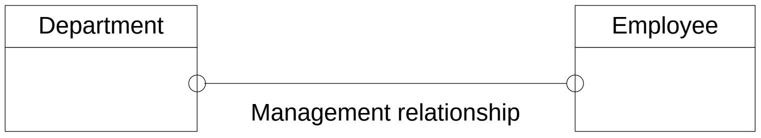
Once a relationship is built into the structure of the database, there is no provision for exceptions. There is nowhere to put a second department head. Duplicating the department entry would involve duplicating the department ID, which is the identifier. Duplicate identifiers are not allowed.

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.

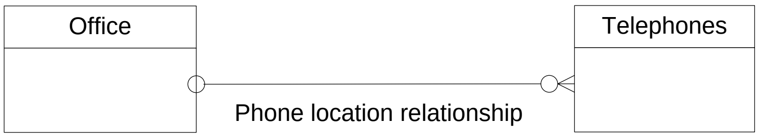
Cardinality of relationships

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

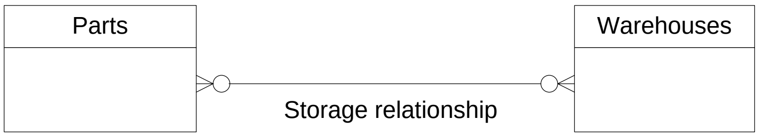
- **One-to-one relationships** You depict a relationship by drawing a line between two entities. The line may have other markings on it such as the two little circles shown below. Later sections explain the purpose of these marks. In the following diagram, one employee manages one department.



- **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.



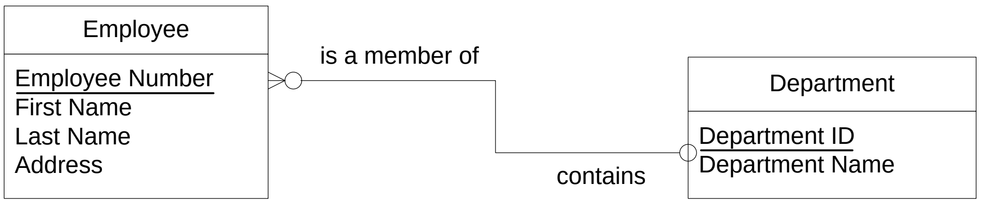
- **Many-to-many relationships** In this case, draw multiple lines for the connections to both entities. This means that one warehouse can hold many different parts, and one type of part can be stored at many warehouses.



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.



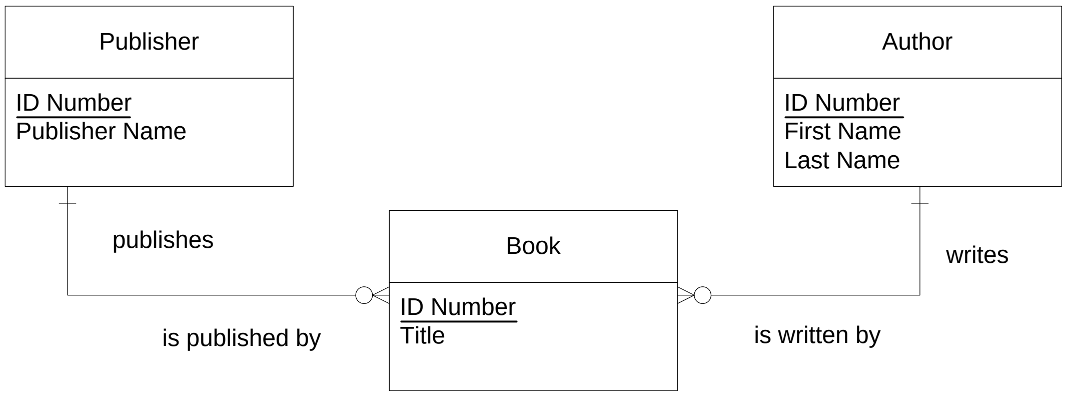
Roles are very important because they afford you a convenient and effective means of verifying your work.

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.



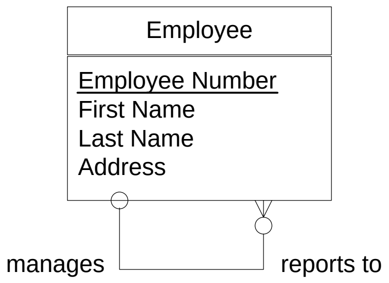
This diagram corresponds to the following four statements.

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.



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.

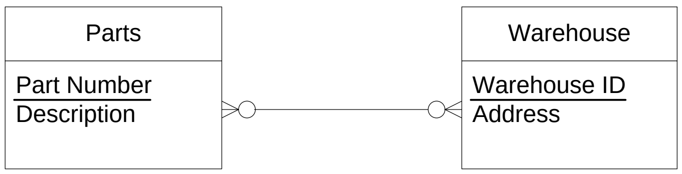
Notice that in the case of this relation, it is essential that the relation be optional in both directions. Some employees are not managers. Similarly, at least one employee should head the organization and hence report to no one.

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.

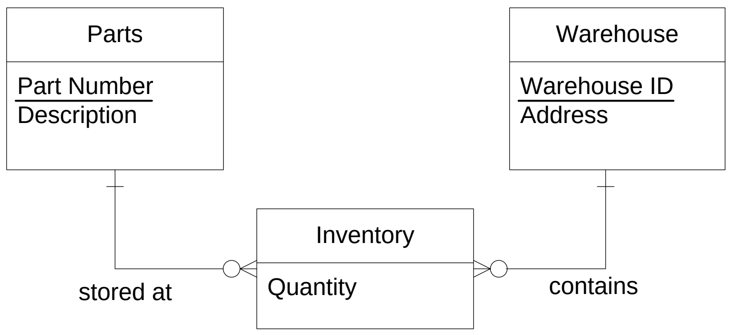
Changing many-to-many relationships into entities

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.



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:



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. The triangles that appear between the circles and the multiple lines that join the two new relationships to the new Inventory entity denote the dependencies.

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, but the triangles denote this dependence more clearly.

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 13
- "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 29.

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 *maintain 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:

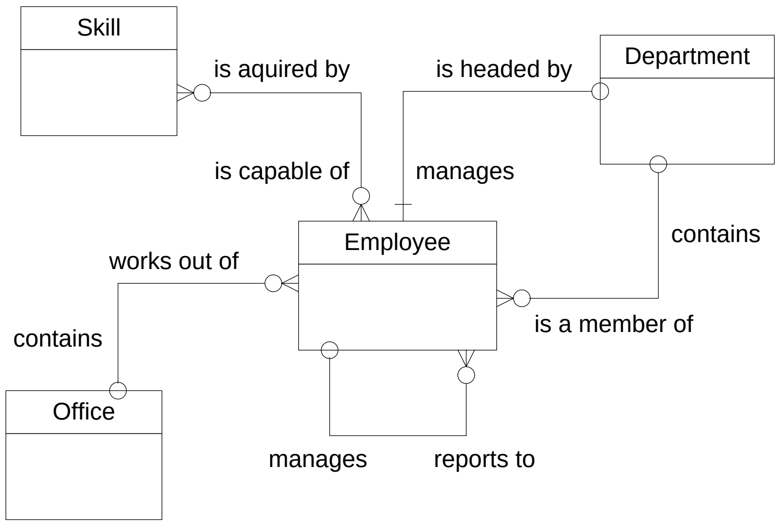
- Hire employees.
- Terminate employees.
- Maintain personal employee information.
- Maintain information on skills required for the company.
- Maintain information on which employees have which skills.
- Maintain information on departments.
- Maintain information on offices.

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).

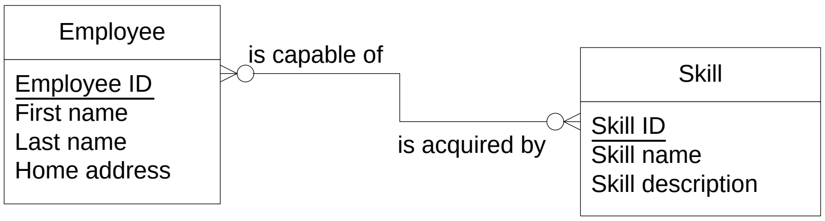
6. List the data, if any, that applies to each relationship.

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.

Employee	**Skill**	**Expert In**
Employee ID	Skill ID	Skill level
Employee first name	Skill name	Date skill was acquired
Employee last name	Description of skill	
Employee department		
Employee office		
Employee address		

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



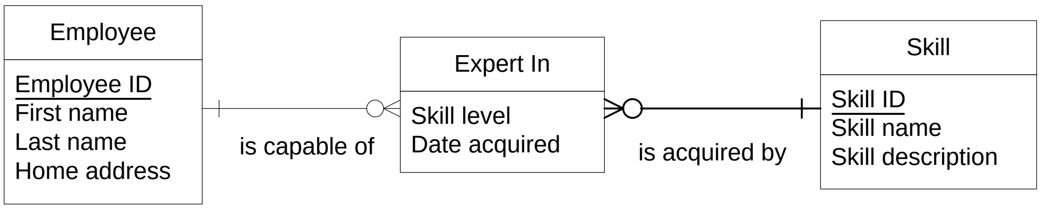
Observe that not all of the attributes you listed appear in this diagram. The missing items fall into two categories:

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 crucial that the data be associated with the correct entity. You can use your intuition. In the next section, you'll apply tests to check your judgment.

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.

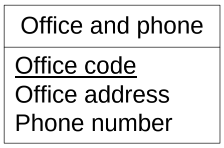
Entity or *relationship*	**Attributes**
Office	*Office code Office address Phone number
Works in	*Office code *Employee ID
Department	*Department ID Department name
Heads	*Department ID *Employee ID

Entity or *relationship*	**Attributes**
Member of	*Department ID *Employee ID
Skill	*Skill ID Skill name Skill description
Expert in	*Skill ID *Employee ID Skill level Date acquired
Employee	*Employee ID Last name First name Social security number Address Phone number Date of birth

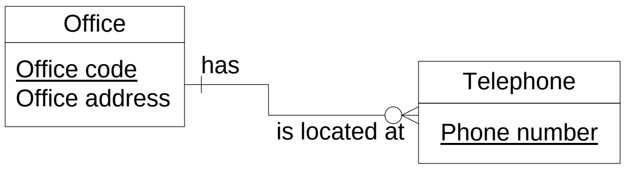
Putting data in first normal form

- To test for first normal form, look for attributes that can have repeating values.
- 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.



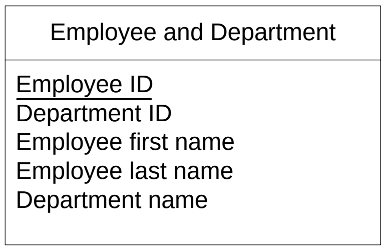
Remove the repeating attribute and make a new entity called Telephone. Set up a relationship between Office and Telephone.



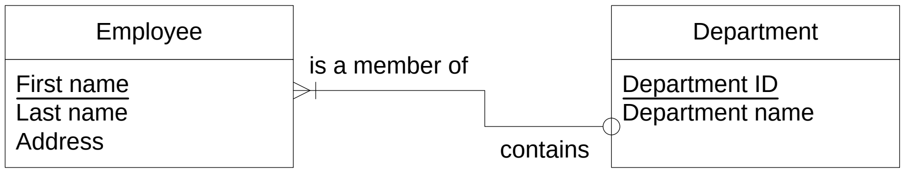
Putting data in second normal form

- ♦ 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.



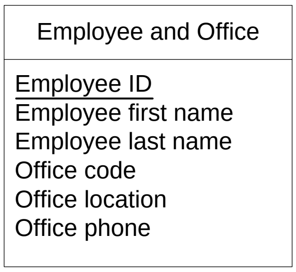
Move the identifier Department ID, which the other employee data does not depend on, to an entity of its own called Department. Also move any attributes that depend on it. Create a relationship between Employee and Department.



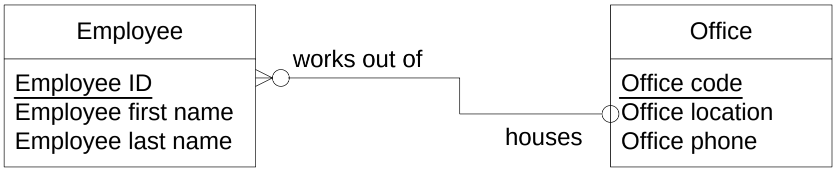
Putting data in third normal form

- ♦ 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.

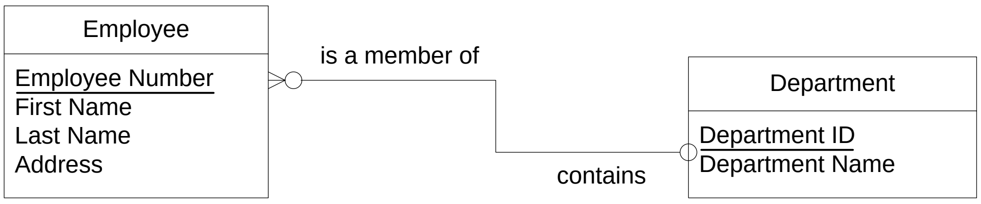
Whereas the conceptual model is largely independent of implementation details, the physical data model is tightly bound to the table structure and options available in a particular database application. In this case, that application is SQL Anywhere.

Resolving relationships that do not carry data

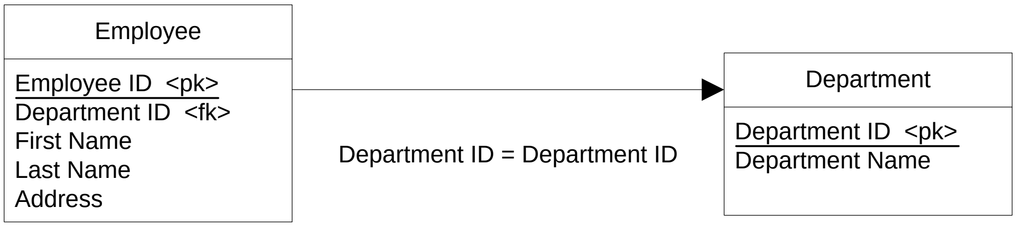
To implement relationships that do not carry data, you define foreign keys. A **foreign key** is a column or set of columns that contains primary key values from another table. The foreign key allows you to access data from more than one table at one time.

A database design tool such as PowerDesigner can generate the physical data model for you. However, if you're 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.

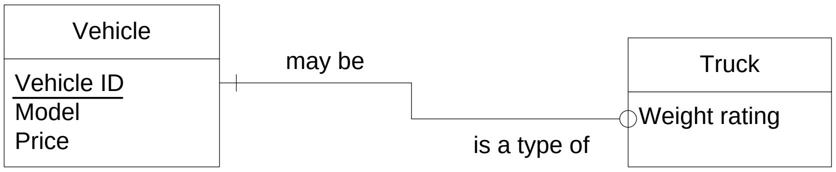


Notice that entities become tables. Identifiers in entities become (at least part of) the primary key in a table. Attributes become columns. In a one-to-many relationship, the identifier in the *one* entity will appear as a new foreign key column in the *many* table.

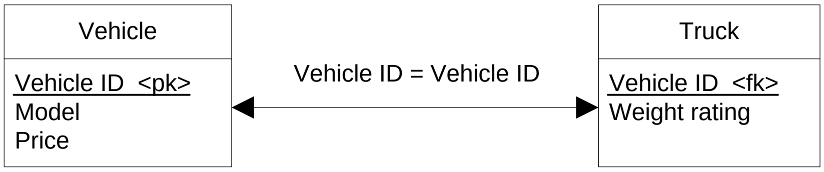


In this example, the Employee *entity* becomes an Employees *table*. Similarly, the Department entity becomes a Departments table. A foreign key called Department ID appears in the Employee table.

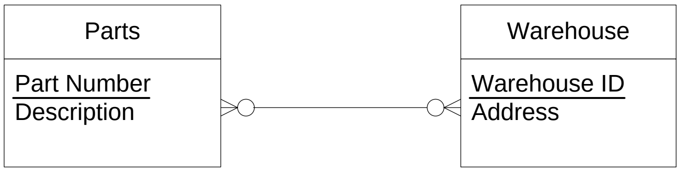
- **One to one** In a one-to-one relationship, the foreign key can go into either table. If the relationship is mandatory on one side, but optional on the other, it should go on the optional side. In this example, put the foreign key (Vehicle ID) in the Truck table because a vehicle does not have to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



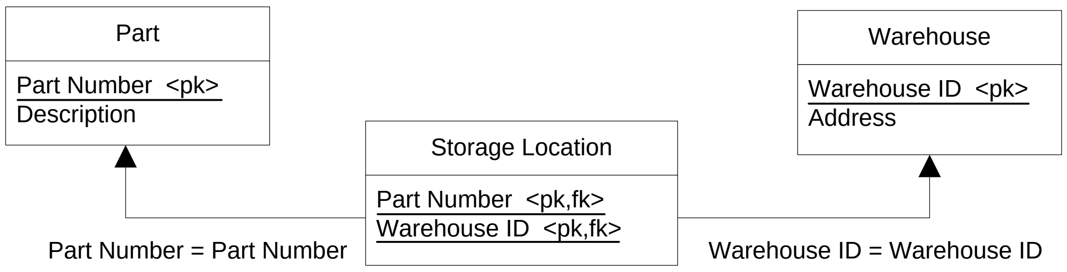
The above entity-relationship model thus resolves to the database base structure below.



- **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.

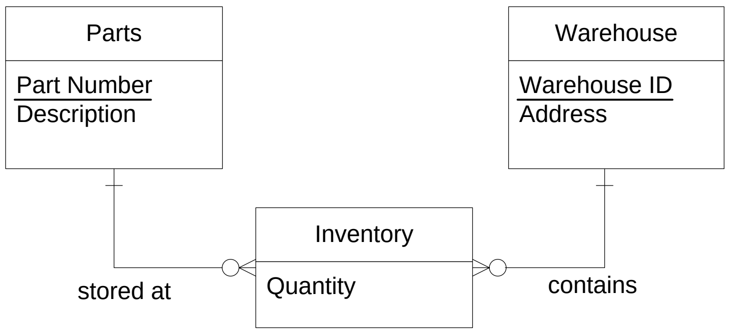


The new Storage Location table relates the Part and Warehouse tables.

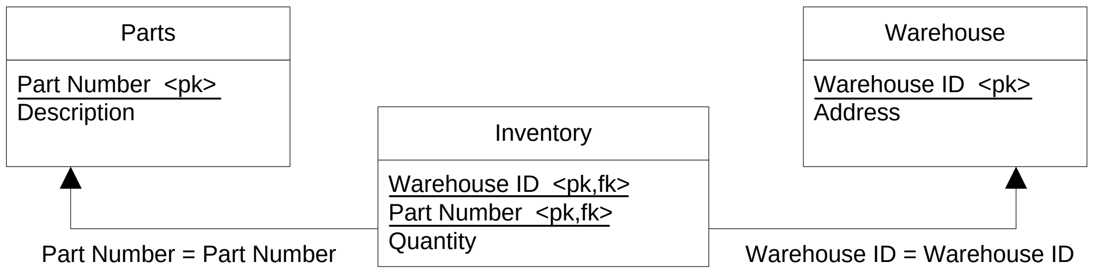


Resolving relationships that carry data

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

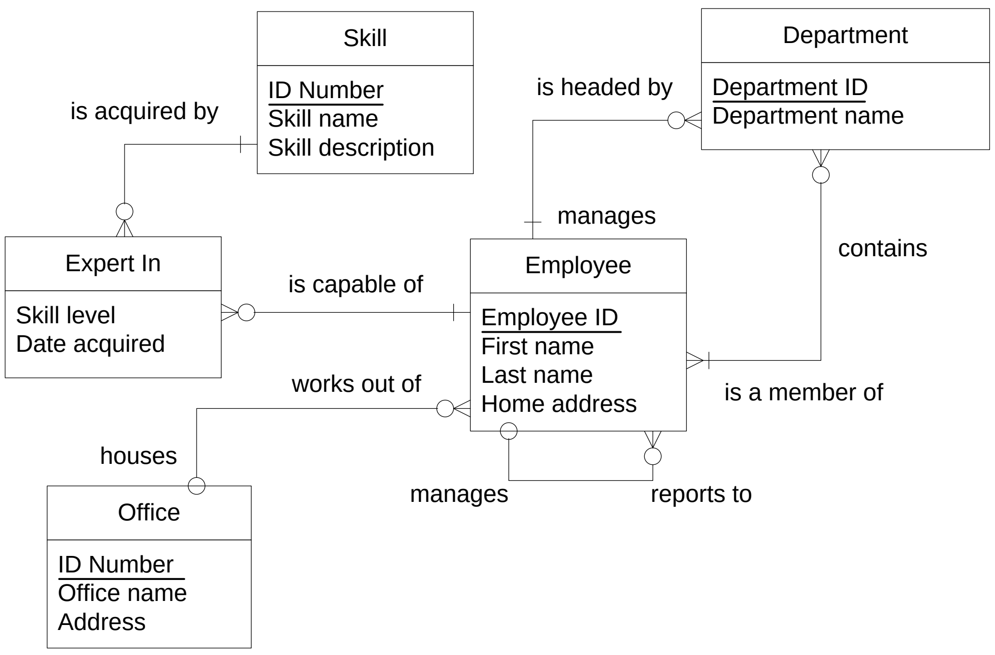
Before you implement your design, you need to make sure that it supports your needs. Examine the activities you identified at the start of the design process and make sure you can access all of the data that the activities require.

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

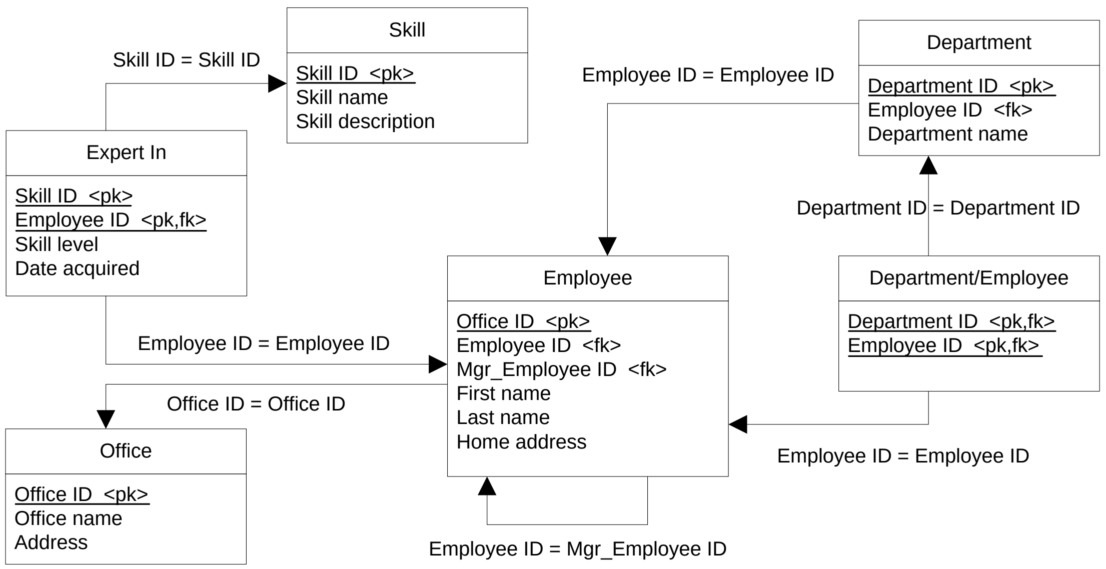
If you can answer yes to all the questions above, you are ready to implement your design.

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.


Skill
Skill ID <pk>
Skill name
Skill description
Skill ID = Skill ID
Expert In
Skill ID <pk>
Employee ID <pk,fk>
Skill level
Date acquired
Employee ID = Employee ID
Department
Department ID <pk>
Employee ID <fk>
Department name
Department ID = Department ID
Department/Employee
Department ID <pk,fk>
Employee ID <pk,fk>
Employee
Office ID <pk>
Employee ID <fk>
Mgr_Employee ID <fk>
First name
Last name
Home address
Employee ID = Employee ID
Office ID = Office ID
Office
Office ID <pk>
Office name
Address
Employee ID = Employee ID
Employee ID = Mgr_Employee ID

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:

Column	Data type	Size	Null/not null	Constraint
DepartmentID	integer	—	NOT NULL	None
DepartmentName	char	40	NOT NULL	None
DepartmentHea-dID	integer	—	NULL	None

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 on 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 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.

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

In the case of compressed columns, regardless of the settings of INLINE and PREFIX, the behavior is as though INLINE and PREFIX were set to 0. That is, no prefix is stored, and the BLOB is stored in table extension pages, and if INDEX was specified (the default), BLOB indexing is still performed. If, at a later time, the column is uncompressed, whatever settings were previously in effect for INLINE and PREFIX are restored.

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.

If you specify that a column is compressed, database server activities such as indexing, data comparisons, and statistics generation may be slightly slower if they involve the compressed column because the values must be compressed when written, and decompressed when read.

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

Column compression is not beneficial for columns containing values under 130 bytes, or values that are already in a compressed format, such as JPG files. In fact, compressing columns that contain compressed values 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*]
- ♦ "ALTER TABLE statement" [*SQL Anywhere Server - SQL Reference*]
- ♦ "sa_column_stats system procedure" [*SQL Anywhere Server - SQL Reference*]
- ♦ "Table encryption" [*SQL Anywhere Server - Database Administration*]

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 89.

CHAPTER 2

Working with Database Objects

Contents

About this chapter

This chapter describes the mechanics of creating, altering, and dropping database objects such as tables, views, and indexes.

Introduction

With the SQL Anywhere tools, you can create a database file to hold your data. Once this file is created, you can begin managing the database. For example, you can add database objects, such as tables or users, and you can set overall database properties.

This chapter describes how to create a database and the objects within it. It includes procedures for Sybase Central, Interactive SQL, and command line utilities. If you want more conceptual information before you begin, see the following topics:

- ♦ "Designing Your Database" on page 3
- ♦ "Ensuring Data Integrity" on page 89
- ♦ "Sybase Central" [*SQL Anywhere Server - Database Administration*]
- ♦ "Interactive SQL" [*SQL Anywhere Server - Database Administration*]

The SQL statements for performing the tasks in this chapter 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 723.

Chapter contents

This chapter contains the following material:

- ♦ An introduction to working with database objects
- ♦ A description of how to create and work with the database itself
- ♦ A description of how to create and alter tables, views, and indexes

Working with databases

This section describes how to create and work with a database. As you read this section, keep the following simple concepts in mind:

♦ The databases that you can create (called relational databases) are a collection of tables, related by primary and foreign keys. These tables hold the information in a database, and the tables and keys together define the structure of the database. A database can be stored in one or more database files, on one or more devices.

♦ A database file also contains the system tables, which hold the schema definition as you build your database.

Creating a database

SQL Anywhere provides a number of ways to create a database: in Sybase Central, in Interactive SQL, and at the command line. Creating a database is also called **initializing** it. Once the database is created, you can connect to it and build the tables and other objects that you need in the database.

Other application design systems, such as Sybase PowerDesigner Physical Model, contain tools for creating database objects. These tools construct SQL statements that are submitted to the database server, typically through its ODBC interface. If you are using one of these tools, you do not need to construct SQL statements to create tables, assign permissions, and so on. See "About PowerDesigner Physical Data Model" [*SQL Anywhere 10 - Introduction*].

This chapter describes the SQL statements for defining database objects. You can use these statements directly if you are building your database using a tool such as Interactive SQL. Even if you are using an application design tool, you may want to use SQL statements to add features to the database if they are not supported by the design tool.

☞ For more information about database design, see "Designing Your Database" on page 3.

Transaction log

When you create a database, you must decide where to place the transaction log. This log stores all changes made to a database, in the order in which they are made. In the event of a media failure on a database file, the transaction log is essential for database recovery. It also makes your work more efficient. By default, it is placed in the same directory as the database file, but this is not recommended for production use.

☞ For more information on placing the transaction log, see "Configuring your database for data protection" [*SQL Anywhere Server - Database Administration*].

Database file compatibility

A SQL Anywhere database is an operating system file. It can be copied to other locations just as any other file is copied.

Database files are compatible among all operating systems, except where file system file size limitations or SQL Anywhere support for large files apply. See "Size and number limitations" [*SQL Anywhere Server - Database Administration*].

A database created from any operating system can be used from another operating system by copying the database file(s). Similarly, a database created with a personal database server can be used with a network database server.

Creating databases (Sybase Central)

You can create a database in Sybase Central using the Create Database wizard.

☞ For more information, see "Creating databases (SQL)" on page 32, and "Creating databases (command line)" on page 33.

♦ To create a new database (Sybase Central)

1. From Tools menu, choose SQL Anywhere 10 ► Create Database.

   The Create Database wizard appears.

2. Follow the instructions in the wizard.

Creating databases for Windows CE

You can create databases for Windows CE by copying a SQL Anywhere database file to the device.

Sybase Central has features to make database creation easy for Windows CE databases. If you have Windows CE services installed on your Windows desktop, you have the option to create a Windows CE database when you create a database from Sybase Central. Sybase Central omits some features for Windows CE databases, and optionally copies the resulting database file to your Windows CE device.

Enable checksums for Windows CE databases
When creating a database that will be deployed to Windows CE, you should enable checksums. This helps to provide early detection if the database file becomes corrupt.

☞ For more information, see "SQL Anywhere for Windows CE" [*SQL Anywhere Server - Database Administration*].

Creating databases (SQL)

In Interactive SQL, use the CREATE DATABASE statement to create databases. You need to connect to an existing database before you can use this statement.

♦ To create a new database (SQL)

1. Start a database server named sample.

```
dbeng10 -n sample
```

2. Start Interactive SQL.

3. Connect to an existing database. If you don't have a database then you can connect to the utility database utility_db. See "Connecting to the utility database" [*SQL Anywhere Server - Database Administration*].

4. Execute a CREATE DATABASE statement.

☞ For more information, see "CREATE DATABASE statement" [*SQL Anywhere Server - SQL Reference*].

Example

Create a database file in the *c:\temp* directory with the file name *temp.db*.

```
CREATE DATABASE 'c:\\temp\\temp.db';
```

The directory path is relative to the database server. You set the permissions required to execute this statement on the server command line, using the -gu option. The default setting requires DBA authority.

The backslash is an escape character in SQL, and must be doubled in some cases. The \x and \n sequences can be used to specifying hexadecimal and newline characters. Letters other than n and x do not have any special meaning if they are preceded by a backslash. Here are some examples where this is important.

```
CREATE DATABASE 'c:\\temp\\\x41\x42\x43xyz.db';
```

The initial \\ sequence represents a backslash. The \x sequences represent the characters A, B, and C, respectively. The file name here is *ABCxyz.db*.

```
CREATE DATABASE 'c:\temp\\nest.db';
```

To avoid having the \n sequence interpreted as a newline character, the backslash is doubled.

☞ For more information, see "Strings" [*SQL Anywhere Server - SQL Reference*].

Creating databases (command line)

You can create a database from a command line with the Initialization utility (dbinit). With this utility, you can include command line options to specify different settings for the database.

♦ To create a new database (command line)

1. Open a command prompt.

2. Run the dbinit utility. Include any necessary parameters.

   For example, to create a database called *company.db* with a 4 KB page size, enter:

   ```
   dbinit –p 4k company.db
   ```

☞ For more information, see "The Initialization utility" [*SQL Anywhere Server - Database Administration*].

Starting a database

With both Sybase Central and Interactive SQL, you can start a database without connecting to it.

♦ To start a database on a server without connecting (Sybase Central)

1. Select the desired server and then choose File ► Start Database.
2. In the Start Database dialog, enter the required values.

   The database appears under the database server as a disconnected database.

♦ To start a database on a server without connecting (SQL)

- Execute a START DATABASE statement.

☞ For more information, see "START DATABASE statement" [*SQL Anywhere Server - SQL Reference*].

Example

Start the database file *c:\temp\temp.db* on the database server named sample.

```
START DATABASE 'c:\\temp\\temp.db'
AS tempdb ON 'sample'
AUTOSTOP OFF;
```

You must be connected to a database to start another database.

The AUTOSTOP OFF prevents the database from being stopped automatically when all connections have been terminated. It is used here to illustrate a point later on in the discussion.

Connecting to a database

Before you can begin working with a database, you must connect to it.

When a database server is started, it assigns a unique connection ID to each new connection to it. You can obtain a user's *connection-id* using the CONNECTION_PROPERTY function to request the connection number. The following statement returns the connection ID of the current connection:

```
SELECT CONNECTION_PROPERTY( 'number' );
```

♦ To connect to a database (Sybase Central)

1. Choose Connections ► Connect with SQL Anywhere 10.

   The Connect dialog appears.
2. In the Connect dialog, specify the connection information, or ODBC data source name, to use to connect to the database, and then choose OK.

♦ To connect to a database (SQL)

- Execute a CONNECT statement.

☞ For more information, see "CONNECT statement [ESQL] [Interactive SQL]" [*SQL Anywhere Server - SQL Reference*].

Example 1

The following statement shows how to use CONNECT from Interactive SQL to connect to a database that has been started:

```
CONNECT to 'sample' DATABASE tempdb
AS conn1
USER 'DBA'
IDENTIFIED BY 'sql';
```

In this example, sample is the server name, tempdb is the database name, conn1 is the connection identifier, DBA is the user name, and sql is the password.

Example 2

The following statement shows how to use CONNECT in embedded SQL:

```
EXEC SQL CONNECT "DBA"
IDENTIFIED BY "sql";
```

Setting properties for database objects

Most database objects (including the database itself) have properties that you can either view or set. Some properties are non-configurable and reflect the settings chosen when you created the database or object. Other properties are configurable.

The best way to view and set properties is to use the property sheets 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 options 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 desired properties.

   You can also view and edit properties on the object's property sheet. To view the property sheet, select the object and then choose File ▶ Properties.

Setting database options

Database options are configurable settings that change the way the database behaves or performs. In Sybase Central, all of these options are grouped together in the Database Options dialog. In Interactive SQL, you can change an option setting using a SET OPTION statement.

♦ To set options for a database (Sybase Central)

1. Open the desired server.
2. Right-click the desired database and choose Options from the popup menu.
3. Edit the desired values.

♦ To set the options for a database (SQL)

- Specify the desired values within a SET OPTION statement.

Tips
With the Database Options dialog, you can also set database options for specific users and groups (when you open this dialog for a user or group, it is called the User Options dialog or Group Options dialog, respectively).
When you set options for the database itself, you are actually setting options for the PUBLIC group in that database because all users and groups inherit option settings from PUBLIC.

☞ For more information about options, see "Introduction to database options" [*SQL Anywhere Server - Database Administration*].

Specifying a consolidated database

In Sybase Central, you can specify the consolidated database for SQL Remote replication. The consolidated database is the one that serves as the master database in the replication setup. The consolidated database contains all of the data to be replicated, while its remote databases can only contain their own subsets of the data. In case of conflict or discrepancy, the consolidated database is considered to have the primary copy of all data.

☞ For more information, see "Consolidated and remote databases" [*SQL Anywhere 10 - Introduction*].

♦ To set a consolidated database (Sybase Central)

1. Select the desired database and then choose File ► Properties.
2. Click the SQL Remote tab.
3. Select the This Remote Database Has a Corresponding Consolidated Database option.
4. Configure the desired settings.

Displaying system objects in a database

In a database, a table, view, stored procedure, or domain is a system object. **System tables** store the database's schema, or information about the database itself. System views, procedures, and domains largely support Sybase Transact-SQL compatibility.

All the information about system objects in a database appears in the system tables. The information is distributed among several tables.

♦ To display system objects in a database (Sybase Central)

1. Open the desired database server.
2. Select the desired connected database and then choose File ► Filter Objects by Owner.
3. Select **SYS** and **dbo**, and then click OK.

   The system tables, system views, and system procedures appear in their respective folders. For example, system tables appear alongside normal tables in the Tables folder.

♦ 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 objects in the database. A join is made to the SYSTAB system view to return the object name, and SYSUSER system view to return the owner name.

```
SELECT b.table_name 'Object Name', c.user_name Owner, b.object_id,
 a.object_type, a.status
  FROM ( SYSOBJECT a JOIN SYSTAB b
   ON a.object_id = b.object_id )
    JOIN SYSUSER c
  ORDER BY table_name;
```

☞ For more information on the SYSOBJECT, SYSTAB, and SYSUSER system views, see "SYSOBJECT system view" [*SQL Anywhere Server - SQL Reference*], "SYSTAB system view" [*SQL Anywhere Server - SQL Reference*], and "SYSUSER system view" [*SQL Anywhere Server - SQL Reference*].

Logging SQL statements

As you work with a database in Sybase Central, the application automatically generates SQL statements depending on your actions. You can keep track of these statements in a separate pane, called Server Messages and Executed SQL or save the information to a file. The Server Messages and Executed SQL pane has a tab for each database and database server. The tab for database servers contains the same information as the Server Messages window.

When you work with Interactive SQL, you can also log statements that you execute.

☞ For more information, see "Logging commands" [*SQL Anywhere Server - Database Administration*].

♦ To log SQL statements generated by Sybase Central

1. From the View menu, choose Server Messages and Executed SQL.

   The Server Messages and Executed SQL pane appears at the bottom of the Sybase Central window.

2. Right-click in the Server Messages and Executed SQL pane and choose Options from the popup menu.

   The Options dialog appears.

3. In the resulting dialog, specify the desired settings. If you want to save the logging information to a file, click Save.

Adding jConnect metadata support to an existing database

If a database was created without jConnect metadata support, you can use Sybase Central to install it at a later date.

♦ To add jConnect metadata support to an existing database (Sybase Central)

1. Connect to the database you want to upgrade.

2. From the Tools menu, choose SQL Anywhere 10 ► Upgrade Database.

   The Upgrade Database wizard appears.

3. Click Next on the introductory page of the wizard.

4. Select the database you want to upgrade from the list. Click Next.

5. You can choose to create a backup of the database if you want. Click Next.

6. Select the Install jConnect Meta-Information Support option.

7. Follow the remaining instructions in the wizard.

Disconnecting from a database

When you are finished working with a database, you can disconnect from it. SQL Anywhere also gives you the ability to disconnect other users from a given database; for more information about doing this in Sybase Central, see "Managing connected users" [*SQL Anywhere Server - Database Administration*].

When a database server is started, it assigns a unique connection ID to each new connection to the database server. You can obtain a user's *connection-id* using the CONNECTION_PROPERTY function to request the connection number. The following statement returns the connection ID of the current connection:

```
SELECT CONNECTION_PROPERTY( 'number' );
```

♦ To disconnect from a database (Sybase Central)

1. Select a database.
2. Choose File ▶ Disconnect.

♦ To disconnect from a database (SQL)

- Execute a DISCONNECT statement.

☞ For more information, see "DISCONNECT statement [ESQL] [Interactive SQL]" [*SQL Anywhere Server - SQL Reference*], and "DROP CONNECTION statement" [*SQL Anywhere Server - SQL Reference*].

Example 1

The following statement shows how to use the DISCONNECT statement to disconnect the current connection, conn1, in Interactive SQL:

```
DISCONNECT conn1;
```

Example 2

The following statement shows how to use DISCONNECT in embedded SQL:

```
EXEC SQL DISCONNECT :conn-name
```

♦ To disconnect other users from a database (SQL)

1. Connect to an existing database with DBA authority.
2. Use the sa_conn_info system procedure to determine the connection ID of the user you want to disconnect.
3. Execute a DROP CONNECTION statement.

Example

The following statement drops connection number 4.

```
DROP CONNECTION 4;
```

Stopping a database

With both Sybase Central and Interactive SQL, you can stop a database running on a database server. You cannot stop a database you are currently connected to. You must first disconnect from the database, and then stop it. You must be connected to another database on the same database server in order to stop a database.

♦ To stop a database on a server after disconnecting (Sybase Central)

1. Make sure you are connected to at least one other database on the same database server. If there is no other database running on the server, you can connect to the utility database.

2. Select the database you want to stop and choose File ▶ Stop Database.

When disconnecting from the database, the database may disappear from the left pane. This occurs if your connection was the only remaining connection, and if AUTOSTOP was specified when the database was started. AUTOSTOP causes the database to be stopped automatically when the last connection is terminated.

♦ To stop a database on a server after disconnecting (SQL)

1. If you aren't connected to any database on the server, then connect to a database such as the utility database.
2. Execute a STOP DATABASE statement.

☞ For more information, see "STOP DATABASE statement" [*SQL Anywhere Server - SQL Reference*].

☞ For information on connecting to the utility database, see "Connecting to the utility database" [*SQL Anywhere Server - Database Administration*].

Example

The following statements connect to the utility database and stops the tempdb database.

```
CONNECT to 'TestEng' DATABASE utility_db
AS conn2
USER 'DBA'
IDENTIFIED BY 'sql';
STOP DATABASE tempdb;
```

You must be connected to a database to stop another database.

Erasing a database

Erasing a database deletes all tables and data from disk, including the transaction log that records alterations to the database. All database files are read-only to prevent accidental modification or deletion of database files.

In Sybase Central, you can erase a database using the Erase Database wizard. You need to connect to a database to access this wizard, but the Erase Database wizard lets you specify any database for erasing. To erase a non-running database, a database server must be running.

In Interactive SQL, you can erase a database using the DROP DATABASE statement. Required permissions can be set using the database server -gu server option. The default setting is to require DBA authority.

You can also erase a database from a command line with the dberase utility. The database to be erased must not be running when the dberase utility, the Erase Database wizard, or DROP DATABASE statement is used.

♦ To erase a database (Sybase Central)

1. From the Tools menu, choose SQL Anywhere 10 ▶ Erase Database.

2. Follow the instructions in the wizard.

☞ For more information, see “Erase Database wizard” [*SQL Anywhere Server - Database Administration*].

♦ To erase a database (SQL)

1. Connect to a database other than the one you want to erase. For example, connect to the utility database.
2. Execute a DROP DATABASE statement.

☞ For more information, see “DROP statement” [*SQL Anywhere Server - SQL Reference*].

☞ For information on connecting to the utility database, see “Connecting to the utility database” [*SQL Anywhere Server - Database Administration*].

Example

The following DROP DATABASE statement erases the temp database.

```
DROP DATABASE 'c:\\temp\\temp.db';
```

You must be connected to a database to drop another database.

♦ To erase a database (command line)

- From a command line, run the dberase utility.

  For example, the following command removes the temp database.

  ```
  dberase c:\temp\temp.db
  ```

☞ For more information, see “Erase utility (dberase)” [*SQL Anywhere Server - Database Administration*].

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. For more information about temporary tables, see "Working with temporary tables" on page 87.

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 a database.
2. Open the Tables folder.
3. From the File menu, choose New ► Table.

   The Create Table wizard appears.
4. Follow the instructions in the wizard.

   The new table appears in the Table folder.

5. On the Columns tab in the right pane, you can add columns to the table.

6. Choose File ► Save when finished.

♦ **To create a table (SQL)**

1. Connect to the database 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
);
```

☞ For more information, 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 on altering database object properties, see "Setting properties for database objects" on page 35.

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

Table alterations and view dependencies

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.

When you alter the schema of a table, the definition for the table in the database is updated. If there are dependent, non-materialized views, the database server automatically recompiles them after you perform the table alteration. If the database server cannot recompile a dependent non-materialized 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 views" on page 59.

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 70.

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 altering database objects affects view dependencies, see "View dependencies" on page 63.

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.

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*].

☞ For more information about view dependencies, see "View dependencies" on page 63.

♦ To alter an existing table (Sybase Central)

1. Connect to the database as the DBA 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. Open the Views folder and select the materialized view.
   b. Choose File ▶ Disable.
3. Open the Tables folder and select the table you want to alter.
4. Click the Columns tab in the right pane and make the necessary changes.
5. Choose File ▶ Save.

   The database server saves the changes to the table.
6. If you disabled materialized views, re-enable and initialize each one as follows:
   a. Open the Views folder and select the materialized view.
   b. Choose File ▶ Recompile and Enable.
   c. Choose File ▶ Refresh Data.

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 desired 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 74
- "Ensuring Data Integrity" on page 89
- "View dependencies" on page 63

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.

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*].

☞ For more information about view dependencies, see "View dependencies" on page 63.

♦ To alter an existing table (SQL)

1. Connect to the database 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 non-materialized 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 views" on page 59
- "Enabling and disabling materialized views" on page 74
- "ALTER MATERIALIZED VIEW statement" [*SQL Anywhere Server - SQL Reference*]
- "Ensuring Data Integrity" on page 89
- "View dependencies" on page 63

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, non-materialized 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 views" on page 59.

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 70.

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 63.

♦ To drop a table (Sybase Central)

1. Connect to the database as a DBA or as the owner of the table.
2. If you are dropping a table on which materialized views depend, disable each materialized view as follows:
   a. Open the Views folder.
   b. In the left pane, select the materialized view.
   c. Select File ▶ Disable.
3. Open the Tables folder for that database.
4. Select the table and then choose Edit ▶ Delete.

♦ To drop a table (SQL)

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

Example

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

```
DROP TABLE Skills;
```

Like the CREATE statement, the DROP statement automatically executes a COMMIT statement before and after dropping the table. This makes all changes to the database since the last COMMIT or ROLLBACK permanent. The DROP statement also drops all indexes on the table.

☞ For more information, 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:

```
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.

Managing primary keys

The **primary key** is a unique identifier that is comprised of a column or combination of columns with values that 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. In the left pane, open the Tables folder.
2. Select the table for which you want to create or modify the primary key, and choose File ▶ Set Primary Key.

   The Set Primary Key wizard appears.
3. Follow the instructions in the wizard to create or modify the primary key.
4. When you have finished configuring the primary key in the Set Primary Key wizard, select Finish.

The new primary key information is automatically saved for the table.

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

Example 1

The following statement creates the same Skills table as before, except that it adds a primary key using the values in the SkillID column:

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

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

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

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

☞ For more information, see “ALTER TABLE statement” [*SQL Anywhere Server - SQL Reference*], and “Managing primary keys (Sybase Central)” on page 48.

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.

In this example, the Employees table (which contains the foreign key in the relationship) is 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. Select the table for which you want to create a foreign key.
2. Click the Constraints tab in the right pane.
3. From the File menu, choose New ► Foreign Key.

The Create Foreign Key wizard appears.

4. Follow the instructions in the wizard.

♦ To delete a foreign key (Sybase Central)

1. Select the table for which you want to delete a foreign key.
2. Click the Constraints tab in the right pane.
3. Select the foreign key you want to delete and then choose Edit ► Delete.

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. In the left pane, select the desired table.
2. In the right pane, click the Referencing Constraints tab.

Tips
When you create a foreign key using the wizard, you can set properties for the foreign key. To view properties after the foreign key is created, select the foreign key on the Constraints tab and then choose
File ► Properties.
You can view the properties of a referencing foreign key by selecting the table on the Referencing Constraints tab and then choosing File ► Properties.

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 with DBA authority.
2. Execute an ALTER TABLE statement.

Example 1

In this 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.

```
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 REFERENCES Skills ( Id )
   );
```

Notice that EmployeeSkills.SkillID has a foreign key relationship with the primary key column (Id) of the Skills table.

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;
```

☞ For more information, see “ALTER TABLE statement” [*SQL Anywhere Server - SQL Reference*], and .

In Sybase Central, you can also specify properties in the Create Foreign Key wizard or on the foreign key's property sheet. See “Managing foreign keys (Sybase Central)” on page 50.

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 this query the predicate in the WHERE clause is not sargable since it does not refer to a single base column. See "Predicate analysis" on page 492. If the size of the Shipments table is relatively large, an indexed retrieval might be appropriate rather than a sequential scan. You can do this by creating a computed column named AverageCost for the Shipments table, 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

```
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.
- **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, which can aid the DBA during complex unload/reload sequences, or when it is vital that the value of a computed column stay constant when the COMPUTE expression refers to non-deterministic values, such as CURRENT TIMESTAMP.

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; however, since computed columns are base columns like any other, they can be directly referenced in predicates and in expressions when it makes sense to do so.

Computed columns are *not* recalculated when queried. Instead, 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.

If you use a time-dependent expression, or one that depends on the state of the database in some other way, then the computed column can not give a proper result.

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.

☞ For information about copying database objects in Sybase Central, see "Copying database objects in the SQL Anywhere plug-in" [*SQL Anywhere Server - Database Administration*].

☞ If you are not using Sybase Central, see one of the following locations:

- 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 views

Views are computed tables. 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: non-materialized views, also referred to as just views, and materialized views. The two types of view differ in that non-materialized views are recomputed each time you invoke them. Materialized views are computed, stored on disk similar to a base table, and need their data refreshed periodically.

Since materialized views are stored and managed slightly differently, their tasks are documented separately. If you want information about materialized views, see "Working with materialized views" on page 68.

Views have a status associated with them that indicates whether the data they contain is up to date, and whether they are available to answer a query. See "View status" on page 64.

Comparing materialized views, non-materialized views, and base tables

The following table highlights the things you can and cannot do with views and tables.

Capability	Materialized views	Non-materialized views	Base tables
Allow access permissions	Yes	Yes	Yes
Allow SELECT on them	Yes	Yes	Yes
Allow UPDATE	No	Some	Yes
Allow INSERT	No	Some	Yes
Allow DELETE	No	Some	Yes
Allow dependent views	Yes	Yes	Yes
Allow indexes	Yes	No	Yes
Allow integrity constraints	No	No	Yes
Allow keys	No	No	Yes

Benefits of using views

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

- **Efficient resource use** Non-materialized 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. This 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.

Creating views

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

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

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

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

When you create a 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 view (Sybase Central)

1. Connect to a database.
2. Open the Views folder for that database.
3. From the File menu, choose New ► View.

   The Create View wizard appears.
4. Follow the instructions in the wizard.

   The new view appears in the Views folder.
5. When the wizard exits, you can edit the view definition in the SQL tab in the right pane. If you do so, you must save the changes. From the File menu, choose Save.

♦ To create a new view (SQL)

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

Example

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

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

☞ For more information, see "CREATE VIEW statement" [*SQL Anywhere Server - SQL Reference*].

Using views

Restrictions on SELECT statements

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

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

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

Updating views

UPDATE, INSERT, and DELETE statements are allowed on some non-materialized 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 operation. In all these cases, there is no way to translate the UPDATE into an action on the underlying tables.

Copying views

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

Using the WITH CHECK OPTION clause

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

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

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

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

```
SELECT * FROM SalesEmployees;
```

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

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

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

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

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

The check option is inherited

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

Altering views

You can alter a 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 directly. Instead, you must create a new view with the new name, copy the previous definition to it, and then drop the old view.

☞ For information on altering database objects, see “Setting properties for database objects” on page 35.

☞ For information on setting permissions, see “Granting permissions on tables” [*SQL Anywhere Server - Database Administration*] and “Granting permissions on views” [*SQL Anywhere Server - Database Administration*].

☞ For information on revoking permissions, see "Revoking user permissions" [*SQL Anywhere Server - Database Administration*].

View alterations and view dependencies

If you want to alter the definition for a view, and there are other views dependent on the view, there may be additional steps to make after the view 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 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 view consistent with the definitions of the dependents views, or vice versa.

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

☞ For information on how view alterations affect view dependencies, see "View dependencies" on page 63.

♦ To alter a view definition (Sybase Central)

1. Connect to the database as the DBA or as the owner of the view.
2. Open the Views folder.
3. Select the desired 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 from the popup menu.

5. From the File menu, select Save.

♦ To alter a view definition (SQL)

1. Connect to the database as a DBA or as the owner of the view.
2. Execute an ALTER VIEW statement.

Examples

This example shows that when you are changing schema-related aspects of the view, you are effectively replacing the definition of the view. In this case, the view definition is being changed to have column names that are more informative.

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

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

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

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

☞ For more information altering views, see "ALTER VIEW statement" [*SQL Anywhere Server - SQL Reference*].

Dropping views

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

View deletions and view dependencies

If you drop a view that has dependent views, then the dependent views are made INVALID as part of the drop operation. The dependent views will not be usable until they are changed or the original dropped view is recreated. See "Altering views" on page 59.

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

☞ For information on how view dependencies affect view alterations, see "View dependencies" on page 63.

♦ To drop a view (Sybase Central)

1. Connect to the database as a DBA, or as the owner of the view.
2. Open the Views folder.
3. Select the desired view and then choose Edit ▶ Delete.

♦ To drop a view (SQL)

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

Examples

Remove a view called DepartmentSize.

```
DROP VIEW DepartmentSize;
```

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

Enabling and disabling views

The section describes enabling and disabling non-materialized views. For information on enabling and disabling materialized views, see "Enabling and disabling materialized views" on page 74.

You can control whether a view is available for use by the database server by enabling or disabling it. When you disable a view, the database server keeps the definition of the view in the database; however, the view is not available for use. 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 start using 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 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; otherwise, the view remains unchanged. 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 view, you must re-enable any other views that it references (if they are disabled).

♦ To enable a view (Sybase Central)

1. Connect to the database as a DBA or as the owner of the view.
2. Open the Views folder for that database.
3. Select the view to enable.
4. Choose File ► Recompile and Enable.

♦ To disable a view (Sybase Central)

1. Connect to the database as a DBA or as the owner of the view.
2. Open the Views folder for that database.
3. Select the view to disable.
4. Choose File ► Disable.

♦ To enable a view (SQL)

1. Connect to the database as a DBA or as the owner of the view.
2. Execute an ALTER VIEW ... ENABLE statement.

♦ **To disable a view (SQL)**

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

Examples

The following example disables a view called ViewSalesOrders.

```
ALTER VIEW ViewSalesOrders DISABLE;
```

The following example re-enables the ViewSalesOrders view.

```
ALTER VIEW 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*]

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 can be considered **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 refer to another view, which may itself refer to other views and tables.

Consider the following set of tables and views:

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

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

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

The database server keeps track of columns, tables, and views referenced by any 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 invalid state. In the case of non-materialized views, the database server provides this guarantee by automatically recompiling all referencing views whenever schema changes are made to referenced objects. In the case of materialized views, schema changes to a referenced object are not permitted if the object has any enabled materialized views.

Following is a list of considerations for view dependencies:

- A non-materialized view can reference tables or views.
- A materialized view can only reference base tables.
- Before you drop a table, or alter a table or view, you must disable or drop all dependent materialized views.
- When you disable or drop a view, all dependent non-materialized views are automatically disabled.
- The DISABLE VIEW DEPENDENCIES clause of the ALTER TABLE statement only disables non-materialized dependent views. Any dependent materialized views must be explicitly disabled or dropped.
- Once you disable a view, it can only be re-enabled explicitly.
- Unlike non-materialized views, which are disabled if a referenced object is disabled, materialized views can only be disabled explicitly.

View status

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 select Properties to examine the Status value.

Following are descriptions of the possible statuses:

- **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.

  A VALID materialized view may not yet have data in it. This can occur if the materialized view was never initialized, or if initialization (or a refresh) failed. If the database server receives a request that references a materialized view that is in an uninitialized state, it attempts to initialize it.

  Note
  A materialized view with stale data can still have the status VALID. The validity of a view relates to its schema being consistent with the schema(s) of the underlying objects, not to the staleness of data.

  In the SYSOBJECT system view, the value 1 indicates a status of VALID.

- **INVALID** *This status does not apply to materialized views.* 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.

- **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 non-materialized 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.

  A materialized view has a DISABLED state only if you explicitly disable it, for example using the ALTER MATERIALIZED VIEW … DISABLE statement. When you disable a materialized view, the data for the view is dropped.

  ☞ For information on enabling and disabling views, see “Enabling and disabling views” on page 62, and “Enabling and disabling materialized views” on page 74.

  In the SYSOBJECT system view, the value 4 indicates a status of DISABLED.

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 first disable dependent materialized views before proceeding with the operation. See “Enabling and disabling materialized views” on page 74.

2. The database server obtains exclusive schema locks on the table or view being modified, as well as on all dependent views.

3. The database server sets the status of all dependent views to INVALID.

4. The database server performs the schema-altering operation. If the operation fails, the locks are released, the status of dependent views is reset to VALID, an error is returned, and the following step does not occur.

5. The database server recompiles the dependent views, setting each view's status to VALID when successful. If compilation fails for any view, the status of that view remains INVALID. Subsequent requests for an INVALID 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, for it to compile successfully.

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*].

Browsing data in views

To browse the data held within the views, you can use the Interactive SQL utility. This utility lets you execute queries to identify the data you want to view.

☞ For more information about using these queries, see "Queries: Selecting Data from a Table" on page 263.

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.

Viewing system table data

Data in the system tables is only viewable using views—specifically, system views; you cannot query a system table directly. With a few exceptions, almost all of the system tables have a corresponding view. The system views are named similar to the system tables, but without an I at the beggining. For example, the list of all views in the database is stored in the ISYSTAB system table. You can access this information using the SYSTAB system view. You can also use the more readable system view SYSVIEWS (recommended) to retrieve the same information.

☞ For more information about these, see "SYSTAB system view" [*SQL Anywhere Server - SQL Reference*], and "SYSVIEWS consolidated view" [*SQL Anywhere Server - SQL Reference*].

☞ For a list of views provided in SQL Anywhere, as well as a description of the type of information they contain, see "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 the DBA.
2. Open the Views folder.
3. Select the view corresponding to the desired system table.
4. In the right pane, switch to the Data tab.

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

1. Connect to the database as the DBA.
2. Execute a SELECT statement that references the system view corresponding to the desired 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 SYS.SYSTAB system view:

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

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

```
SELECT viewtext
FROM SYS.SYSVIEWS
WHERE viewname = 'SYSTAB';

OUTPUT TO viewtext.sql
FORMAT ASCII
ESCAPES OFF
QUOTE '';
```

Working with 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, frequent queries result in repetitive aggregation and join operations on large amounts of data, and access to up-to-the-moment data is not a critical requirement. For example, materialized views are ideal for use with data warehousing applications.

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) 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.

Materialized views get their data from the execution of the underlying queries and 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. For more information about Column statistics, see "Optimizer estimates and column statistics" on page 488.

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

Considerations when using 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 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 periodically refreshed. 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 changing data renders the data in materialized views obsolete soon after the view is refreshed.
    - **Cost of refreshing** Depending on the complexity of the underlying query, and the amount of data involved, the computation of a materialized view may be very expensive. Frequently refreshing such a view may impose an unacceptable level of load on the database server.
    - **Data concurrency requirements of applications** The use of materialized views by the database server to answer queries means that the database server can present stale data to applications. Stale data is data that no longer represents the current state of the database. 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 on managing data staleness in materialized views, see "Setting the optimizer staleness threshold for materialized views" on page 77.

- ♦ **Data consistency requirements** When refreshing materialized views, you must determine the consistency with which the materialized view 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 529.

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. See "Application profiling" on page 188.

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. For more information about this tracing type, see "Choosing a tracing level" on page 202.

☞ For more information on how the optimizer uses materialized views, see "Improving performance with materialized views" on page 494.

Restrictions when managing materialized views

The following restrictions apply when creating, initializing and refreshing materialized views, and 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; everything 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 in order for the view to be used in by the optimizer.
  - ♦ ansi_integer_overflow=On
  - ♦ ansinull=On
  - ♦ conversion_error=On
  - ♦ divide_by_zero_error=On

    - float_as_double=Off
    - sort_collation=Null
    - string_rtruncation=On

- The following database option settings are remembered for each materialized view when it is created. The current option values for the connection must match the values remembered 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
    - uuid_has_hyphens

- When a view is refreshed, the connection settings for all of the above options are ignored. Instead, the values for these settings at the time of the view's creation 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.

This is different behavior than non-materialized views for which the ORDER BY clause orders the results returned.

☞ For information on materialized views and view matching by the optimizer, see “Improving performance with materialized views” on page 494.

Creating materialized views

When creating a materialized view, you first create and store its definition, or schema, 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.

Before creating, initializing, or refreshing materialized views, ensure that all materialized view restrictions have been met. See “Restrictions when managing materialized views” on page 69.

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, its definition is stored in the database, and the new materialized view appears in the Views folder. However, there is no data in the materialized view. If you want the materialized view to be populated with data, you must initialize it. See "Initializing materialized views" on page 72.

♦ To create a new materialized view (Sybase Central)

1. Connect to the database with DBA or RESOURCE privileges.
2. Open the Views folder.
3. From the File menu, choose New ► Materialized View.

   The Create Materialized View wizard appears.
4. Follow the instructions in the wizard.
5. You must now initialize the materialized view so that it contains data. See "Initializing materialized views" on page 72.

♦ To create a materialized view (SQL)

1. Connect to the database with DBA or RESOURCE privileges.
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.
3. You must now initialize the materialized view so that it contains data. See "Initializing materialized views" on page 72.

Example

The following statement creates a materialized view, EmployeeConfidential, containing confidential information about employees.

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

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 10 - Introduction*]

Initializing materialized views

An uninitialized material view has no data, and exists as a definition in the database. A materialized view is in an uninitialized state just after creation, or just after being re-enabled. A failed refresh attempt can also return the materialized view to an uninitialized state. You must initialize a materialized view to make it available for use by the database server.

Before creating, initializing, or refreshing materialized views, ensure that all materialized view restrictions have been met. See “Restrictions when managing materialized views” on page 69.

♦ To initialize a materialized view (Sybase Central)

1. Connect to the database as the DBA, or with INSERT permission on the materialized view.
2. Open the Views folder.
3. Select the materialized view and choose File ► Refresh Data.

♦ To initialize a materialized view (SQL)

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

Example

In this example, the EmployeeConfidential materialized view is initialized:

```
REFRESH MATERIALIZED VIEW EmployeeConfidential;
```

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*].

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 74

Refreshing materialized views

Data in a materialized view becomes stale when the data changes in the tables referenced by the materialized view. You should consider the acceptable degree of data staleness for the materialized view, and automate a refresh strategy accordingly. In your strategy, you should factor the time it takes to refresh the view, since it is not available for querying during the refresh process. Once you have decided how often to refresh the view, you can create a scheduled event or trigger that refreshed the view.

The isolation level for the connection is used when refreshing the data in the materialized view.

You can also 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 77.

Before creating, initializing, or refreshing materialized views, ensure that all materialized view restrictions have been met. See "Restrictions when managing materialized views" on page 69.

♦ To refresh a materialized view (Sybase Central)

1. Connect to the database as the DBA, or with INSERT permission on the materialized view.
2. Open the Views folder for that database.
3. Select the materialized view and choose File ► Refresh Data.

♦ To refresh a materialized view (SQL)

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

Example

The following statement refreshes the EmployeeConfidential materialized view.

```
REFRESH MATERIALIZED VIEW EmployeeConfidential;
```

See also

- ♦ "Automating Tasks Using Schedules and Events" [*SQL Anywhere Server - Database Administration*]
- ♦ "materialized_view_optimization option [database]" [*SQL Anywhere Server - Database Administration*]

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.

♦ To encrypt a materialized view (Sybase Central)

1. Connect to the database as the DBA.
2. Open the Views folder and right-click the materialized view.

3. Select Properties.

   The Properties page for the materialized view displays.

4. On the Miscellaneous tab, place a checkmark next to View data is encrypted, and then choose OK.

♦ To decrypt a materialized view (Sybase Central)

1. Connect to the database as the DBA.
2. Open the Views folder and right-click the materialized view.
3. Select Properties.

   The Properties page for the materialized view displays.

4. On the Miscellaneous tab, remove the checkmark next to View data is encrypted, and then choose OK.

♦ To encrypt a materialized view (SQL)

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

Examples

The following statement encrypts the EmployeeConfidential materialized view. Note that the database must already be configured to allow encrypted tables for this statement to work:

```
ALTER MATERIALIZED VIEW EmployeeConfidential ENCRYPTED;
```

The following statement decrypts the EmployeeConfidential materialized view:

```
ALTER MATERIALIZED VIEW EmployeeConfidential NOT ENCRYPTED;
```

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.

Non-materialized 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, all indexes for it are also dropped and must be recreated, if necessary, when the view is re-enabled.

♦ To enable a materialized view (Sybase Central)

1. Connect to the database as the DBA, or as owner of the materialized view.
2. Open the Views folder for that database.
3. Select the materialized view and then choose File ► Recompile and Enable.

♦ To disable a materialized view (Sybase Central)

1. Connect to the database as the DBA, or as owner of the materialized view.
2. Open the Views folder for that database.
3. Select the materialized view and then choose File ► Disable.

♦ To enable a materialized view (SQL)

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

♦ To disable a materialized view (SQL)

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

Examples

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

```
ALTER MATERIALIZED VIEW EmployeeConfidential DISABLE;
```

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

```
ALTER MATERIALIZED VIEW EmployeeConfidential ENABLE;
REFRESH MATERIALIZED VIEW EmployeeConfidential;
```

See also

- ♦ "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 63
- ♦ "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 494.

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

1. Connect to the database as the DBA.
2. Open the Views folder and right-click the materialized view.
3. Select Properties.

   The Properties page for the materialized view displays.
4. On the General tab, place a checkmark next to Used in optimization, and then choose OK.

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

1. Connect to the database as the DBA.
2. Open the Views folder and right-click the materialized view.
3. Select Properties.

   The Properties page for the materialized view displays.
4. On the General tab, remove the checkmark next to Used in optimization, and then choose OK.

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

1. Connect to the database as the DBA, 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 the DBA, 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 enables the EmployeeConfidential view for use in optimization:

```
ALTER MATERIALIZED VIEW EmployeeConfidential ENABLE USE IN OPTIMIZATION;
```

The following statement disables the EmployeeConfidential view from use in optimization:

```
ALTER MATERIALIZED VIEW EmployeeConfidential DISABLE USE IN OPTIMIZATION;
```

See also

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

Setting the optimizer staleness threshold for materialized views

Data in a materialized view becomes stale when the data changes in the tables referenced by the materialized view. The materialized_view_optimization database option allows you to configure a staleness threshold beyond which the optimizer should no longer consider using it when processing queries. However, if a query references a materialized view directly, the view will be used to process the query, regardless of staleness. Also, the SELECT statement of a query may contain an OPTION clause that overrides the setting of the materialized_view_optimization database option.

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

☞ For information on how to use the OPTION clause of the SELECT statement to override the materialized_view_optimization database option, see "SELECT statement" [*SQL Anywhere Server - SQL Reference*].

☞ For information on using events and triggers, see "Automating Tasks Using Schedules and Events" [*SQL Anywhere Server - Database Administration*].

☞ For information on determining whether the materialized view has been considered by optimizer, see "Reading execution plans" on page 529 and "Monitor query performance" on page 241.

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 SQL statements.

♦ To hide a materialized view (SQL)

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

Example

The following statement hides the EmployeeConfidential materialized view.

```
ALTER MATERIALIZED VIEW EmployeeConfidential SET HIDDEN;
```

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

If you drop a materialized view, all views dependent upon it become INVALID. To use the dependent views, you need to either change the definition for the dependent views, or recreate the dropped materialized view.

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*].

☞ For information on how dropping materialized views affects view alterations, see "View dependencies" on page 63.

♦ To drop a materialized view (Sybase Central)

1. Connect to the database as the DBA, or as the owner of the view.
2. Open the Views folder for that database.
3. Select the materialized view and then choose File ► Delete.

♦ To drop a materialized view (SQL)

1. Connect to the database as the DBA, or as the owner of the view.

2. Execute a DROP MATERIALIZED VIEW statement.

Example

The following statement drops the EmployeeConfidential materialized view.

```
DROP MATERIALIZED VIEW EmployeeConfidential;
```

See also

- ♦ "DROP statement" [*SQL Anywhere Server - SQL Reference*]
- ♦ "sa_dependent_views system procedure" [*SQL Anywhere Server - SQL Reference*]
- ♦ "View dependencies" on page 63

Retrieving information about materialized views

General 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*].

☞ For more information statuses for materialized views, see "View status" on page 64.

Database options information

Using the name of the materialized view, the options that were stored for it at creation time can be obtained from the SYSMVOPTION system view using a query. For example, to find the options for the EmployeeConfidential materialized view, you could execute the following statement:

```
SELECT option_name, option_value
FROM SYSMVOPTION JOIN SYSMVOPTIONNAME
WHERE SYSMVOPTION.view_object_id=(
   SELECT object_id FROM SYSTAB
   WHERE table_name='EmployeeConfidential' )
ORDER BY option_name;
```

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*].

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.

☞ For more information about the Index Consultant, see "Using the Index Consultant" on page 195.

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, say, 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.

☞ For more information about the Index Consultant, see "Using the Index Consultant" on page 195.

Use indexes for frequently-searched columns

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.

☞ For more information about performance, see “Using indexes” on page 251.

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.

SQL Anywhere automatically uses indexes to improve the performance of any database statement whenever it can. There is no need to explicitly refer to indexes once they are created. Also, the index is updated automatically when rows are deleted, updated, or inserted.

Index hints

SQL Anywhere lets you supply index hints, which override the optimizer's choice of query access plan by forcing the use of a particular index. This feature should be used only by advanced users and database administrators. It can lead to poor performance.

☞ To supply an index hint, add a FORCE INDEX or WITH INDEX clause to your query. For more information, see “FROM clause” [*SQL Anywhere Server - SQL Reference*].

☞ For information on altering database object properties, see “Setting properties for database objects” on page 35.

Using clustered indexes

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

You can further improve an index's retrieval performance 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 cannot be 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 ISYSPHYSIDEX 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 clustering property of an index can be added or removed at any time. To reorder the rows in a table to match a clustered index, see the "REORGANIZE TABLE statement" [*SQL Anywhere Server - SQL Reference*].

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

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

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

- The UNLOAD TABLE statement allows you to unload a table in the order of the clustered index key. See "UNLOAD statement" [*SQL Anywhere Server - SQL Reference*].

- The LOAD TABLE statement inserts rows into the table in the order of the clustered index key. See "LOAD TABLE statement" [*SQL Anywhere Server - SQL Reference*].

- The INSERT statement attempts to put new rows on the same table page as the one containing adjacent rows, as per the clustered index key. See "INSERT statement" [*SQL Anywhere Server - SQL Reference*].

- The REORGANIZE TABLE statement restores the clustering of a table by rearranging the rows according to the clustered index. If REORGANIZE TABLE is used with tables where clustering is not specified, the tables are reordered using the primary key. See "REORGANIZE TABLE statement" [*SQL Anywhere Server - SQL Reference*].

Creating 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 for a given table (Sybase Central)

1. Connect to the database as the DBA.

2. Open the Tables folder and select the table for which you want to create an index.
3. In the right pane, click the Indexes tab.
4. From the File menu, choose New ► Index.

   The Create Index wizard appears.
5. Follow the instructions in the wizard.

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

♦ To create a new index for a given table (SQL)

1. Connect to a database 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. For more information, 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);
```

☞ For more information, 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 a 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, open the Indexes folder.

3. Select the desired index and then choose File ▶ Validate.

♦ To validate an index (SQL)

1. Connect to a database 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)

1. Open a command prompt.
2. 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
```

☞ For more information, see "VALIDATE statement" [*SQL Anywhere Server - SQL Reference*], and "The Validation utility" [*SQL Anywhere Server - Database Administration*].

Rebuilding indexes

Sometimes it is necessary to rebuild an index. For example, when an index becomes skewed over time as indicated by the sa_index_density system procedure. 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 all logical indexes that share the same physical index. For more information about logical and physical indexes, see "Index sharing using logical indexes" on page 550.

♦ To rebuild an index (Sybase Central)

1. Connect to a database with DBA authority or as the owner of the table on which the index is created.
2. In the left pane, open the Indexes folder.
3. Select the desired index and then choose File ▶ Rebuild.

♦ To rebuild an index (SQL)

1. Connect to a database 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

- ♦ "ALTER INDEX statement" [*SQL Anywhere Server - SQL Reference*]
- ♦ "sa_index_density system procedure" [*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 a database with DBA authority or as the owner of the table on which the index is created.
2. In the left pane, open the Indexes folder.
3. Select the desired index and then choose Edit ▶ Delete.

♦ To drop an index (SQL)

1. Connect to a database 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;
```

☞ For more information, 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 550.
- ♦ **ISYSPHYSIDX system table** Each row in the ISYSPHYSIDX system table defines a physical index in the database. See "SYSPHYSIDX system view" [*SQL Anywhere Server - SQL Reference*] and "Index sharing using logical indexes" on page 550.

- **ISYSIDXCOL system table** Just as each row in the SYSIDX system view describes one index in the database, each row in the SYSIDXCOL system view describes one column of an index described in the SYSIDX system view. See "SYSIDXCOL system view" [*SQL Anywhere Server - SQL Reference*].
- **ISYSFKEY system table** Every foreign key in the database is defined by one row in the ISYSFKEY system table and one row in the ISYSIDX system table. See "SYSFKEY system view" [*SQL Anywhere Server - SQL Reference*].

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.

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.

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.

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*]

CHAPTER 3

Ensuring Data Integrity

Contents

About this chapter

Building integrity constraints right into the database is the best way to make sure your data stays in good shape. This chapter describes the facilities in SQL Anywhere for ensuring that the data in your database is valid and reliable.

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.

The SQL statements in this chapter use the CREATE TABLE and ALTER TABLE statements, basic forms of which were introduced in "Working with Database Objects" on page 29.

Data integrity overview

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.

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 on procedures and triggers see “Using Procedures, Triggers, and Batches” on page 723.

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 on these and other column defaults, see "Using column defaults" on page 93.

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 723.

As well, column constraints can be inherited from domains. For more information on these and other table and column constraints, see "Using table and column constraints" on page 99.

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 106. For more information about designing appropriate primary and foreign key relations, see "Designing 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 723.

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 sheet.

♦ To display the property sheet for a column

1. Connect to the database.
2. Open the Tables folder for the database.
3. Double-click the table containing the column you want to change.
4. In the right pane, click the Columns tab.
5. Select the desired column.
6. Choose File ► Properties.

   The column's property sheet appears.

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 106).

☞ 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 on IDENTITY columns, see "The special IDENTITY column" on page 577.

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 106). For example, autoincrement default is effective when the column is the first column of an index, because the server uses an index or key definition to find the highest value.

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.

☞ For more information, see the “NEWID function [Miscellaneous]” [*SQL Anywhere Server - SQL Reference*], or the “UNIQUEIDENTIFIER data type” [*SQL Anywhere Server - SQL Reference*].

The NULL default

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

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

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

String and number defaults

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

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

Constant expression defaults

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

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

Using table and column constraints

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

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.

Note
Column CHECK tests fail if the condition returns a value of FALSE. If the condition returns a value of UNKNOWN, the behavior is as though it returns TRUE, and the value is allowed.
For more information about valid conditions, see "Search conditions" [*SQL Anywhere Server - SQL Reference*].

Example 1

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

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

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

Example 2

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

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

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

Example 3

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

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

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

Using CHECK constraints on tables

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

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

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

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

☞ 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 property sheet.

♦ To manage constraints

1. Open the Tables folder.
2. In the right pane, double-click the table you want to alter.
3. On the Constraints tab, make the appropriate changes to the constraint you want to modify.

   For example, to add a table or column constraint, click the Constraints tab and choose File ► New ► Table Check Constraint or File ► New ► Column Check Constraint.

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:

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

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

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

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

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

      ```
      ALTER TABLE table_name
      DELETE CHECK;
      ```

Sybase Central lets you add, alter and drop both table and column CHECK constraints. For more information, see "Working with table and column constraints in Sybase Central" on page 101.

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 desired 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. Select the Domains folder.
2. Choose File ▶ New ▶ Domain.

   The Create Domain wizard appears.
3. Follow the instructions in the wizard.

All domains appear in the Domains folder in Sybase Central.

♦ To assign domains to columns (Sybase Central)

1. For the desired table, click the Columns tab in the right pane.
2. In the data type column for the desired column do one of the following:
   - ♦ Choose the domain from the dropdown list.
   - ♦ Click the button beside the dropdown list and choose the domain on the property sheet.

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.

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

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

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

Example 2: Default values, check constraints, and identifiers

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

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

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

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

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

```
CREATE TABLE 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.

```
CREATE DOMAIN MONEY NUMERIC(19,4)
NULL;
```

☞ For more information, see “CREATE DOMAIN statement” [*SQL Anywhere Server - SQL Reference*].

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. Open the Domains folder.
2. Click the desired domain and choose Edit ▶ Delete.

♦ To drop a domain (SQL)

1. Connect to a database.
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.

☞ For more information about creating primary keys, see "Managing primary keys" on page 48. For the detailed syntax of the CREATE TABLE statement, see "CREATE TABLE statement" [*SQL Anywhere Server - SQL Reference*]. For information about changing table structure, see the "ALTER TABLE statement" [*SQL Anywhere Server - SQL Reference*].

Enforcing referential integrity

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

Example 1

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

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

Example 2

Suppose the database also contained an office table listing office locations. The Employees table might have a foreign key for the office table that indicates which city the employee's office is in. The database designer can choose to leave an office location unassigned at the time the employee is hired, for example, either because they haven't been assigned to an office yet, or because they don't work out of an office. In this case, the foreign key can allow NULL values, and is optional.

Foreign keys enforce referential integrity

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

☞ For more information about creating foreign keys, see "Managing primary keys" on page 48.

Losing referential integrity

Your database can lose referential integrity if someone:

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

SQL Anywhere provides protection against both types of integrity loss.

If a client application breaches referential integrity

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

Example

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

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

Referential integrity actions

Maintaining referential integrity when updating or deleting a referenced primary key can be as simple as disallowing the update or drop. Often, however, it is also possible to take a specific action on each foreign key to maintain referential integrity. The CREATE TABLE and ALTER TABLE statements allow database administrators and table owners to specify what action to take on foreign keys that reference a modified primary key when a breach occurs.

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

- **RESTRICT** Generates an error and prevents the modification if an attempt to alter a referenced primary key value occurs. This is the default referential integrity action.
- **SET NULL** Sets all foreign keys that reference the modified primary key to NULL.
- **SET DEFAULT** Sets all foreign keys that reference the modified primary key to the default value for that column (as specified in the table definition).

- **CASCADE** When used with ON UPDATE, this action updates all foreign keys that reference the updated primary key to the new value. When used with ON DELETE, this action deletes all rows containing foreign keys that reference the deleted primary key.

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

Referential integrity checking

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

Using a database option to control check time

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

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

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

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

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

Integrity rules in the system tables

All the information about database integrity checks and rules is held in the following system tables:

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

CHAPTER 4

Using Transactions and Isolation Levels

Contents

About this chapter

You can group SQL statements into transactions, which have the property that either all statements are executed or none is executed. You should design each transaction to perform a task that changes your database from one consistent state to another.

This chapter describes transactions and how to use them in applications. It also describes how you can set isolation levels in SQL Anywhere to limit the interference among concurrent transactions.

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.

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 114.

Using transactions

SQL Anywhere expects you to group your commands into transactions. Knowing which commands or actions signify the start or end of a transaction lets you take full advantage of this feature.

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 on 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 lets you control when and how transactions from your application terminate:

- 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*].

☞ 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 564.

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:

This isolation level...	Has these characteristics...
0—read uncommitted	♦ Read permitted on row with or without write lock ♦ No read locks are applied ♦ No guarantee that concurrent transaction will not modify row or roll back changes to row ♦ Corresponds to table hints NOLOCK and READUNCOMMITTED
1—read committed	♦ Read only permitted on row with no write lock ♦ Read lock acquired and held for read on current row only, but released when cursor moves off the row ♦ No guarantee that data will not change during transaction ♦ Corresponds to table hint READCOMMITTED
2—repeatable read	♦ Read only permitted on row with no write lock ♦ Read lock acquired as each row in the result set is read, and held until transaction ends ♦ Corresponds to table hint REPEATABLEREAD
3—serializable	♦ Read only permitted on rows in result without write lock ♦ Read locks acquired when cursor is opened and held until transaction ends ♦ Corresponds to table hints HOLDLOCK and SERIALIZABLE
snapshot[1]	♦ No read locks are applied ♦ Read permitted on any row ♦ Database snapshot of committed data is taken when the first row is read or updated by the transaction
statement-snapshot[1]	♦ No read locks are applied ♦ Read permitted on any row ♦ Database snapshot of committed data is taken when the first row is read by the statement

This isolation level...	Has these characteristics...
readonly-statement-snap-shot[1]	♦ No read locks are applied ♦ Read permitted on any row ♦ Database snapshot of committed data is taken when the first row is read by a read-only statement ♦ Uses the isolation level (0, 1, 2, or 3) specified by the updatable_statement_isolation option for an updatable statement

[1] Snapshot isolation must be enabled for the database by setting the allow_snapshot_isolation option to On for the database to use these isolation levels. See "Enabling snapshot isolation" on page 120.

The default isolation level is 0, except for Open Client, jConnect, and TDS connections, which have a default isolation level of 1.

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 inconsistences 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).

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 within a snapshot transaction:

- ALTER INDEX
- ALTER TABLE
- CREATE INDEX
- DROP INDEX
- REFRESH MATERIALIZED VIEW
- REORGANIZE TABLE

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 TABLE 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:

```
SET OPTION PUBLIC.allow_snapshot_isolation = 'On';
            SET TEMPORARY OPTION isolation_level = 'snapshot';
   SELECT * FROM Products; --transaction begins and the statement only

                            --sees changes that are already committed
   INSERT INTO Products
            SELECT ID + 30, Name, Description,
    'Extra large', Color, 50, UnitPrice, NULL
            FROM Products
            WHERE Name = 'Tee Shirt';
COMMIT; --transaction ends
```

Enabling snapshot isolation

Snapshot isolation is enabled or disabled for a database using the allow_snapshot_isolation option. When the option is set to On, row versions are maintained in the temporary file, and connections are allowed to use any of the snapshot isolation levels. When this option is set to Off, any attempt to use snapshot isolation results in an error.

Enabling a database to use snapshot isolation can affect performance because copies of all modified rows must be maintained, regardless of the number of transactions that use snapshot isolation. See "Cursor sensitivity and isolation levels" [*SQL Anywhere Server - Programming*].

The following statement enables snapshot isolation for a database:

```
SET OPTION PUBLIC.allow_snapshot_isolation = 'On';
```

The setting of the allow_snapshot_isolation option can be changed, even when there are users connected to the database. When you change the setting of this option from Off to On, all current transactions must complete before new transactions can use snapshot isolation. When you change the setting of this option from On to Off, all outstanding transactions using snapshot isolation must complete before the database server stops maintaining row version information.

You can view the current snapshot isolation setting for a database by querying the value of the SnapshotIsolationState database property:

```
SELECT DB_PROPERTY ( 'SnapshotIsolationState' );
```

The SnapshotIsolationState property has one of the following values:

- **On** Snapshot isolation is enabled for the database.
- **Off** Snapshot isolation is disabled for the database.
- **in_transition_to_on** Snapshot isolation will be enabled once the current transactions complete.
- **in_transition_to_off** Snapshot isolation will be disabled once the current transactions complete.

When snapshot isolation is enabled for a database, row versions must be maintained for a transaction until the transaction commits or rolls back, even if snapshots are not being used. Therefore, it is best to set the allow_snapshot_isolation option to Off if snapshot isolation is never used.

Snapshot isolation example

The following example uses two connections to the SQL Anywhere sample database to illustrate how snapshot isolation can be used to maintain consistency without blocking.

♦ To use snapshot isolation

1. Execute the following command to create an Interactive SQL connection (Connection1), to the SQL Anywhere sample database:

   ```
   dbisql -c "DSN=SQL Anywhere 10
   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 10
   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;
   ```

ID	Name	Descrip-tion	Size	Color	Quantity	...
300	Tee Shirt	Tank Top	Small	White	28	…
301	Tee Shirt	V-neck	Medium	Orange	54	…
302	Tee Shirt	Crew Neck	One size fits all	Black	75	…
400	Baseball Cap	Cotton Cap	One size fits all	Black	112	…
…	…	…	…	…	…	…

5. In Connection2, execute the following command:

   ```
   UPDATE Products
   SET Name = 'New Tee Shirt'
   WHERE ID = 302;
   ```

6. In Connection1, execute the SELECT statement again:

   ```
   SELECT * FROM Products;
   ```

   The SELECT statement is blocked and cannot proceed because the UPDATE statement in Connection2 has not been committed or rolled back. The SELECT statement must wait until the transaction in Connection2 is complete before it can proceed. This ensures that the SELECT statement does not read uncommitted data into its result.

7. In Connection2, execute the following command:

   ```
   ROLLBACK;
   ```

The transaction in Connection2 completes, and the SELECT statement in Connection1 proceeds.

8. Using the statement snapshot isolation level achieves the same concurrency as isolation level 1, but without blocking.

   In Connection1, execute the following command to allow snapshot isolation:

   ```
   SET OPTION PUBLIC.allow_snapshot_isolation = 'On';
   ```

9. In Connection 1, execute the following command to change the isolation level to statement snapshot:

   ```
   SET TEMPORARY OPTION isolation_level = 'statement-snapshot';
   ```

10. In Connection1, execute the following statement:

    ```
    SELECT * FROM Products;
    ```

11. In Connection2, execute the following statement:

    ```
    UPDATE Products
    SET Name = 'New Tee Shirt'
    WHERE ID = 302;
    ```

12. In Connection1, issue the SELECT statement again:

    ```
    SELECT * FROM Products;
    ```

    The SELECT statement executes without being blocked, but does not include the data from the UPDATE statement executed by Connection2.

13. In Connection2, finish the transaction by executing the following command:

    ```
    COMMIT;
    ```

14. In Connection1, finish the transaction (the query against the Products table), and then execute the SELECT statement again to view the updated data:

    ```
    COMMIT;
    SELECT * FROM Products;
    ```

ID	Name	Descrip-tion	Size	Color	Quantity	...
300	Tee Shirt	Tank Top	Small	White	28	…
301	Tee Shirt	V-neck	Medium	Orange	54	…
302	New Tee Shirt	Crew Neck	One size fits all	Black	75	…
400	Baseball Cap	Cotton Cap	One size fits all	Black	112	…
…	…	…	…	…	…	…

15. Undo the changes to the SQL Anywhere sample database by executing the following statement:

    ```
    UPDATE Products
    SET Name = 'Tee Shirt'
    ```

```
    WHERE id = 302;
    COMMIT;
```

☞ For additional examples about using snapshot isolation, see:

- “Using snapshot isolation to avoid dirty reads” on page 139
- “Using snapshot isolation to avoid non-repeatable reads” on page 145
- “Using snapshot isolation to avoid phantom rows” on page 149

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, an error is given as soon as the update or delete is attempted.

Update conflicts cannot occur when using readonly-statement-snapshot because updatable statements run at a non-snapshot isolation, and always see the most recent version of the database. Therefore, the readonly-statement-snapshot isolation level has many of the benefits of snapshot isolation, without requiring large changes to an application originally designed to run at another isolation level. When using the readonly-statement-snapshot isolation level:

- Read locks are never acquired for read-only statements
- Read-only statements always see a committed state of the database

Typical types of inconsistency

There are three typical types of inconsistency that can occur during the execution of concurrent transactions. This list is not exhaustive as other types of inconsistencies can also occur. These three types are mentioned in the ISO SQL/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 137.

- **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 141.

- **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. Transaction A must then repeat the read and obtains the updated set of rows.

For more information about phantom rows, see “Tutorial: phantom rows” on page 146.

Isolation levels and dirty reads, non-repeatable reads, and phantom rows

SQL Anywhere allows dirty reads, non-repeatable reads, and phantom rows, depending on the isolation level that is used. An X in the following table indicates that the behavior is allowed for that isolation level.

Isolation level	Dirty reads	Non-repeatable reads	Phantom rows
0–read uncommitted	X	X	X
readonly-statement-snapshot	X[1]	X[2]	X[3]
1–read committed		X	X
statement-snapshot		X[2]	X[3]
2–repeatable read			X
3–serializable			
snapshot			

[1] Dirty reads can occur for updatable statements within a transaction if the isolation level specified by the updatable_statement_isolation option does not prevent them from occurring.

[2] Non-repeatable reads can occur for statements within a transaction if the isolation level specified by the updatable_statement_isolation option does not prevent them from occurring. Non-repeatable reads can occur because each statement starts a new snapshot, so one statement may see changes that another statement does not see.

[3] Phantom rows can occur for statements within a transaction if the isolation level specified by the updatable_statement_isolation option does not prevent them from occurring. Phantom rows can occur because each statement starts a new snapshot, so one statement may see changes that another statement does not see.

This table demonstrates two points:

- Each isolation level eliminates one of the three typical types of inconsistencies.
- Each level eliminates the types of inconsistencies eliminated at all lower levels.
- For statement snapshot isolation levels, non-repeatable reads and phantom rows can occur within a transaction, but not within a single statement in a transaction.

The isolation levels have different names under ODBC. These names are based on the names of the inconsistencies that they prevent. See “The ValuePtr parameter” on page 127.

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 on programming SQL procedures and cursors, see "Using Procedures, Triggers, and Batches" on page 723.

☞ 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 the "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 120.

♦ **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 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 128.

Setting the isolation level from an ODBC-enabled application

ODBC applications call SQLSetConnectAttr with Attribute set to SQL_ATTR_TXN_ISOLATION and ValuePtr set according to the corresponding isolation level:

The ValuePtr parameter

ValuePtr	Isolation level
SQL_TXN_READ_UNCOMMITTED	0
SQL_TXN_READ_COMMITTED	1
SQL_TXN_REPEATABLE_READ	2
SQL_TXN_SERIALIZABLE	3
SA_SQL_TXN_SNAPSHOT	snapshot
SA_SQL_TXN_STATEMENT_SNAPSHOT	statement-snapshot
SA_SQL_TXN_READONLY_STATEMENT_SNAPSHOT	readonly-statement-snapshot

Changing an isolation level via ODBC

You can change the isolation level of your connection via ODBC using the function SQLSetConnectOption in the library *ODBC32.dll*.

The SQLSetConnectOption function takes three parameters: the value of the ODBC connection handle, the fact that you want to set the isolation level, and the value corresponding to the isolation level. These values appear in the table below.

String	Value
SQL_TXN_ISOLATION	108
SQL_TXN_READ_UNCOMMITTED	1
SQL_TXN_READ_COMMITTED	2
SQL_TXN_REPEATABLE_READ	4
SQL_TXN_SERIALIZABLE	8
SA_SQL_TXN_SNAPSHOT	32
SA_SQL_TXN_STATEMENT_SNAPSHOT	64
SA_SQL_TXN_READONLY_STATEMENT_SNAPSHOT	128

Do not use the SET OPTION statement to change an isolation level from within an ODBC application. Since the ODBC driver does not parse the statements, execution of any statement in ODBC will not be recognized by the ODBC driver. This could lead to unexpected locking behavior.

Example

The following function call sets the isolation level of the connection MyConnection to isolation level 2:

```
SQLSetConnectOption( MyConnection.hDbc,
                     SQL_TXN_ISOLATION,
                     SQL_TXN_REPEATABLE_READ )
```

ODBC uses the isolation feature to support assorted database lock options. For example, in PowerBuilder you can use the Lock attribute of the transaction object to set the isolation level when you connect to the database. The Lock attribute is a string, and is set as follows:

```
SQLCA.lock = "RU"
```

The Lock option is honored only at the moment the CONNECT occurs. Changes to the Lock attribute after the CONNECT have no effect on the connection.

Changing isolation levels within a transaction

Different isolation levels may be suitable for different parts of a single transaction. SQL Anywhere allows you to change the isolation level of your database in the middle of a transaction.

When you change the isolation_level option in the middle of a transaction, the new setting affects only the following:

- Any cursors opened after the change
- Any statements executed after the change

You may want to change the isolation level during a transaction to control the number of locks your transaction places. You may find a transaction needs to read a large table, but perform detailed work with only a few of the rows. If an inconsistency would not seriously affect your transaction, set the isolation to a low level while you scan the large table to avoid delaying the work of others.

You may also want to change the isolation level mid-transaction if, for example, just one table or group of tables requires serialized access.

For an example in which the isolation level is changed in the middle of a transaction, see "Tutorial: phantom rows" on page 146.

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 the "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:

```
SET OPTION PUBLIC.allow_snapshot_isolation = 'On';

BEGIN TRANSACTION
    SET OPTION isolation_level = 3;
    INSERT INTO Departments
        ( DepartmentID, DepartmentName, DepartmentHeadID )
        VALUES( 700, 'Foreign Sales', 129 );
    SET TEMPORARY OPTION isolation_level = 'snapshot';
    SELECT * FROM Departments;
```

Viewing the isolation level

You can inspect the isolation level of the current connection using the CONNECTION_PROPERTY function.

♦ To view the isolation level for the current connection

- Execute the following statement:

```
SELECT CONNECTION_PROPERTY('isolation_level')
```

Transaction blocking and deadlock

When a transaction is being executed, the database server places locks on rows to prevent other transactions from interfering with the affected rows. **Locks** control the amount and types of interference permitted.

SQL Anywhere uses **transaction blocking** to allow transactions to execute concurrently without interference, or with limited interference. Any transaction can acquire a lock to prevent other concurrent transactions from modifying or even accessing a particular row. This transaction blocking scheme always stops some types of interference. For example, a transaction that is updating a particular row of a table always acquires a lock on that row to ensure that no other transaction can update or delete the same row at the same time.

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 130.

The blocking option

If two transactions have each acquired a read lock on a single row, the behavior when one of them attempts to modify that row depends on the setting of the blocking option. To modify the row, that transaction must block the other, yet it cannot do so while the other transaction has it blocked.

- ♦ If the blocking is option is set to On (the default), then the transaction that attempts to write waits until the other transaction releases its read lock. At that time, the write goes through.
- ♦ If the blocking option has been set to Off, then the statement that attempts to write receives an error.

When the blocking option is set to Off, the statement terminates instead of waiting and any partial changes it has made are rolled back. In this event, try executing the transaction again, later.

Blocking is more likely to occur at higher isolation levels because more locking and more checking is done. Higher isolation levels usually provide less concurrency. How much less depends on the individual natures of the concurrent transactions.

For more information about the blocking option, see “blocking option [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.

SQL Anywhere automatically rolls back the last statement that became blocked (eliminating the deadlock situation), and returns an error to that transaction indicating which form of deadlock occurred.

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*].

The database server provides more detailed deadlock reporting using the log_deadlocks option and sa_report_deadlocks system procedure. When you turn on the log_deadlocks option, the database server records information about the blocked connections in an internal buffer.

♦ To use deadlock reporting

1. Turn on the log_deadlocks option.

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

   The database server records information about deadlocks in an internal buffer.

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

2. Retrieve the deadlock information using sa_report_deadlocks.

   ```
   CALL sa_report_deadlocks()
   ```

   For more information, see "sa_report_deadlocks system procedure" [*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 dialog 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.

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 levels of isolation: 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 117.

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 123 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 indices, 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 a 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 155.

☞ For more information on 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. Connect to the SQL Anywhere sample database as the Sales Manager:
   - In the Connect dialog, choose the SQL Anywhere 10 Demo ODBC data source.
   - To make the window easier to identify, click the Advanced tab, and then type **Sales Manager** in the Connection Name field.
   - Click OK to connect.
3. Start a second instance of Interactive SQL.
4. Connect to the SQL Anywhere sample database as the Accountant:
   - In the Connect dialog, choose the SQL Anywhere 10 Demo ODBC data source.
   - To make the window easier to identify, click the Advanced tab, and then type **Accountant** in the Connection Name field.
   - Click OK to connect.
5. As the Sales Manager, raise the price of all the tee shirts by $0.95:
   - In the window labeled Sales Manager, execute the following commands:

```
UPDATE Products
   SET UnitPrice = UnitPrice + 95
   WHERE Name = 'Tee Shirt';
SELECT ID, Name, UnitPrice
   FROM Products;
```

The result is:

ID	name	UnitPrice
300	Tee Shirt	104.00
301	Tee Shirt	109.00
302	Tee Shirt	109.00
400	Baseball Cap	9.00
…	…	…

You observe immediately that you should have entered 0.95 instead of 95, but before you can fix your error, the Accountant accesses the database from another office.

6. The company's Accountant is worried that too much money is tied up in inventory. As the Accountant, execute the following commands to calculate the total retail value of all the merchandise in stock:

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

   The result is:

Inventory
21453.00

   Unfortunately, this calculation is not accurate. The Sales Manager accidentally raised the price of the 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 116.

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;
   ```

ID	name	UnitPrice
300	Tee Shirt	9.95
301	Tee Shirt	14.95
302	Tee Shirt	14.95
400	Baseball Cap	9.00
…	…	…

8. The Accountant does not know that the amount he calculated was in error. You can see the correct value by executing the SELECT statement again in the Accountant's window.

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

Inventory
6687.15

9. Finish the transaction in the Sales Manager's window. The Sales Manager would enter a COMMIT statement to make the changes permanent, but you should execute a ROLLBACK, instead, to avoid changing the local copy of the SQL Anywhere sample database.

   ```
   ROLLBACK;
   ```

The Accountant unknowingly receives erroneous information from the database because the database server is processing the work of both the Sales Manager and the Accountant concurrently.

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. Connect to the SQL Anywhere sample database as the Sales Manager:
   - In the Connect dialog, choose the SQL Anywhere 10 Demo ODBC data source.
   - On the Advanced tab, type **Sales Manager** in the Connection Name field to make the window easier to identify.
   - Click OK to connect.
3. Execute the following statement to enable snapshot isolation for the database:

   ```
   SET OPTION PUBLIC.allow_snapshot_isolation = 'ON';
   ```

4. Start a second instance of Interactive SQL.

5. Connect to the SQL Anywhere sample database as the Accountant:

   ♦ In the Connect dialog, choose the SQL Anywhere 10 Demo ODBC data source.

   ♦ On the Advanced tab, type **Accountant** in the Connection Name field to make the window easier to identify.

   ♦ Click OK to connect.

6. As the Sales Manager, raise the price of all the tee shirts by $0.95:

   ♦ In the window labeled Sales Manager, execute the following command to :

     ```
     UPDATE Products
     SET UnitPrice = UnitPrice + 0.95
     WHERE Name = 'Tee Shirt';
     ```

   ♦ Calculate the total retail value of all merchandise in stock using the new tee shirt price for the Sales Manager:

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

     The result is:

Inventory
6687.15

7. As the Accountant, execute the following command to calculate the total retail value of all the merchandise in stock. Because this transaction uses the snapshot isolation level, the result is calculated only for data that has been committed to the database.

   ```
   SET OPTION isolation_level = 'Snapshot';
   SELECT SUM( Quantity * UnitPrice )
    AS Inventory
      FROM Products;
   ```

   The result is:

Inventory
6538.00

8. As the Sales Manager, commit your changes to the database by executing the following statement:

   ```
   COMMIT;
   ```

9. As the Accountant, execute the following statements to view the updated retail value of the current inventory:

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

The result is:

Inventory
6687.15

Because the snapshot used for the Accountant's transaction began with the first read operation, you must execute a COMMIT to end the transaction and allow the Accountant to see changes made to the data after the snapshot transaction began. See "Understanding snapshot transactions" on page 119.

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 137 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. Connect to the SQL Anywhere sample database as the Sales Manager:
    - ♦ In the Connect dialog, choose the ODBC data source SQL Anywhere 10 Demo.
    - ♦ To make the window easier to identify, click the Advanced tab, and then type **Sales Manager** in the Connection Name field.
    - ♦ Click OK to connect.
3. Start a second instance of Interactive SQL.
4. Connect to the SQL Anywhere sample database as the Accountant:
    - ♦ In the Connect dialog, choose the ODBC data source SQL Anywhere 10 Demo.
    - ♦ To make the window easier to identify, click the Advanced tab, and then type **Accountant** in the Connection Name field.

- Click OK to connect.

5. Set the isolation level to 1 for the Accountant's connection by executing the following command.

   ```
   SET TEMPORARY OPTION isolation_level = 1
   ```

6. Set the isolation level to 1 in the Sales Manager's window by executing the following command:

   ```
   SET TEMPORARY OPTION isolation_level = 1
   ```

7. The Accountant decides to list the prices of the visors. As the Accountant, execute the following command:

   ```
   SELECT ID, Name, UnitPrice FROM Products
   ```

ID	name	UnitPrice
300	Tee Shirt	9.00
301	Tee Shirt	14.00
302	Tee Shirt	14.00
400	Baseball Cap	9.00
401	Baseball Cap	10.00
500	Visor	7.00
501	Visor	7.00
...	...	...

8. The Sales Manager decides to introduce a new sale price for the plastic visor. As the Sales Manager, execute the following command:

   ```
   SELECT ID, Name, UnitPrice FROM Products
   WHERE Name = 'Visor';
   UPDATE Products
   SET UnitPrice = 5.95 WHERE ID = 501;
   COMMIT;
   SELECT ID, Name, UnitPrice FROM Products
   WHERE Name = 'Visor';
   ```

ID	Name	UnitPrice
500	Visor	7.00
501	Visor	5.95

9. Compare the price of the visor in the Sales Manager window with the price for the same visor in the Accountant window. The Accountant window still displays the old price, even though the Sales Manager has entered the new price and committed the change.

   This inconsistency is called a **non-repeatable read**, because if the Accountant did the same SELECT a second time in the *same transaction*, he wouldn't get the same results. Try it for yourself. As the

Accountant, execute the select command again. Observe that the Sales Manager's sale price now displays.

```
SELECT ID, Name, UnitPrice
FROM Products
```

ID	Name	UnitPrice
300	Tee Shirt	9.00
301	Tee Shirt	14.00
302	Tee Shirt	14.00
400	Baseball Cap	9.00
401	Baseball Cap	10.00
500	Visor	7.00
501	Visor	5.95
…	…	…

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. 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 will start to execute, and then her window will appear 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.

ID	Name	UnitPrice
500	Visor	7.00
501	Visor	7.00

13. The Sales Manager can finish now. She wants to commit her change to restore the original price.

```
COMMIT;
```

Types of locks and different isolation levels

When you upgraded the Accountant's isolation from level 1 to level 2, the database server used read locks where none had previously been acquired. In general, each isolation level is characterized by the types of locks needed and by how locks held by other transactions are treated.

At isolation level 0, the database server needs only write locks. It makes use of these locks to ensure that no two transactions make modifications that conflict. For example, a level 0 transaction acquires a write lock on a row before it updates or deletes it, and inserts any new rows with a write lock already in place.

Level 0 transactions perform no checks on the rows they are reading. For example, when a level 0 transaction reads a row, it 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 130.

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. Connect to the SQL Anywhere sample database as the Sales Manager:
   - ♦ In the Connect dialog, choose the SQL Anywhere 10 Demo ODBC data source.
   - ♦ On the Advanced tab, type **Sales Manager** in the Connection Name field to make the window easier to identify.
   - ♦ Click OK to connect.
3. Start a second instance of Interactive SQL.
4. Connect to the SQL Anywhere sample database as the Accountant:
   - ♦ In the Connect dialog, choose the SQL Anywhere 10 Demo ODBC data source.
   - ♦ On the Advanced tab, type **Accountant** in the Connection Name field to make the window easier to identify.
   - ♦ Click OK to connect.
5. Execute the following statements to enable snapshot isolation for the database and specify that the snapshot isolation level is used:

   ```
   SET OPTION PUBLIC.allow_snapshot_isolation = 'On';
   SET TEMPORARY OPTION isolation_level = snapshot;
   ```

6. The Accountant decides to list the prices of the visors. As the Accountant, execute the following command:

   ```
   SELECT *
   FROM Products
   ORDER BY ID;
   ```

ID	name	UnitPrice
300	Tee Shirt	9.00
301	Tee Shirt	14.00
302	Tee Shirt	14.00

ID	name	UnitPrice
400	Baseball Cap	9.00
401	Baseball Cap	10.00
500	Visor	7.00
501	Visor	7.00
...	...	...

7. The Sales Manager decides to introduce a new sale price for the plastic visor. As the Sales Manager, execute the following command:

```
UPDATE Products
SET UnitPrice = 5.95 WHERE ID = 501;
COMMIT;
SELECT ID, Name, UnitPrice FROM Products
WHERE Name = 'Visor';
```

8. The Accountant executes his query again, and does not see the change in price because the data that was committed at the time of the first read is used for the transaction.

```
SELECT ID, Name, UnitPrice
FROM Products
```

9. As the Sales Manager, change the plastic visor back to its original price.

```
UPDATE Products
SET UnitPrice = 7.00
WHERE ID = 501;
COMMIT;
```

   The database server does not place a read lock on the rows in the Products table that the Accountant is reading because the Accountant is viewing a snapshot of committed data that was taken before the Sales Manager made any changes to the Products table.

10. The Accountant is finished looking at the prices. He doesn't want to risk accidentally changing the database, so he completes his transaction with a ROLLBACK statement.

```
ROLLBACK;
```

Tutorial: phantom rows

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 141, 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;
```

DepartmentID	DepartmentName	DepartmentHeadID
100	R & D	501
200	Sales	902
300	Finance	1293
400	Marketing	1576
500	Shipping	703

4. The Sales Manager decides to set up a new department to focus on the foreign market. Philip Chin, who has EmployeeID 129, will head the new department.

```
INSERT INTO Departments
   (DepartmentID, DepartmentName, DepartmentHeadID)
   VALUES(600, 'Foreign Sales', 129);

COMMIT;
```

The final command creates the new entry for the new department. It appears as a new row at the bottom of the table in the Sales Manager's window.

In the Sales Manager window, enter the following command to list all the departments.

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

DepartmentID	DepartmentName	DepartmentHeadID
100	R & D	501
200	Sales	902
300	Finance	1293
400	Marketing	1576
500	Shipping	703
600	Foreign Sales	129

5. The Accountant, however, is not aware of the new department. At isolation level 2, the database server places locks to ensure that no row changes, but places no locks that stop other transactions from inserting new rows.

   The Accountant will only discover the new row if he executes his SELECT command again. In the Accountant's window, execute the SELECT statement again. You will see the new row appended to the table.

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

DepartmentID	DepartmentName	DepartmentHeadID
100	R & D	501
200	Sales	902
300	Finance	1293
400	Marketing	1576
500	Shipping	703
600	Foreign Sales	129

   The new row that appears is called a **phantom row** because, from the Accountant's point of view, it appears like an apparition, seemingly from nowhere. The Accountant is connected at isolation level 2. At that level, the database server acquires locks only on the rows that he is using. Other rows are left untouched 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 will pause during execution because the Accountant's locks block the command. From the toolbar, click Interrupt the SQL Statement (or choose Stop from the SQL menu) 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 on locking, see "How locking works" on page 155.

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 146 which describe how to start two instances of Interactive SQL.

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

```
SET OPTION PUBLIC. allow_snapshot_isolation = 'On';
SET TEMPORARY OPTION isolation_level = snapshot;
```

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

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

DepartmentID	DepartmentName	DepartmentHeadID
100	R & D	501
200	Sales	902
300	Finance	1293
400	Marketing	1576

DepartmentID	DepartmentName	DepartmentHeadID
500	Shipping	703

3. The Sales Manager decides to set up a new department to focus on the foreign market. Philip Chin, who has EmployeeID 129, will head the new department.

   ```
   INSERT INTO Departments
      (DepartmentID, DepartmentName, DepartmentHeadID)
      VALUES(600, 'Foreign Sales', 129);
   COMMIT;
   ```

   The final command creates the new entry for the new department. It appears as a new row at the bottom of the table in the Sales Manager's window.

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

DepartmentID	DepartmentName	DepartmentHeadID
100	R & D	501
200	Sales	902
300	Finance	1293
400	Marketing	1576
500	Shipping	703
600	Foreign Sales	129

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

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

DepartmentID	DepartmentName	DepartmentHeadID
100	R & D	501
200	Sales	902
300	Finance	1293
400	Marketing	1576
500	Shipping	703

5. The Sales Manager would like to add a second department to handle sales initiative aimed at large corporate partners. Execute the following command in the Sales Manager's window.

```
INSERT INTO Departments
 (DepartmentID, DepartmentName, DepartmentHeadID)
   VALUES(700, 'Major Account Sales', 902);
```

The Sales Manager's change is not blocked because the Accountant is using snapshot isolation.

6. The Accountant must end his snapshot transaction to see the changes the Sales Manager committed to the database.

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

Now the Accountant sees the Foreign Sales department, but not the Major Account Sales department.

DepartmentID	DepartmentName	DepartmentHeadID
100	R & D	501
200	Sales	902
300	Finance	1293
400	Marketing	1576
500	Shipping	703
600	Foreign Sales	129

7. To avoid changing the SQL Anywhere sample database, you should roll back the incomplete transaction that inserts the Major Account Sales department row and use a second transaction to delete the Foreign Sales department.

   a. Execute the following command in the Sales Manager's window to rollback the last, incomplete transaction:

   ```
   ROLLBACK
   ```

   b. Also in the Sales Manager's window, execute the following two statements to delete the row that you inserted earlier and commit this operation.

   ```
   DELETE FROM Departments
   WHERE DepartmentID = 600;

   COMMIT;
   ```

Tutorial: practical locking implications

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 146 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, which 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;
   ```

EmployeeID	GivenName	Surname	April sales
129	Philip	Chin	2160.00
195	Marc	Dill	2568.00
299	Rollin	Overbey	5760.00
467	James	Klobucher	3228.00
…	…	…	…

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.

EmployeeID	GivenName	Surname	April sales
129	Philip	Chin	4560.00
195	Marc	Dill	2568.00
299	Rollin	Overbey	5760.00
467	James	Klobucher	3228.00
…	…	…	…

   Imagine that the Accountant now marks all orders placed in April to indicate that commission has been paid. The order that the Sales Manager just entered might be found in the second search and marked as paid, even though it was not included in Philip's total April sales!

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.

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

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

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

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

```
SELECT EmployeeID, GivenName, Surname,
   SUM(SalesOrderItems.Quantity * UnitPrice)
      AS "April sales"
FROM Employees
   KEY JOIN SalesOrders
   KEY JOIN SalesOrderItems
   KEY JOIN Products
WHERE '2001-04-01' <= OrderDate
   AND OrderDate < '2001-05-01'
GROUP BY  EmployeeID, GivenName, Surname;
```

Because you set the isolation to level 3, the database server automatically places anti-insert locks to ensure that the Sales Manager cannot insert April order items until the Accountant finishes his transaction.

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

   ```
   INSERT INTO SalesOrders
   VALUES ( 2653, 174, '2001-04-22',
            'r1','Central', 129);
   ```

   The Sales Manager's window will stop responding; the operation will not complete. On the toolbar, click Interrupt the SQL Statement (or choose Stop from the SQL menu) 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.

   ```
   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 indices. 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.

    ```
    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 Stop from the SQL menu) 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.

    ```
    ROLLBACK;
    ```

    Enter the same command in the Accountant's window.

    ```
    ROLLBACK;
    ```

You may now close both windows.

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 167.

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 levels of isolation 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 area in Sybase Central. Both methods provide you with 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 a tab named Table Locks appears 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:

```
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 on the sa_locks system procedure, see “sa_locks system procedure” [*SQL Anywhere Server - SQL Reference*].

☞ For information on the NOLOCK table hint, see “FROM clause” [*SQL Anywhere Server - SQL Reference*].

☞ For more information on 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 is locked in shared (read) mode.
- ♦ **Exclusive locks** The table 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.

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.

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 117.

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.

	Shared	**Intent**	**Exclusive**
Shared		conflict	conflict
Intent	conflict		conflict
Exclusive	conflict	conflict	conflict

Shared table locks

A shared table lock can be acquired explicitly through the use of the LOCK TABLE ... IN SHARED MODE statement. When a transaction acquires a shared table lock, that transaction can prevent changes from being made to the table until the transaction completes. Any number of transactions can hold shared table locks on the same table.

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

You can acquire an exclusive table lock explicitly by using of the LOCK TABLE ... IN EXCLUSIVE MODE statement. 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.

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.

	read (R)	**intent (R)**	**write (R)**	**shared (T)**	**intent (T)**	**exclu-sive (T)**	**phantom (P)**	**insert (P)**
read (R)			conflict			conflict		
intent (R)			conflict			conflict		
write (R)		conflict	conflict	conflict		conflict		
shared (T)			conflict		conflict	conflict		
intent (T)				conflict		conflict		
exclusive (T)	conflict	conflict	conflict	conflict	conflict	conflict	conflict	conflict

	read (R)	intent (R)	write (R)	shared (T)	intent (T)	exclu-sive (T)	phantom (P)	insert (P)
phantom (P)						conflict		conflict
insert (P)						conflict	conflict	

Locking during queries

The locks that SQL Anywhere uses when a user enters a SELECT statement depend on the transaction's isolation level. All SELECT statements, regardless of isolation level, acquire schema locks on the referenced tables.

SELECT statements at isolation level 0

No locking operations are required when executing a SELECT statement at isolation level 0. Each transaction is not protected from changes introduced by other transactions. It is 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 uses almost no more locks when running a transaction at isolation level 1 than it does at isolation level 0. Indeed, 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 the READPAST hint for more details.

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 very 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 wish 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 at snapshot isolation

SELECT statements that execute at snapshot, statement-snapshot, or readonly-statement-snapshot do not acquire read locks. This is because each snapshot transaction (or statement) sees a snapshot of a committed state of the database at some previous point in time. The specific point in time is determined by which of the three snapshot isolation levels is being used by the statement. As such, read transactions never block update transactions and update transactions never block readers. Therefore, snapshot isolation can give considerable concurrency benefits in addition to the obvious consistency benefits. However, there is a tradeoff; snapshot isolation can be very expensive. This is because the consistency guarantee of snapshot isolation means that copies of changed rows must be saved, tracked, and (eventually) deleted for other concurrent transactions.

Locking during inserts

INSERT operations create new rows. SQL Anywhere utilizes various types of locks during insertions to ensure data integrity. The following sequence of operations occurs for INSERT statements executing at any isolation level.

1. Acquire a shared schema lock on the table, if one is not already held.
2. Acquire an intent-to-write table lock on the table, if one is not already held.
3. Find an unlocked position in a page to store the new row. To minimize lock contention, the server will not immediately reuse space made available by deleted (but as yet uncommitted) rows. A new page may be allocated to the table (and the database file may grow) to accommodate the new row.
4. Fill the new row with any supplied values.
5. Place an insert lock in the table to which the row is being added. Recall that insert locks are exclusive, so once the insert lock is acquired, no other isolation level 3 transaction can block the insertion by acquiring a phantom lock.
6. Write lock the new row. The insert lock is released once the write lock has been obtained.
7. Insert the row into the table. Other transactions at isolation level 0 can now, for the first time, see that the new row exists. However, these other transactions cannot modify or delete the new row because of the write lock acquired earlier.
8. Update all affected indexes and verify uniqueness where appropriate. Primary key values must be unique. Other columns may also be defined to contain only unique values, and if any such columns exist, uniqueness is verified.
9. If the table is a foreign table, acquire a shared schema lock on the primary table (if not already held), and acquire a read lock on the matching primary row in the primary table if the foreign key column values being inserted are not NULL. The database server must ensure that the primary row still exists when the inserting transaction COMMITs. It does so by acquiring a read lock on the primary row. With

the read lock in place, any other transaction is still free to read that row, but none can delete or update it.

If the corresponding primary row does not exist a referential integrity constraint violation is given.

After step(9) 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.
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 164. 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 Step 7, any AFTER UPDATE triggers may fire. Upon COMMIT, the server will verify referential integrity by ensuring that the number of orphans produced by this transaction is 0, and release all locks.

Modifying a column's 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.

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 will not be 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 will record 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. Indeed, 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. If it were to release a lock sooner, another transaction could modify that row making it impossible to roll back the first transaction.

Read locks are released only in one, special 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 133.

Particular concurrency issues

This section discusses the following particular concurrency issues:

- “Primary key generation” on page 168
- “Data definition statements and concurrency” on page 169

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 will 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 97.

- Use a column with a default value of AUTOINCREMENT.

For example,

```
CREATE TABLE Orders (
   OrderID INTEGER NOT NULL DEFAULT AUTOINCREMENT,
   OrderDate DATE,
   primary key( OrderID )
)
```

On inserts into the table, if a value is not specified for the autoincrement column, a unique value is generated. If a value is specified, it will be used. If the value is larger than the current maximum value for the column, that value will be used as a starting point for subsequent inserts. The value of the most recently inserted row in an autoincrement column is available as the global variable @@identity.

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 [*MobiLink - Server Administration*] and SQL Remote [*SQL Remote*].

Summary

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.

CHAPTER 5

Tutorial: Creating a SQL Anywhere Database

Contents

About this chapter

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 10 - Introduction*].

☞ For information on the principles of database design, see "Designing 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.

♦ To create a new database file

1. Start Sybase Central.
2. From the Tools menu, choose SQL Anywhere 10 ► Create Database.

   The Create Database wizard appears.
3. Read the information on the introductory page, and then click Next.
4. Select Create a Database on This Computer, and then click Next.
5. Choose a location and name for your database file.

   Type the file name *c:\temp\mysample.db*. If your temporary directory is somewhere other than *c:\temp*, specify the appropriate path.
6. Click Finish to create the database.

   Other options are available when creating a database, but the default choices are typically sufficient.

   The Creating Database window displays the progress of the task. A Completed status indicates that the database file has been created.
7. When the status is Completed, click Close.

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 177.

♦ To connect to your database

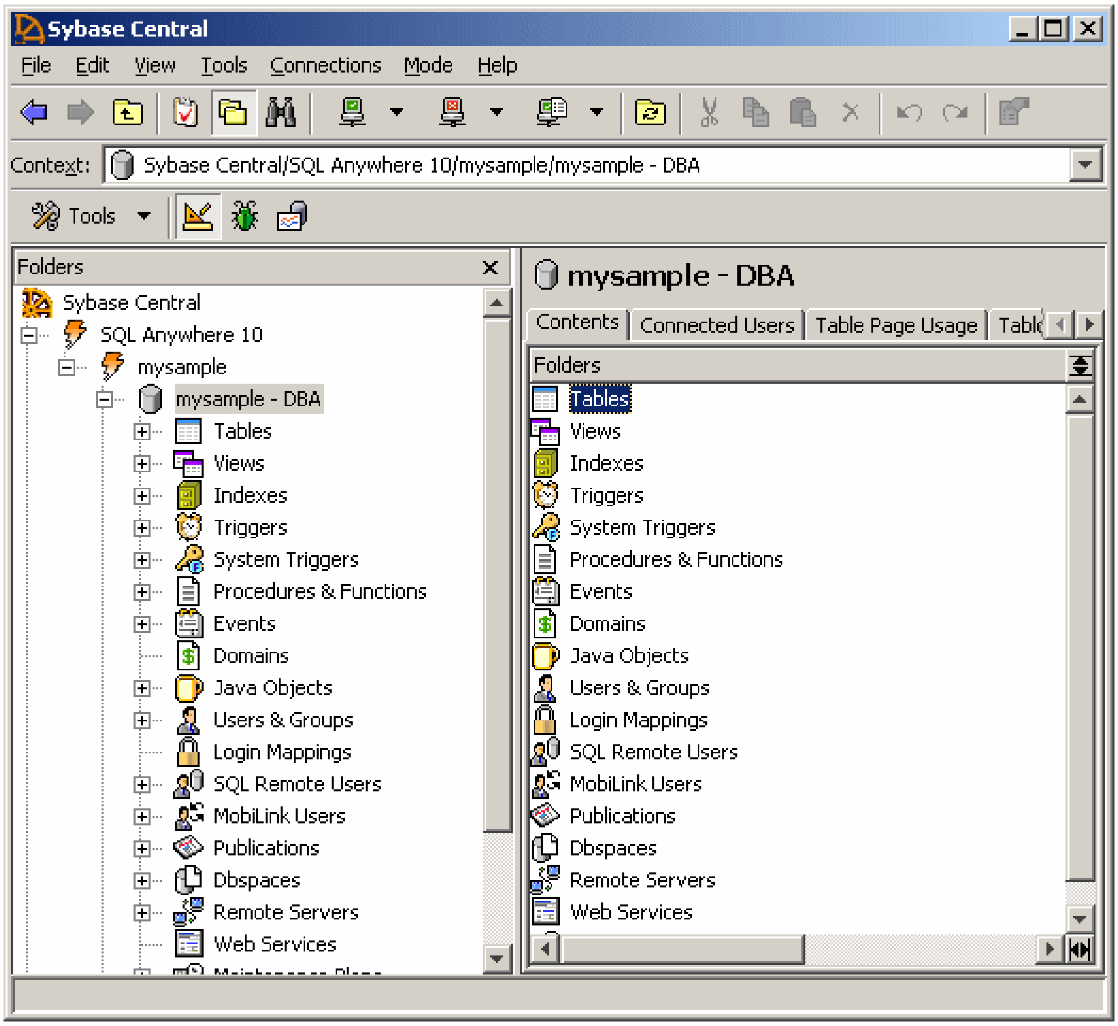
1. Start Sybase Central.
2. From the Connections menu, choose Connect with SQL Anywhere 10.

   The Connect dialog appears.
3. Specify the user ID and password.

   On the Identification tab, enter a User ID of **DBA** and a Password of **sql**. These are the default values for new databases.
4. Select **None** in the profile options at the bottom of the tab.
5. On the Database tab, type the full path of your database file in the Database File field. For example, if you followed the suggestion in the previous lesson, you should type the following:

   ```
   c:\temp\mysample.db
   ```
6. Click OK.

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


Sybase Central
File Edit View Tools Connections Mode Help
Context: Sybase Central/SQL Anywhere 10/mysample/mysample - DBA
Tools
Folders
Sybase Central
SQL Anywhere 10
mysample
mysample - DBA
Tables
Views
Indexes
Triggers
System Triggers
Procedures & Functions
Events
Domains
Java Objects
Users & Groups
Login Mappings
SQL Remote Users
MobiLink Users
Publications
Dbspaces
Remote Servers
Web Services
mysample - DBA
Contents
Connected Users
Table Page Usage
Folders
Tables
Views
Indexes
Triggers
System Triggers
Procedures & Functions
Events
Domains
Java Objects
Users & Groups
Login Mappings
SQL Remote Users
MobiLink Users
Publications
Dbspaces
Remote Servers
Web Services

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 the Tables folder.
2. From the File menu, choose New ► Table.

   The Create Table wizard appears.
3. Name your table **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 Column tab defines a column in the table.
5. Define the first column in your table, and make it the primary key.

   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 10 - Introduction*].

   a. **PKey** Place a checkmark in the PKey column to specify that the column is used as part of the primary key for the table.

   b. **Name** Give the column a name of **ProductID**.

   c. **Data Type** Specify the data type as follows:

      ♦ Give the column the **integer** data type.

        An ellipsis displays in the right half of the Data Type column.

      ♦ Select the ellipsis to access the property sheet for the column.

      ♦ On the Value tab, select Default Value.

      ♦ Select System-defined, and then choose autoincrement from the dropdown list.

   d. Click OK to close the Column property sheet.
6. Create an additional column.

   From the File menu, choose New ► Column, and add a column with the following properties:

   ♦ **Name** Give the column a name of **ProductName**.

   ♦ **Data Type** Give the column the **char** data type.

- **Size** Enter a maximum length of **15** in the Size column.

7. From the File menu, choose Save.

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. Open the Tables folder.
2. Select the Products table in the left pane, and then click the Columns tab in the right pane.
3. Select the ProductID column.
4. From the File menu, choose Properties to open the Column property sheet.
5. On the Constraints tab, 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.
- ♦ **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.

Lesson 5: Create relationships between tables using foreign keys

In this lesson, you learn about creating relationships between tables using foreign keys.

☞ For information about designing relationships, see "Entities and relationships" on page 5.

♦ To create a foreign key

1. In Sybase Central, open the Tables folder.
2. Open the SalesOrders table.
3. In the right pane of Sybase Central, click the Constraints tab.
4. From the File menu, choose New ► Foreign Key to open the Create Foreign Key wizard.
5. Select the **Customers** table as the table to which you want the foreign key to refer, name the new foreign key **CustomerID**, and then click Next.
6. Choose **Primary key** for the foreign key to reference.

   Select **ID** from the Foreign Column dropdown list, and then click Finish.
7. Repeat steps 1 through 5 to create the following foreign keys:

   - ♦ A foreign key from the ID column in SalesOrderItems, referencing the ID column in SalesOrders.
   - ♦ A foreign key from the ProductID column in SalesOrderItems, referencing the ProductID column in Products.
   - ♦ A foreign key from the CustomerID column in SalesOrders, referencing the ID column in Customers.

This completes this introductory section on creating relational databases.

☞ For more information on designing databases, see "Entities and relationships" on page 5.

Summary

In this chapter, you applied principles of database design to create a new database using Sybase Central.

Part II. Monitoring and Improving Database Performance

This part describes performance analysis, improvements, and monitoring.

CHAPTER 6

Monitoring and Improving Performance

Contents

About this chapter

This chapter describes how to monitor and improve the performance of your database.

Overview

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 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 200.
- **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 215.
- **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 195.
- **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 189.

  ☞ For more information on how to perform procedure profiling tasks using commands, see "Procedure profiling using system procedures" on page 220.

Note
In the documentation, the terms *application profiling* and *diagnostic tracing* are often used interchangeably. This is because they are essentially the same, with diagnostic tracing being a more advanced approach to 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 200.

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.

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.

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 on the database files created for application profiling and diagnostic tracing, see "Tracing session data" on page 200.

The Application Profiling wizard cannot be used to create a tracing session for a database running on Windows CE. You must use the Database Tracing wizard. See "Creating a tracing session" on page 210.

♦ To use the Application Profiling wizard in Sybase Central

1. Connect to the database as the DBA.
2. Choose Mode ► Application Profiling.
   - ♦ If the Application Profiling wizard appears, follow the wizard instructions.
   - ♦ If the wizard does not appear, choose Application Profiling ► Open Application Profiling wizard, and then follow the wizard 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.
3. 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 on the indexing recommendations returned from the Application Profiling wizard, see "Understanding Index Consultant recommendations" on page 196.

☞ For more information on the procedure profiling information gathered during the tracing session, see "How to read procedure profiling results" on page 192.

Procedure profiling in Application Profiling mode

Procedure profiling shows you how long it takes your procedures, user-defined functions, events, system triggers, and triggers to execute. You can also view line-by-line execution times for these objects, once they have run during profiling. Then, using the information provided in the procedure profiling results, you can determine which objects should be fine-tuned to improve performance within your database.

Procedure profiling can also help you analyze specific database procedures (including stored procedures, functions, events and triggers) found to be expensive via request logging. It can also help you discover expensive hidden procedures, for example, triggers, events, and nested stored procedure calls. As well, it can help pin-point potential problem areas within the body of a procedure.

Procedure profiling results are stored in memory by the database server and can be accessed either via Application Profiling mode in Sybase Central (the recommended method), or using functions and system procedures. Profiling information is cumulative, and accurate to 1 ms. This section explains how to perform procedure profiling via Application Profiling mode.

You can also perform procedure profiling using SQL commands. See "Procedure profiling using system procedures" on page 220.

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

1. Connect to your database as the DBA.
2. Choose Mode ▶ Application Profiling.
   - ♦ If the Application Profiling wizard appears, follow the wizard instructions.
   - ♦ If the wizard does not appear, choose Application Profiling ▶ Open Application Profiling wizard, and then follow the wizard instructions.
3. On the Profiling Options page of the Application Profiling wizard, select Stored Procedure, Function, Trigger or Event Execution Time. Do not select any of the other options.
4. Select Finish.

   The database server begins procedure profiling. If you attempt to switch to another mode, the database server asks whether you want to continue procedure profiling. Select No to continue working in other modes while profiling continues.

See also

- ♦ "Resetting procedure profiling" on page 190
- ♦ "Disabling procedure profiling" on page 191
- ♦ "Analyzing procedure profiling results" on page 192

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

1. Switch to Application Profiling mode. If the Application Profiling wizard appears, click Cancel to close it.
2. Open the Database property sheet for the database.
   - ♦ If procedure profiling is enabled: in the Application Profiling Details pane at the bottom of Sybase Central, click the desired database to select it, and then click View Profiling Settings on Selected Databases.

- ♦ If procedure profiling is not enabled: In the Folders pane on the left side of Sybase Central, right-click the desired database, and choose Properties from the popup menu.

   The Database property sheet appears.

3. On the Profiling Settings tab, click Reset Now.

   The database server clears any existing profiling information.

4. Click OK to close the property sheet.

See also

- ♦ "Enabling procedure profiling" on page 190
- ♦ "Disabling procedure profiling" on page 191
- ♦ "Analyzing procedure profiling results" on page 192

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

1. Switch to Application Profiling mode. If the Application Profiling wizard appears, click Cancel to close it.
2. Select the database in the Application Profiling Details area (lower pane), and click Stop Collecting Profiling Information on Selected Databases.

   The database server stops collecting profiling information.

♦ To delete profiling information and disable procedure profiling

1. Switch to Application Profiling mode. If the Application Profiling wizard appears, click Cancel to close it.
2. In the Application Profiling Details pane, select the desired database, and then click View Profiling Settings on Selected Databases.

   The Database property sheet appears.

3. On the Profiling Settings tab, click Clear Now.

   Procedure profiling is disabled (notice that the checkmark is cleared from Capture and View Profiling Information in this Database for) and profiling data is removed from the database.

4. Click OK to close the property sheet.

See also

- ♦ “Enabling procedure profiling” on page 190
- ♦ “Resetting procedure profiling” on page 190
- ♦ “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 in Sybase Central

1. If you have not already done so, connect to the database as user DBA, and enable procedure profiling. See “Enabling procedure profiling” on page 190.

2. In the left pane, select the type of object for which you would like to see profiling information. Your choice must be one of the following: Triggers, System Triggers, Procedures & Functions, or Events.

   A list of all objects in the database for that type appears in the right pane. For example, if you selected Procedures & Functions, the list of all procedures and user-defined functions in the database appears.

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. For example, if you selected Procedures & Functions, a list of all of the procedures and user-defined functions that have executed since you enabled procedure profiling displays.

   If you expected to find an object there but it is missing, for example an event, it may be because it hasn't executed yet. Or, it may have executed but the view has not yet been refreshed. While details are refreshed periodically, you can cause a refresh at any time by pressing F5.

   If you find more objects listed than you expected, remember that 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, for example, line-by-line execution details, 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:

Column	Description
Name	The name of the object.
Owner	The owner of the object.

Column	Description
Table or Table Name	The table a trigger belongs to (this column only appears on the database Profile tab).
Event	The type of object, for example, a procedure.
Type	The type of trigger for system triggers. This can be Update or Delete.
# Execs.	The number times each object has been called.
# msecs.	The total execution time for each object.

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

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

Column	Description
Execs	The number of times the line of code in the object was executed.
Milliseconds	The total amount of time that a line took to execute.
%	The percent of total time that a line took to execute.
Line	The line number within the object.
Source	The code that was executed.

Lines with long execution times compared to other lines in the code should be analyzed to see whether there is a more efficient way to achieve the same functionality. You must be connected to the database, have profiling enabled, and have DBA authority to access procedure profiling information.

Baselining using a profiling log file

In Sybase Central, you can save procedure profiling results to file to use as baseline information. For example, if you are making incremental changes to a procedure to see if it runs faster, you can run it after each change you make, and compare the results to the results that were saved to file.

In order to understand how to use baselining with procedure profiling in Sybase Central, you must first be familiar with the results provided for procedure profiling. See "Analyzing procedure profiling results" on page 192.

♦ To perform baselining using a profiling log file

1. Enable procedure profiling on the database you wish to profile.

   a. Connect to your database as the DBA.

   b. Choose Mode ► Application Profiling.

- ♦ If the Application Profiling wizard appears, follow the wizard instructions.
- ♦ If the wizard does not appear, choose Application Profiling ► Open Application Profiling wizard, and then follow the wizard instructions.

c. On the Profiling Options page of the Application Profiling wizard, select Stored Procedure, Function, Trigger or Event Execution Time. Do not select any of the other options.

d. Select Finish.

The database server begins procedure profiling. If you attempt to switch to another mode, the database server asks whether you want to continue procedure profiling. Select No to continue working in other modes while profiling continues.

2. Right-click the procedure in the Procedures & Functions folder and choose Execute From Interactive SQL.

   The database server executes the procedure from Interactive SQL. Since procedure profiling is enabled, execution details for the procedure are captured.

3. Close Interactive SQL.

4. Save the profiling results.

   a. Right-click the database and choose Properties.

      The Database property sheet for the database 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 choose a location and file name for the profiling log file.

   d. Click Apply.

      The procedure profiling information gathered from the time procedure profiling was enabled is saved to the specified profiling log file (*.plg*).

5. Enable baselining against the profiling log file.

   a. On the Profiles Settings tab of the Database property sheet, select Use the Profiling Information in the Following Profiling Log File as a Baseline for Comparison.

   b. Browse to and select the profiling log file you created.

   c. Click Apply.

   d. Click OK to close the Database property sheet.

6. Make the desired changes to the procedure.

   a. In the left pane, browse to, and select, the procedure in the Procedures & Functions folder.

   b. On the SQL tab in the right pane, make the desired changes to the SQL code for the procedure.

   c. Choose File ► Save.

7. Right-click the procedure and choose Execute Fom Interactive SQL.

   The database server executes the procedure from Interactive SQL.

8. Close Interactive SQL.

9. In the right pane in Sybase Central, click the Profiling Results tab to view execution details.

   Notice that there are two new columns: Execs. +/-, and ms. +/-. These columns result from comparing statistics in the profiling log file to the statistics captured during the most recent execution of 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 have improved 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. For example, -3 indicates that the line of code in the procedure ran 3 milliseconds faster than the baseline.

Index Consultant

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 unclustered 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.

☞ For information on how to use the Index Consultant, see “Using the Index Consultant” on page 195.

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 on how to locate and use Index Consultant in Sybase Central when connecting to a version 9 database server.

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

1. In Interactive SQL, type the query in the SQL Statements pane.
2. Choose Tools ▶ Index Consultant.

   The Index Consultant performs its analysis and returns recommendations.

Obtaining index recommendations for a database

To obtain Index Consultant recommendations for an entire database, use the Application Profiling wizard in Sybase Central. This is the easiest way to obtain Index Consultant recommendations.

♦ To obtain Index Consultant recommendations for a database

1. Connect to the database you want Index Consultant to analyze as the DBA.
2. Choose Mode ▶ Application Profiling.
3. If the Application Profiling wizard does not automatically start, choose Application Profiling ▶ Open Application Profiling wizard.
4. Follow the instructions in the wizard, including running your applications for a period of time in order to capture tracing information.
5. When you are done capturing data, click Finish in the Application Profiling wizard.
6. Choose Application Profiling ▶ Run Index Consultant on Tracing Database.

   The Index Consultant appears.
7. Follow the instructions provided in Index Consultant.

   The Index Consultant analyzes the tracing data and returns its recommendations. For more information about Index Consultant recommendations, see "Understanding Index Consultant recommendations" on page 196.

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.

☞ For more information about clustered indexes, see "Using clustered indexes" on page 81.

- **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 unclustered index.

  ☞ For more information about clustered indexes, see "Using clustered indexes" on page 81.

- **Pages** The estimated number of database pages required to hold the index if you choose to create it.

  ☞ For more information about database page sizes, see "The Initialization utility" [*SQL Anywhere Server - Database Administration*].

- **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 198.

- **Total Benefit** The cost decrease associated with the index, summed over all operations in the tracing session, measured in internal units of cost.

  ☞ For more information on the cost model, see “Optimizer estimates and column statistics” on page 488.

- **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.

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

- “Using SQL command files” on page 668
- “REORGANIZE TABLE statement” [*SQL Anywhere Server - SQL Reference*]

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 timestamps 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
- circumstances that cause resource contention (CPUs, Disk I/O, and memory)
- 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. For more information about application profiling using tracing data, see “Application profiling” on page 188.

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 215.

Tracing session data

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

1. using the Database Tracing wizard in Sybase Central
2. transparently, as part of the automated activities of the Application Profiling wizard
3. 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 on 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.

Note
When you create a tracing session for a database running on Windows CE, you must use the Database Tracing wizard (you cannot use the Application Profiling wizard). As well, you must trace from the Windows CE device to a copy of the Windows CE database running on a database server on a desktop computer. You cannot automatically create a tracing database from a Windows CE device, and you cannot trace to the local database on a CE device.

The tables that hold the tracing data are referred to as the **diagnostic tracing tables**. These tables are owned by dbo. For more information on these tables, see "Diagnostic tracing tables" [*SQL Anywhere Server - SQL Reference*].

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 a 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 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 in the production (local) database, or in an external tracing database (recommended). You are also given an opportunity to create an external tracing database, if you do not already have one, using the wizard. An external tracing database has the extension *.db*.

☞ For information on how to create an external tracing database, see "Creating a separate tracing database" on page 216.

Configuring diagnostic tracing

Tracing settings are stored in the sa_diagnostic_tracing_level system table. See "sa_diagnostic_tracing_level table" [*SQL Anywhere Server - SQL Reference*].

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.
- using system procedures to change settings stored in the diagnostic tracing tables. For more information on 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*].

Note
The Application Profiling wizard in Sybase Central uses pre-configured tracing settings, and is not impacted by the tracing settings you specify when using the Database Tracing wizard.

The SendingTracingTo and ReceivingTracingFrom database properties identify the tracing and production databases, respectively. For more information on these properties, see "Database-level properties" [*SQL Anywhere Server - Database Administration*].

Choosing a tracing level

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 **tracing types**. Following are descriptions of the levels you can specify, and the tracing types they include. For a description of the tracing types mentioned below, see "Diagnostic tracing types" on page 205.

Customizing tracing settings allows you to reduce the amount of unwanted tracing data in the 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 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 tracing level to 3, and then customize the scope of the tracing to be USER, and specify AliceB as the user name. Allow the tracing session to run for a couple of hours, and then examine the results.

The recommended method for customize tracing settings is using the Database Tracing wizard. See "Changing 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 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 tracing settings while a tracing session is in progress" on page 209.

Diagnostic tracing levels

The following is a list of tracing levels and the types of information (tracing types) that they include. These levels reflect the settings specified in the Database Tracing wizard. For a description of the various tracing types, see "Diagnostic tracing types" on page 205.

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 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 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 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 a column name or database name, which are stored in the identifier column. The values in the scope column reflect the settings specified in the Database Tracing wizard.

Values in the scope column	Description
DATABASE	Records tracing data for any event occurring within the database, assuming the event corresponds to the specified level and condition. Used for long-term background monitoring of the database, or for short-term diagnostics, when it is necessary to determine the source of costly queries. There is no identifier to specify when you specify DATABASE.
ORIGIN	Records tracing data for the queries originating from either outside or inside the database. There are two possible identifiers you can specify when specifying the scope ORIGIN: External or Internal. External specifies to log the statement text and associated details for queries that come from outside the database server, and that correspond to the specified level and condition. Internal specifies to log the same information for queries that come from within the database server, and that correspond to the level and condition specified.
USER	Records tracing data only for the queries issued by the specified user, and by connections created by the specified user. This scope is used to diagnose problematic queries originating from a particular user. The identifier for this scope is the name of the user for whom the tracing is to be performed.
CONNECTION_NAME, or CONNECTION_NUMBER	Records tracing data only for the statements executed by the current connection. These scopes are used when the user has multiple connections, one of which is executing costly statements. The identifier for this scope is the name of the connection, or the connection number, respectively.
FUNCTION, PROCEDURE, EVENT, TRIGGER, or TABLE	Records tracing data for the statements that use the specified object. If the object references other objects, all the data for those objects is recorded as well. For example, if tracing is being done for a procedure that uses a function which, in turn, triggers an event, statements for all three objects are logged, providing they correspond to the specified level and condition provided for logging. Used when use of a specific object is costly, or when the statements that reference the object take an unusually long time to finish. The TABLE scope is used for tables, materialized views, and non-materialized views. The identifier for this scope is the fully qualified name of the object.

See also

- "Diagnostic tracing types" on page 205
- "Diagnostic tracing conditions" on page 207

Diagnostic tracing types

Following is the list of **tracing types** you can set for diagnostic tracing. Tracing types control the type of information for which to generate tracing data. Each trace type requires a corresponding condition, as noted below. Trace type values are stored in the trace_type column of the dbo.sa_diagnostic_tracing_level diagnostic table, and may have corresponding 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.

Value in the trace_type column	Description
VOLATILE_STATISTICS	Periodically collects a sample of frequently changing database and server statistics at regular intervals. Scopes and conditions: This trace type requires the DATABASE scope, and the SAMPLE_EVERY condition.
NONVOLATILE_STATISTICS	Periodically collects a sample of database and server statistics that do not change frequently. Non-volatile statistics cannot be collected more frequently then volatile statistics. Volatile statistics must be collected in order for non-volatile statistics to be collected, and the time difference between the sampling for non-volatile statistics should be a multiple of the time difference specified for the volatile statistics. Scopes and conditions: This trace type requires the DATABASE scope, and takes the SAMPLE_EVERY condition.
CONNECTION_STATISTICS	Periodically 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 trace type can be used with the DATABASE, USER, CONNECTION_NUMBER, and CONNECTION_NAME scopes, and takes the SAMPLE_EVERY condition.
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 trace type can be used with all of the scopes, and takes any one of the following conditions: NONE, NULL, SAMPLE_EVERY.

Value in the trace_type column	Description
PLANS	Collects execution plans for queries, depending on the condition and scope. Scopes and conditions: This trace type can be used with all of the scopes, and takes any one of the following conditions: NONE, NULL, SAMPLE_EVERY, and ABSOLUTE_COST.
PLANS_WITH_STATISTICS	Collects plans with execution statistics. Plans are recorded at cursor close time. If the RELATIVE_COST_DIFFERENCE condition is specified, part of the statistics in the output might be best-guess statistics. Scopes and conditions: This trace type can be used with all of the scopes, and takes any one of the conditions.
STATEMENTS	Collects SQL statements for the specified scope and condition. Internal variables are collected the first time each procedure is executed. This trace type is automatically included if the STATEMENTS_WITH_VARIABLES, PLANS, PLANS_WITH_STATISTICS, OPTIMIZATION_LOGGING, or OPTIMIZATION_LOGGING_WITH_PLANS tracing type is specified. Scopes and conditions: This trace type can be used with all of the scopes, and takes any one of the conditions.
STATEMENTS_WITH_VARIABLES	Collects SQL statements and the variables attached to the statements. For each variable, either internal or host, all the values that were assigned are collected as well. Scopes and conditions: This trace type can be used with all of the scopes, and takes any one of the conditions.
OPTIMIZATION_LOGGING	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 trace type is automatically added whenever the OPTIMIZATION_LOGGING_WITH_PLANS trace type is specified. This trace type corresponds to all of the scopes, and does not take a condition.
OPTIMIZATION_LOGGING_WITH_PLANS	Collects data about join strategies considered by the optimizer. Information about the cost of execution for each strategy, as well as the complete XML plan describing the join strategy tree structure, is collected. Information about rewrites applied to the query is also collected. If a scope other then DATABASE, CONNECTION_NAME, CONNECTION_NUMBER, ORIGIN, or USER is used, the first recorded statement text might be different then the initial text of the query since some rewrites can be applied before it can be determined that optimization logging should be applied to the current statement. The OPTIMIZATION_LOGGING trace type is automatically added whenever the OPTIMIZATION_LOGGING_WITH_PLANS trace type is specified. This trace type corresponds to all of the scopes, and does not take a condition.

See also

- “Diagnostic tracing scopes” on page 204
- “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 trace type. Most conditions require a value, as noted below. Conditions are stored in the trace_condition column of the dbo.sa_diagnostic_tracing_level diagnostic table, and may have a corresponding value, such as an amount of time in milliseconds, stored in the value column. The values in the condition column reflect the settings specified in the Database Tracing wizard.

Value in the trace_condition column	Description
NONE, or NULL	Records all of the tracing data that satisfies the level and scope requirements. Using expensive tracing levels (plans, for example) with this condition for extended time periods is not recommended.
SAMPLE_EVERY	Records tracing data that satisfies the level and scope requirements if more than the specified time interval has elapsed since the last event was recorded. Values: This condition takes a positive integer, reflecting time in milliseconds.
ABSOLUTE_COST	Records the statements with cost of execution greater than, or equal to, the specified value. Values: This condition takes a cost value, specified in milliseconds.
RELATIVE_COST_DIFFERENCE	Records the statements for which the difference between the expected time for execution and the real time for execution is greater than or equal to the specified value. Values: This condition takes a cost value specified as a percentage. For example, to log statements that are at least twice as slow as estimated, specify a value of 200.

See also

- “Diagnostic tracing scopes” on page 204
- “Diagnostic tracing types” on page 205

Determining current tracing settings

To see the current tracing settings, you start the Database Tracing wizard in Sybase Central. When you are done examining the settings, cancel the wizard. You can also retrieve the tracing settings in effect by querying the sa_diagnostic_tracing_level table.

You can retrieve tracing settings regardless of whether a tracing session is in progress.

♦ To determine the current tracing settings (Sybase Central)

1. Connect to the database as the DBA.
2. Choose Mode ▶ Application Profiling.

   If the Application Profiling wizard appears, click Cancel to close it.
3. Right-click the database, and choose Tracing from the popup menu.

   If this is your first time using tracing, or if you have recently cleared your tracing settings, the Database Tracing wizard appears. Otherwise, you must choose Tracing ▶ Configure and start tracing.
4. Advance in the wizard until you reach the Edit Tracing Levels page. This page contains the settings currently specified for tracing, even if a tracing session is not in progress.
5. Click Cancel to leave the wizard once you are done reviewing the settings.

♦ To determine the current tracing settings (Interactive SQL)

1. Connect to the database as the DBA.
2. Query the sa_diagnostic_tracing_level table for rows in which the enabled column contains a 1.

   The database server returns the tracing settings currently in use. A 1 in the enabled column indicates that the setting is in effect.

Example

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

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

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

id	scope	identifier	trace_type	trace_condition	value	enabled
1	database	(NULL)	volatile_statistics	sample_every	1,000	1
2	database	(NULL)	nonvolatile_statistics	sample_every	60.000	1
3	database	(NULL)	connection_statistics	(NULL)	60,000	1
4	database	(NULL)	blocking	(NULL)	(NULL)	1
5	database	(NULL)	deadlock	(NULL)	(NULL)	1
6	database	(NULL)	plans_with_statistics	sample_every	2,000	1

See also

- "sa_diagnostic_tracing_level table" [*SQL Anywhere Server - SQL Reference*]

Changing tracing configuration settings

Tracing settings are production-database-specific. That is, the tracing settings you configure for a production database do not impact tracing that you perform on another production database. You use the Database Tracing wizard in Sybase Central to change tracing settings when creating a tracing session. To learn how to start the Database Tracing wizard, see "Creating a tracing session" on page 210.

Tracing settings configured in the Database Tracing wizard do not affect settings or behavior for the Application Profiling wizard.

You can also use the sa_set_tracing_level system procedure to change the 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 tracing level (Interactive SQL)

1. Connect to the database as the DBA.
2. Use the sa_set_tracing_level system procedure to set the desired tracing levels.

Example

The following statement uses the sa_set_tracing_level system procedure to set the tracing level to 1:

```
CALL sa_set_tracing_level( 1 );
```

Existing tracing settings are overwritten with the default tracing settings associated with tracing level 1. To see the default settings associated with the various tracing levels, see "Diagnostic tracing levels" on page 203.

Changing tracing settings while a tracing session is in progress

You can change tracing settings while a tracing session is in progress using the Database Tracing wizard in Sybase Central.

♦ To change tracing settings during a tracing session (Sybase Central)

1. Connect to the database as the DBA.
2. From the Context dropdown list, choose the database.
3. Choose File ► Tracing ► Change Tracing Levels.
4. Change the tracing settings, and then click OK.

   The database server overwrites existing diagnostic tracing settings with the new settings.

Creating a tracing session

When you start a 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.

In order 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 ATTACH TRACING and DETACH TRACING, respectively.

♦ To create a tracing session (Sybase Central)

1. Connect to the database you want to profile.
2. Choose Mode ▶ Application Profiling.

   If the Application Profiling wizard appears, click Cancel to close it.
3. Select the database, and choose File ▶ Tracing. If a submenu appears, choose Configure and Start Tracing.

   The Database Tracing wizard appears.
4. Follow the instructions in the Database Tracing wizard to configure and capture a tracing session:

   a. On the Tracing Detail Level page, select the level of tracing, and narrow the tracing scope, if needed.

   b. On the Edit Tracing Levels page, customize tracing settings as required.

   c. On the Create External Database page:

      - Select Create a New Tracing Database.
      - Specify the user name and password that you used to connect to the production database.
      - Select Start Database on the Current Server.
      - Click Create Database.

        The Database Tracing wizard creates the tracing database by unloading the schema and permission information from the production database, and then loading it into the newly-created tracing database.
      - Click Next.

   d. On the Start Tracing page:

- Select Save Tracing Data in an External Database.
- Specify the user name and password that you specified for the tracing database. These should match the user name and password you used to connect to the production database.
- Specify the database server and database name in the form of a partial connect string. For example:

```
ENG=Server47;DBN=TracingDB
```

e. Under Do You Want to Limit the Volume of Trace Data That Is Stored, choose an option for the volume of tracing database to save.

f. Click Finish.

5. Allow applications to interact with the database for a period of time to allow data to be gathered.
6. When you are done gathering tracing data, select the database and choose File ► Tracing ► Stop Tracing with Save.

   The Database Tracing wizard ends the tracing session and stores the captured data in the diagnostic tables.

♦ To create a tracing session (Interactive SQL)

1. Connect to the database as the DBA.
2. Use the sa_set_tracing_level system procedure to set the desired 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 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=dbsrv10;DBN=tracing;LINKS=tcpip' LIMIT HISTORY 2 HOURS;
```

This example shows how to start 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 tracing and save the diagnostic data that was captured during the tracing session:

```
DETACH TRACING WITH SAVE;
```

This example shows how to stop 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*]

Analyzing diagnostic tracing information

Diagnostic tracing data provides a record of all activities that took place on the database server corresponding to the 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.

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 view 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.

Detecting slow statements

You can identify which statements the database server spends the most time processing by using the Summary and Detail views. The Summary view groups similar statements together and reports 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 view). A statement may appear expensive in the summary view because it is an intrinsically expensive statement, or because, although a cheap statement by itself, it is frequently executed.

The Details view shows each statement that was captured as part of the tracing session, along with the time it took to execute it. The duration shown for each statement is the time the database server spent actually processing the request. A cursor may be left open for a long time (as shown by the presence of a long interval between the start time and the cursor close time), but the duration may be short if the database server was only asked to produce a small result set for this cursor.

You can right-click a statement for more details. The full SQL text of the statement appears, along with details about when it was used and in what context. Note that the text displayed for the statement may not match the original text. If this statement was captured after it was parsed by the database server (either because it is part of a compiled database object, such as a stored procedure or trigger), or because it was only selected for inclusion in the trace due to satisfying a sampling or cost condition, it may appear different than it was originally written. In particular, queries over views may appear drastically different, since the view definitions are often flattened into the queries (expanded inline).

If the statement was a query, the More Details dialog also shows details about the plan used to execute the query. Note that the text representation of the query plan is always captured and always accurately represents the actual access plan used by the database server. Depending on the settings selected for plans, the graphical plan shown may not be the actual plan used by the database server. If the description on the graphical plan is Best Guess Plan, then the database server has re-optimized the query, simulating as much as possible the database server conditions when it was first optimized. Usually, this causes the same plan to be selected. However, you should check that the guess plan matches the actual text plan before relying on it. See "Reading execution plans" on page 529.

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 on how to solve 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 228

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 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 779.

Investigating deadlocks

You may also need to investigate deadlocks. Deadlocks occur when two or more connections are waiting for each other to finish, and thus neither is able to finish. If deadlocks occurred during your tracing session, you can investigate them in Application Profiling mode. To do this, open the tracing session, and view the Deadlocks tab in the Application Profiling Details area (lower pane) in Sybase Central.

The Deadlock tab shows a graphical representation of each deadlock that was recorded during the tracing session. You can examine the connections that participated and the statements that each connection blocked while trying to execute. If you traced to the local database, you can also view the primary keys for the rows for which locks could not be acquired.

☞ For more information about deadlocks and what causes them to occur, see "Transaction blocking and deadlock" on page 130.

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 your database as the DBA.
2. Choose Mode ▶ Application Profiling.

   If the Application Profiling wizard appears, click Cancel to close it.
3. Select the database you are going to analyze, and choose File ▶ Tracing. If this is the first time you are using the Database Tracing wizard, or if you previously cleared the tracing configuration settings, the wizard starts automatically. If you are presented with a submenu, choose Configure and Start Tracing.
4. Follow the instructions in the Database Tracing wizard to configure and capture a tracing session. In particular, narrow the Scope to be the users and/or connections for which problems are occurring. The Edit Tracing Levels page provides you with the ability to customize all criteria for the diagnostic data gathered.
5. Once you are finished configuring tracing details, click Finish.

   A tracing session is now in progress. Allow applications to interact with the database for a period of time to allow data to be gathered.

6. When you are done gathering tracing data, select the database and choose File ► Tracing ► Stop Tracing With Save.

   This terminates the tracing session and stores the captured data in for the tracing session.

7. Perform the analysis.

   a. In the Application Profiling Details pane, click Open an Analysis File or Connect to a Tracing Database.

      The Open Analysis or Connect to Tracing Database dialog appears.

   b. Choose In a Tracing Database, and click Open.

      The Connect to a Tracing Database dialog appears.

   c. Specify the user name and password you used to connect to the database you are analyzing, and then click OK.

      The Application Profiling Details pane displays information about the tracing database and the tracing sessions stored in it.

   d. From the Logging Session ID dropdown, select the last entry in the list (this is the last session captured).

      Details for the tracing session appear in the Application Profiling Details pane.

   e. Click the Database Tracing Data tab at the bottom of the Application Profiling Details pane. You can now 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 use the SQL Statements Containing field to enter a string to search for in the list.

   f. To view more details about a specific request, right-click the request and choose Show the Detailed SQL Statements for the Selected Summary Statement. This moves you to the Details tab, where more information is presented. Right-click the row containing the request, and additional choices for information are provided, including viewing additional SQL statement, connection, and blocking details.

Creating a separate 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 a separate tracing database, in order to have a minimal impact on performance.

The easiest method for creating a separate 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 on how to create a tracing session, see "Creating a 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 a separate tracing database using the Unload utility (dbunload)

1. Connect to the database as the DBA.
2. Execute a dbunload command, similar to the following, to unload the schema from the production database into the new tracing database:

   ```
   dbunload -c "UID=DBA;PWD=sql;ENG=sample;DBN=sample" -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

♦ "The Unload utility" [*SQL Anywhere Server - Database Administration*]

Other diagnostic tools and techniques

In addition to application profiling and diagnostic tracing, a variety of other diagnostic tools and techniques are available to help you analyze and monitor the current performance of your SQL Anywhere database.

Request logging

Request logging logs individual requests received from and responses sent to an application. It is most useful for determining what the database server is being asked to do by the application.

Request logging is also a good starting point for performance analysis of a specific application when it is not obvious whether the database server or the client is at fault. You can use request logging to determine the specific request to the database server that might be responsible for problems.

Note
All 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 200.

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 on 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.

☞ For more information on these system procedures, 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.

☞ For more information on the sa_server_option system procedure, see "sa_server_option system procedure" [*SQL Anywhere Server - SQL Reference*].

♦ To filter according to a connection

- Use the following syntax:

```
CALL sa_server_option( 'RequestFilterConn' , connection-id );
```

You can obtain *connection-id* by executing `CALL sa_conn_info( );`.

♦ To filter according to a database

- Use the following syntax:

```
CALL sa_server_option( 'RequestFilterDB' , database-id );
```

The *database-id* can be obtained by executing `SELECT CONNECTION_PROPERTY ( 'DBNumber' )` when connected to that database. Filtering remains in effect until explicitly reset, or until the database server is shut down.

♦ To reset filtering

- Use either of the following two statements to reset filtering either by connection or by database:

```
CALL sa_server_option( 'RequestFilterConn' , -1 );

CALL sa_server_option( 'RequestFilterDB' , -1 );
```

Outputting host variables to request logs

Host variable values can be output to a request log.

♦ **To include host variable values**

- To include host variable values in the request log:
  - ♦ use the -zr server option with a value of **hostvars**
  - ♦ execute the following:

```
CALL sa_server_option( 'RequestLogging' , 'hostvars' );
```

The request log analysis procedure, sa_get_request_times, recognizes host variables in the log and adds them to the global temporary table satmp_request_hostvar.

Procedure profiling using system procedures

Procedure profiling provides valuable informatoin 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 189.

Enabling profiling using sa_server_option

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

☞ For more information on the syntax of, and results returned by, this system procedure, see “sa_server_option system procedure” [*SQL Anywhere Server - SQL Reference*].

You must have DBA authority to enable and use procedure profiling.

♦ **To enable procedure profiling in Interactive SQL**

1. Connect to your database as the DBA.
2. Call the sa_server_option system procedure, setting the ProcedureProfiling option to ON.

   For example, enter:

```
CALL sa_server_option( 'ProcedureProfiling' , 'ON' );
```

If necessary, you can see what procedures a specific user is using, without preventing other connections from using the database. This is useful if the connection already exists, or if multiple users connect with the same user ID.

♦ To filter procedure profiling by user in Interactive SQL

1. Connect to the database as the DBA.
2. Call the sa_server_option system procedure as follows:

   ```
   CALL sa_server_option( 'ProfileFilterUser' , 'userid' );
   ```

   The value of *userid* is the name of the user being monitored.

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.

☞ For more information on the syntax of, and results returned by, this system procedure, see "sa_server_option system procedure" [*SQL Anywhere Server - SQL Reference*].

The following sections assume that you are already connected to your database as the DBA and that procedure profiling is enabled.

♦ To reset profiling in Interactive SQL

- Call the sa_server_option system procedure, setting the ProcedureProfiling option to RESET.

  For example, enter:

  ```
  CALL sa_server_option( 'ProcedureProfiling' , 'RESET' );
  ```

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.

☞ For more information on the syntax of, and results returned by, this system procedure, see "sa_server_option system procedure" [*SQL Anywhere Server - SQL Reference*].

♦ To disable profiling (Interactive SQL)

- Call the sa_server_option system procedure, setting the ProcedureProfiling option to OFF.

  For example, enter:

  ```
  CALL sa_server_option( 'ProcedureProfiling' , 'OFF' );
  ```

♦ To disable profiling and clear existing data (Interactive SQL)

- Call the sa_server_option system procedure, setting the ProcedureProfiling option to CLEAR.

  For example, enter:

  ```
  CALL sa_server_option( 'ProcedureProfiling' , 'CLEAR' );
  ```

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. You must be connected to the database as the DBA. Also, procedure profiling must already be enabled. See "Enabling profiling using sa_server_option" on page 220.

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.

☞ For information on the syntax for, and the results returned by, these system procedures, see "sa_procedure_profile_summary system procedure" [*SQL Anywhere Server - SQL Reference*], and "sa_procedure_profile system procedure" [*SQL Anywhere Server - SQL Reference*].

♦ To view summary profiling information (Interactive SQL)

1. Execute the sa_procedure_profile_summary system procedure.

   For example, enter:

   ```
   CALL sa_procedure_profile_summary;
   ```

2. 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. Execute the sa_procedure_profile system procedure.

   For example, enter:

   ```
   CALL sa_procedure_profile;
   ```

2. Choose SQL ▶ Execute.

   A result set with profiling information appears in the Results pane.

Graphical plans

The graphical plan feature in Interactive SQL displays the execution plan for a query. It is useful for diagnosing performance issues with specific queries. For example, the information in the plan can help you decide where to add an index to your database. You can save the graphical plan for a query for future reference by choosing File ► Save Plan in Interactive SQL. SQL Anywhere graphical plans are saved with the extension *.saplan*.

Note
Graphical plans are also available via Application Profiling mode in Sybase Central. For more information about the Application Profiling features of Sybase Central, see "Application profiling" on page 188.

The graphical plan provides a great deal more information than the short or long plans. You can choose to see the graphical plan either with or without statistics. Both allow you to view quickly which parts of the plan have been estimated as the most expensive. The graphical plan with statistics, though more expensive to view, also 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. Note, however, that the optimizer is often unable to precisely estimate a query's cost, so expect there to be differences. The graphical plan is the default format for access plans.

You can obtain detailed information about the nodes in the plan by clicking the node in the graphical diagram. The graphical plan with statistics shows you all the estimates that are provided with the graphical plan, but also shows actual runtime costs of executing the statement. To do this, the statement must actually be executed. This means that there may be a delay in accessing the plan for expensive queries. It also means that any parts of your query such as deletes or updates are actually executed, although you can perform a rollback to undo these changes.

Use the graphical plan with statistics when you are having performance problems, and the estimated row count or run time differs from your expectations. The graphical plan with statistics provides estimates and actual statistics for you to compare. A large difference between actual and estimate is a warning sign that the optimizer might not have sufficient information to prepare correct estimates.

Following are some of the key statistics you can check in the graphical plan with statistics, and some possible remedies:

- Row count measures the rows in the result set. If the estimated row count is significantly different from the actual row count, the selectivity of underlying predicates is probably incorrect.
- 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 (with, say, a selectivity of 5%), but in reality, the predicate is much less selective (for example, 50%), then performance may suffer. In general, estimates may not be precise. However, a significantly large error does indicate a possible problem. If the predicate is over a base column for which there does not exist a histogram, executing a CREATE STATISTICS statement to create a histogram may correct the problem. If selectivity error remains a problem then, as a last resort, you may want to consider specifying a user estimate of selectivity along with the predicate in the query text.

- Runtime measures the time to execute the query. If the runtime is incorrect 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.
- When the source of the estimate is Guess, the optimizer has no information to use, which may indicate a problem. If the estimate source is Index and the selectivity estimate is incorrect, your problem may be that the index is skewed: you may benefit from defragmenting the index with the REORGANIZE INDEX statement.
- If the number of cache reads and cache hits are exactly the same, then your entire database is in cache —an excellent thing. When reads are greater than hits, it means that the database server is attempting to go to cache but failing, and that it must read from disk. In some cases, such as hash joins, this is expected. In other cases, such as nested loops joins, a poor cache-hit ratio may indicate a performance problem, and you may benefit from increasing your cache size.

See also

- "Graphical plans" on page 539
- "Accessing the execution plan" on page 546
- "Reading execution plans" on page 529

Timing utilities

Some performance testing utilities, including fetchtst, instest, and trantest, are available in *samples-dir \SQLAnywhere*. For more information about the location of *samples-dir*, see "The samples directory" [*SQL Anywhere Server - Database Administration*].

The fetchtst utility measures fetch rates for an arbitrary query. The instest utility determines the time required for rows to be inserted into a table. The trantest utility measures the load that can be handled by a given server configuration given a database design and a set of transactions.

These tools give you more accurate timings than the graphical plan with statistics, and can provide an indication of the best achievable performance (for example, throughput) for a given server and database configuration.

Complete documentation for the tools can be found in the *readme.txt* file in the same folder as the utility.

Monitoring database performance

SQL Anywhere provides a set of statistics you can use to monitor database performance. There are three ways to access these statistics:

- **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 225.
- **Windows Performance Monitor** This is a monitoring tool provided by your Windows operating system. See "Monitoring statistics using Windows Performance Monitor" on page 227.
- **SQL Anywhere Console utility (dbconsole)** The utility provides administration and monitoring facilities for database server connections. See "The SQL Anywhere Console utility" [*SQL Anywhere Server - Database Administration*].
- **SQL functions** These functions allow your application to access SQL Anywhere database statistics directly. See "Monitoring statistics using SQL functions" on page 236.

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 on diagnostic tracing, see "Advanced application profiling using diagnostic tracing" on page 200.

☞ For a complete listing of the SQL Anywhere statistics available for monitoring, see "Performance Monitor statistics" on page 228.

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 227.

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 228.

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 on adding and removing statistics from the Performance Monitor graph, see "Adding and removing statistics" on page 226.

♦ To open the Performance Monitor

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

See also

- "Monitoring statistics using Windows Performance Monitor" on page 227
- "Adding and removing statistics" on page 226

Adding and removing statistics

♦ To add statistics to the Sybase Central Performance Monitor

1. In the left pane of Sybase Central, select the desired server.
2. In the right pane, click the Statistics tab.
3. Right-click a statistic that is not currently being graphed and choose Add to Performance Monitor from the popup menu.

♦ To remove statistics from the Sybase Central Performance Monitor

1. In the left pane of Sybase Central, select the desired server.
2. In the right pane, click the Statistics tab.
3. Right-click a statistic that currently being graphed and choose Remove from Performance Monitor from the popup menu.

Tip
You can also add a statistic to or remove one from the Performance Monitor on the statistic's property sheet.

☞ For a complete listing of the SQL Anywhere statistics available for monitoring, see "Performance Monitor statistics" on page 228.

See also

- ♦ "Opening the Sybase Central Performance Monitor" on page 226
- ♦ "Monitoring statistics using Windows Performance Monitor" on page 227

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 XP/200x. 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 228.

♦ To use the Windows Performance Monitor

1. With a SQL Anywhere server running, start the Performance Monitor:

   - ♦ Choose Administrative Tools from the Windows Control Panel.
   - ♦ Choose Performance to start the Windows Performance Monitor.

2. Choose the Plus sign tool (+) on the toolbar. The Add Counters dialog appears.
3. From the Performance Object list, select one of the following:

   - ♦ **SQL Anywhere 10 Connection** This monitors performance for a single connection. A connection must currently exist in order to see this selection.
   - ♦ **SQL Anywhere 10 Database** This monitors performance for a single database.
   - ♦ **SQL Anywhere 10 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. From 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 10 Connection or SQL Anywhere 10 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 per second. The statistics are grouped into the following areas:

- “Cache statistics” on page 228
- “Checkpoint and recovery statistics” on page 229
- “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 232
- “Memory pages statistics” on page 234
- “Request statistics” on page 235
- “Miscellaneous statistics” on page 235

Cache statistics

These statistics describe the use of the cache.

Statistic	Scope	Description
Cache Hits/sec	Connection and database	Shows the rate at which database page lookups are satisfied by finding the page in the cache.
Cache Reads: Index Interior/sec	Connection and database	Shows the rate at which index internal-node pages are read from the cache.
Cache Reads: Index Leaf/sec	Connection and database	Shows the rate at which index leaf pages are read from the cache.
Cache Reads: Table/sec	Connection and database	Shows the rate at which table pages are read from the cache.
Cache Reads: Total Pages/sec	Connection and database	Shows the rate at which database pages are looked up in the cache.
Cache Replacements: Total Pages/sec	Server	Shows the rate at which database pages are being purged from the cache to make room for another page that is needed.
Cache Size: Current	Server	Shows the current size of the database server cache, in kilobytes.
Cache Size: Maximum	Server	Shows the maximum allowed size of the database server cache, in kilobytes.

Statistic	Scope	Description
Cache Size: Minimum	Server	Shows the minimum allowed size of the database server cache, in kilobytes.
Cache Size: Peak	Server	Shows the peak size of the database server cache, in kilobytes.

Checkpoint and recovery statistics

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

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

Statistic	Scope	Description
Idle Checkpoint Time	Database	Shows the total time spent doing idle checkpoints, in seconds.
Idle Checkpoints/sec	Database	Shows the rate at which checkpoints are completed by the database server's idle thread. An idle checkpoint occurs whenever the idle thread writes out the last dirty page in the cache.
Idle Writes/sec	Database	Shows the rate at which disk writes are issued by the database server's idle thread.
Recovery I/O Estimate	Database	Shows the estimated number of I/O operations required to recover the database.
Recovery Urgency	Database	Shows the recovery urgency expressed as a percentage.

Communications statistics

These statistics describe client/server communications activity.

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

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.

Statistic	Scope	Description
Disk: Active I/Os	Database	Shows the current number of file I/Os issued by the database server which have not yet completed.
Disk: Maximum Active I/Os	Database	Shows the maximum value "Disk: Active I/Os" has reached.

Disk read statistics

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

Statistic	Scope	Description
Disk Reads: Total Pages/sec	Connection and database	Shows the rate at which pages are read from a file.
Disk Reads: Active	Database	Shows the current number of file reads issued by the database server which haven't yet completed.
Disk Reads: Index interior/sec	Connection and database	Shows the rate at which index internal-node pages are being read from disk.

Statistic	Scope	Description
Disk Reads: Index leaf/sec	Connection and database	Shows the rate at which index leaf pages are being read from disk.
Disk Reads: Table/sec	Connection and database	Shows the rate at which table pages are being read from disk.
Disk Reads: Maximum Active	Database	Shows the maximum value "Disk Reads: Active" has reached.

Disk write statistics

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

Statistic	Scope	Description
Disk Writes: Active	Database	Shows the current number of file writes issued by the database server which aren't yet completed.
Disk Writes: Maximum Active	Database	Shows the maximum value "Disk Writes: Active" has reached.
Disk Writes: Commit Files/sec	Database	Shows the rate at which the database server forces a flush of the disk cache. Windows XP/200x and NetWare platforms use unbuffered (direct) I/O, so the disk cache doesn't need to be flushed.
Disk Writes: Database Extends/sec	Database	Shows the rate at which the database file is extended, in pages/sec.
Disk Writes: Temp Extends/sec	Database	Shows the rate at which temporary files are extended, in pages/sec.
Disk Writes: Pages/sec	Connection and database	Shows the rate at which modified pages are being written to disk.
Disk Writes: Transaction Log/sec	Connection and database	Shows the rate at which pages are written to the transaction log.
Translog Group Commits	Connection and database	Shows the rate at which a commit of the transaction log was requested but the log had already been written (so the commit was done for free).

Index statistics

These statistics describe the use of the index.

Statistic	Scope	Description
Index: Adds/sec	Connection and database	Shows the rate at which entries are added to indexes.

Statistic	Scope	Description
Index: Lookups/sec	Connection and database	Shows the rate at which entries are looked up in indexes.
Index: Full Compares/sec	Connection and database	Shows the rate at which comparisons beyond the hash value in an index must be performed.

Memory diagnostic statistics

These statistics describe how the database server is using memory.

Statistic	Scope	Description
Cache: Multi-Page Allocations	Server	Shows the number of multi-page allocations.
Cache: Panics	Server	Shows the number of times the cache manager has failed to find a page to allocate.
Cache: Scavenge Visited	Server	Shows the number of pages visited while scavenging for a page to allocate.
Cache: Scavenges	Server	Shows the number of times the cache manager has scavenged for a page to allocate.
Cache Pages: Allocated Structures	Server	Shows the number of cache pages that have been allocated for database server data structures.
Cache Pages: File	Server	Shows the number of cache pages used to hold data from database files.
Cache Pages: File Dirty	Server	Shows the number of cache pages that are dirty (needing a write).
Cache Pages: Free	Server	Shows the number of cache pages not being used.
Cache Pages: Pinned	Server	Shows the number of pages currently unavailable for reuse.
Cache Replacements: Total Pages/sec	Server	Shows the rate at which database pages are being purged from the cache to make room for another page that is needed.
Heaps: Carver	Connection and server	Shows the number of heaps used for short-term purposes such as query optimization..
Heaps: Query Processing	Connection and server	Shows the number of heaps used for query processing (hash and sort operations).
Heaps: Relocatable	Connection and server	Shows the number of relocatable heaps.
Heaps: Relocatable Locked	Connection and server	Shows the number of relocatable heaps currently locked in the cache.

Statistic	Scope	Description
Map physical memory/sec	Server	Shows the rate at which database page address space windows are being mapped to physical memory in the cache using Address Windowing Extensions.
Mem Pages: Carver	Connection and server	Shows the number of heap pages used for short-term purposes such as query optimization.
Mem Pages: Pinned Cursor	Server	Shows the number of pages used to keep cursor heaps pinned in memory.
Mem Pages: Query Processing	Connection and server	Shows the number of cache pages used for query processing (hash and sort operations).

Memory pages statistics

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

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

Request statistics

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

Statistic	Scope	Description
Cursors	Connection	Shows the number of declared cursors currently maintained by the database server.
Cursors Open	Connection	Shows the number of open cursors currently maintained by the database server.
Lock Count	Connection and database	Shows the number of locks.
Requests	Server	Shows the rate at which the database server is entered to allow it to handle a new request or continue processing an existing request.
Requests: Active	Server	Shows the number of database server threads that are currently handling a request.
Requests: Exchange	Server	Shows the number of database server threads that are currently being used for parallel execution of a query.
Requests: Unscheduled	Server	Shows the number of requests that are currently queued up waiting for an available database server thread.
Snapshot Count	Connection and database	Shows the number of active snapshots.
Statements	Connection	Shows the number of prepared statements currently maintained by the database server.
Statement Prepares	Connection	Shows the rate at which statement prepares are being handled by the database server.
Transaction Commits	Connection	Shows the rate at which Commit requests are handled.
Transaction Rollbacks	Connection	Shows the rate at which Rollback requests are handled.

Miscellaneous statistics

Statistic	Scope	Description
Avail IO	Server	Shows the current number of available I/O control blocks.
Connection Count	Database	Shows the number of connections to this database.

Statistic	Scope	Description
Main Heap Bytes	Server	Shows the number of bytes used for global database server data structures.
Query: Plan cache pages	Connection and database	Shows the number of pages used to cache execution plans.
Query: Low memory strategies	Connection and database	Shows the number of times the database server changed its execution plan during execution because of low memory conditions.
Query: Rows material-ized/sec	Connection and database	Shows the rate at which rows are written to work tables during query processing.
Requests: GET DATA/ sec	Connection and database	Shows the rate at which a connection is issuing GET DATA requests.
Temporary Table Pages	Connection and database	Shows the number of pages in the temporary file used for temporary tables.
Version Store Pages	Database	Shows the number of pages of the temporary file currently being used for the row version store when snapshot isolation is enabled.

Monitoring statistics using SQL functions

SQL Anywhere provides a set of system functions that can access information on a per-connection, per-database, or server-wide basis. The kind of information available ranges from static information (such as the database server name) to detailed performance-related statistics (such as disk and memory usage).

Functions that retrieve system information

The following functions retrieve system information:

- **PROPERTY function** This function provides the value of a given property on a server-wide basis. See “PROPERTY function [System]” [*SQL Anywhere Server - SQL Reference*]
- **DB_PROPERTY and DB_EXTENDED_PROPERTY functions** These functions provide the value of a given property for a given database, or by default, for the current database. See “DB_PROPERTY function [System]” [*SQL Anywhere Server - SQL Reference*], and “DB_EXTENDED_PROPERTY function [System]” [*SQL Anywhere Server - SQL Reference*].
- **CONNECTION_PROPERTY and CONNECTION_EXTENDED_PROPERTY functions** These functions provide the value of a given property for a given connection, or by default, for the current connection. See “CONNECTION_PROPERTY function [System]” [*SQL Anywhere Server - SQL Reference*], and “CONNECTION_EXTENDED_PROPERTY function [String]” [*SQL Anywhere Server - SQL Reference*].

Supply as an argument only the name of the property you want to retrieve. The functions return the value for the current server, connection, or database.

☞ For a complete list of the properties available from the system functions, see “System functions” [*SQL Anywhere Server - SQL Reference*].

Examples

The following statement sets a variable named server_name to the name of the current server:

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

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

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

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

```
SELECT DB_PROPERTY( 'file' );
```

Improving query efficiency

For better performance, a client application monitoring database activity should use the PROPERTY_NUMBER function to identify a named property, and then use the number to repeatedly retrieve the statistic.

Property names obtained in this way are available for many different database statistics, from the number of transaction log page write operations and the number of checkpoints performed, to the number of reads of index leaf pages from the memory cache.

The following set of statements illustrates the process from Interactive SQL:

```
CREATE VARIABLE propnum INT ;
CREATE VARIABLE propval INT ;
SET propnum = PROPERTY_NUMBER( 'CacheRead' );
SET propval = DB_PROPERTY( propnum );
```

☞ For more information on 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 tool. See “Monitoring statistics using Sybase Central Performance Monitor” on page 225.

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, make sure your server meets the minimum CPU, memory, and disk requirements:

- ♦ SQL Anywhere can run with as little as 4 MB of memory. If you are using the administration tools, such as Sybase Central and Interactive SQL, SQL Anywhere requires at least 32 MB of RAM. Your computer must have this much memory in addition to the requirements for the operating system.
- ♦ Enough disk space to hold your database and log files.
- ♦ Keep in mind that these are the minimums. If you are meeting only the minimum hardware requirements, and find that performance is suffering, consider upgrading some or all of 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

You might think that SQL Anywhere would run faster without a transaction log because it would have to maintain less information on disk. Yet, the opposite is actually true. Not only does a transaction log provide a large amount of protection, it can dramatically improve performance.

When operating without a transaction log, SQL Anywhere must perform a checkpoint at the end of every transaction. Writing these changes consumes considerable resources.

With a transaction log, however, SQL Anywhere need only write 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.

Tip
Always use a transaction log. It helps protect your data and it greatly improves performance.

If you can store the transaction log on a different physical device than the one containing the main database file, you can further improve performance. 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 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 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 priority

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 pick 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.

You can use the FASTFIRSTROW table hint in a query's FROM clause to set the optimization goal for a specific query to First-row, without having to change the optimization_goal setting.

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*]

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.

☞ For more information, see "Using table and column constraints" on page 99.

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.

On Novell NetWare, the size of the cache is set when you launch the database server. Be sure to allocate as much memory to the database cache as possible, given the requirements of the other applications and processes that run concurrently.

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.

☞ For more information, see "Use the cache to improve performance" on page 252.

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 "Data integrity overview" on page 90.

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 on the location of *samples-dir*, see "The samples directory" [*SQL Anywhere Server - Database Administration*].

☞ For information about system procedures that measure query execution times, see "sa_get_request_profile system procedure" [*SQL Anywhere Server - SQL Reference*] and "sa_get_request_times system procedure" [*SQL Anywhere Server - SQL Reference*].

fetchtst

Function Determines the time required for a result set to be retrieved.

Location *samples-dir\SQLAnywhere\PerformanceFetch*

odbcfet

Function Determines the time required for a result set to be retrieved. This tool is similar to fetchtst, but with less functionality.

Location *samples-dir\SQLAnywhere\PerformanceFetch*

instest

Function Determines the time required for rows to be inserted into a table.

Location *samples-dir\SQLAnywhere\PerformanceInsert*

trantest

Function Measures the load that can be handled by a given database server configuration given a database design and a set of transactions.

Location *samples-dir\SQLAnywhere\PerformanceTransaction*

Normalize your table structure

One or more database tables may contain multiple copies of the same information (for example, a column that is repeated in several tables), and your table may need to be normalized.

Normalization reduces duplication in a relational database. For example, suppose 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.

You can, however, take normalization too far. 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 "Step 3: Normalize the data" on page 15.

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.

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 almost always improve database performance when you put different physical database files on different physical devices. For example, while one disk drive is busy swapping database pages to and from the cache, another device can be writing to the log file.

Notice that to gain these benefits, the devices must be independent. A single disk, partitioned into smaller logical drives, is unlikely to yield benefits.

SQL Anywhere uses four types of files:

1. database (*.db*)
2. transaction log (*.log*)
3. transaction log mirror (*.mlg*)
4. temporary file (*.tmp*)

The **database file** holds the entire contents of your database. A single file can contain a single database, or you can add up to 12 dbspaces, which are additional files holding portions of the same database. You choose a location for the database file, 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 "Change Log File Settings wizard" [*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 (for all connections except shared memory connections on Unix). 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 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. This can be accomplished either by changing all CREATE TABLE statements to specify the dbspace, or by changing the setting of the default_dbspace option before creating any tables. See "default_dbspace option [database]" [*SQL Anywhere Server - Database Administration*], and "CREATE TABLE statement" [*SQL Anywhere Server - SQL Reference*].

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 258
- "Backup and Data Recovery" [*SQL Anywhere Server - Database Administration*]
- "The Transaction Log utility" [*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 655.

Reduce fragmentation

Fragmentation occurs naturally as you make changes to your database. Performance can suffer if your files, tables, or indexes are excessively fragmented. This 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 XP/200x. The database server displays the following information in the 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.

☞ For more information, 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 may 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' ] ] );
```

☞ See “sa_table_fragmentation system procedure” [*SQL Anywhere Server - SQL Reference*].

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*]
- “The Rebuild utility” [*SQL Anywhere Server - Database Administration*]

Reducing index fragmentation

Indexes are designed to speed up searches on particular columns, but they can become fragmented if many DELETEs are performed on the indexed table. This may result in reduced performance if the index is accessed frequently and the cache is not large enough to hold all of the index.

The sa_index_density system procedure provides information about the degree of fragmentation in a database's indexes. You must have DBA authority to run this procedure. The following statement calls the sa_index_density system procedure:

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

If your index is highly fragmented, you can run REORGANIZE TABLE. You can also drop the index and recreate it. However, if the index is a primary key, you also have to drop and recreate the foreign key indexes.

See also

- ♦ “sa_index_density system procedure” [*SQL Anywhere Server - SQL Reference*]
- ♦ “REORGANIZE TABLE statement” [*SQL Anywhere Server - SQL Reference*]
- ♦ “Working with indexes” on page 80

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.

Reduce table widths

Tables with a large number of columns are known as wide tables. When the number of columns in a table causes the size of individual rows to exceed the database page size, each row is split across two or more database pages. The more pages a row takes up, the longer it takes to read each row. If you find performance slowing, and you know you have tables with many columns, consider 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.

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 732.

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 735.

Strategic sorting of query results

Reduce the amount of unecessary 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 309, or "The GROUP BY clause: organizing query results into groups" on page 301.

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 rewrites can be skipped. Also, if a query is updatable, then depending on the execution plan chosen by the optimizer, the execution engine 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.

☞ For more information, 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 a Windows XP/200x system, 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 or string to values that only contain numeric data. And whenever possible, choose economical 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 ON. 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 huge 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 on improving bulk operation performance, see “Performance aspects of bulk operations” on page 634.

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 80.

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 195.

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 550, and "Using clustered indexes" on page 81.

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 1,000,000 rows in the Employees table.

☞ For more information about how primary and foreign keys work, see "Relations between tables" [*SQL Anywhere 10 - Introduction*].

Using primary keys to improve query performance

A primary key improves performance on the following statement:

```
SELECT *
FROM Employees
WHERE EmployeeID = 390;
```

Information on the Plan tab

The Plan tab in the Results pane contains the following information:

```
Employees <Employees>
```

Whenever the name inside the parentheses on the Plan tab PLAN description is the same as the name of the table, it means that the primary key for the table is used to improve performance.

Using foreign keys to improve query performance

The following query lists the orders from the customer with customer ID 113:

```
SELECT *
FROM SalesOrders
WHERE CustomerID = 113;
```

Information on the Plan tab

The Plan tab in the Results pane contains the following information:

```
SalesOrders <FK_CustomerID_ID>
```

Here FK_CustomerID_ID refers to the foreign key that the SalesOrders table has for the Customers table.

Separate primary and foreign key indexes

Separate indexes are created automatically for primary and foreign keys. This arrangement allows SQL Anywhere to perform many operations more efficiently.

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 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 CE** The formula is as follows:

    ```
    max( 600 KB, min( dbsize, physical-memory ) );
    ```

    The *dbsize* is the total size of the database file or files started, and *physical-memory* is 25% of the physical memory on the computer.
  - **Windows XP/200x and NetWare** The formula is as follows:

    ```
    max( 2 MB, min( dbsize, physical-memory ) );
    ```

    The *dbsize* is the total size of the database file or files started, and *physical-memory* is 25% of the physical memory on the computer.

    If an AWE cache is used on Windows XP/200x the formula is as follows:

    ```
    min( 100% of available memory-128MB, dbsize );
    ```

    An AWE cache is not used if this value is smaller than 2 MB.

    For information about AWE caches, see "-cw server option" [*SQL Anywhere Server - Database Administration*].
  - **Unix** At least 8 MB.

    For information about Unix initial cache size, see "Dynamic cache sizing on Unix" on page 255.

- **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 CE, the default maximum cache size is the amount of available program memory minus 4 MB. On other non-Unix computers, this is usually approximately 90% of total physical memory, but not more than 256 MB. On Unix, the default maximum cache size is calculated as follows:
    - On 32-bit Unix platforms, it is the lesser of 90% of total physical memory or 1,834,880 KB.

      On 64-bit Unix platforms, it is the lesser of 90% of total physical memory and 8,589,672,320 KB.
- **Minimum cache size** You can control the minimum cache size by specifying the database server -cl server option. By default, the minimum cache size is the same as the initial cache size, except on Windows CE. On Windows CE, 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.

There is no dynamic cache resizing on Novell NetWare. When an Address Windowing Extensions (AWE) cache is used, dynamic cache sizing is disabled. You cannot use an AWE cache on Windows CE.

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

Dynamic cache sizing on Windows

On Windows XP/200x and Windows CE, 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 XP/200x 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 CE 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 in Windows XP/200x.

☞ For more information on these properties, see "Server-level properties" [*SQL Anywhere Server - Database Administration*].

☞ For information on monitoring performance, see "Monitoring database performance" on page 225.

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 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 particular network and using your particular 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*]
- "The Validation utility" [*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 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.

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 chapter also includes an in-depth look at how to make use of OLAP functionality in SQL Anywhere.

CHAPTER 7

Queries: Selecting Data from a Table

Contents

About this chapter

The SELECT statement retrieves data from the database. You can use it to retrieve a subset of the rows in one or more tables and to retrieve a subset of the columns in one or more tables.

This chapter focuses on the basics of single-table SELECT statements. Advanced uses of SELECT are described later in this manual.

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.

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 in the form of a result set. The client application 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 can contain several parts, or **clauses**. 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. For more information, 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 on GROUP BY, HAVING, and ORDER BY clauses, see “Summarizing, Grouping, and Sorting Query Results” on page 295.

Most of the clauses are optional, but if they are included then they must appear in the correct order.

☞ For more information about the SELECT statement syntax, see "SELECT statement" [*SQL Anywhere Server - SQL Reference*].

SQL queries

In this manual, SELECT statements and other SQL statements appear with each clause on a separate row, and with the SQL keywords in uppercase. This is not a requirement. You can enter SQL keywords in any case, and you can break lines at any point.

Keywords and line breaks

For example, the following SELECT statement finds the first and last names of contacts living in California from the Contacts table.

```
SELECT GivenName, Surname
FROM Contacts
WHERE State = 'CA'
```

It is equally valid, though not as readable, to enter this statement as follows:

```
SELECT GivenName,
Surname from Contacts
WHERE State
 = 'CA'
```

Case sensitivity of strings and identifiers

Identifiers (that is, 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.

☞ For more information on creating databases, see "Creating a database" on page 31, or "The Initialization utility" [*SQL Anywhere Server - Database Administration*].

Qualifying identifiers

You can qualify the names of database identifiers if there is ambiguity about which object is being referred to. For example, the SQL Anywhere sample database contains several tables with a column called City, so you may have to qualify references to City with the name of the table. In a larger database you may also have to use the name of the owner of the table to identify the table.

```
SELECT Contacts.City
FROM Contacts
WHERE State = 'CA'
```

Since the examples in this chapter 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.

The remaining sections in this chapter analyze the syntax of the SELECT statement in more detail.

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 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. 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 295.

☞ For more information about expressions, see "Expressions" [*SQL Anywhere Server - SQL Reference*].

The following sections provide examples of the kinds of expressions you can use in a select list.

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; and so this statement retrieves every row in the table:

```
SELECT *
FROM Departments
```

The results look like this:

DepartmentID	DepartmentName	DepartmentHeadID
100	R & D	501
200	Sales	902
300	Finance	1293
400	Marketing	1576
…	…	…

You get exactly the same results by listing all the column names in the table in order after the SELECT keyword:

```
SELECT DepartmentID, DepartmentName, DepartmentHeadID
FROM Departments
```

Like a column name, "*" can be qualified with a table name, as in the following query:

```
SELECT Departments.*
FROM Departments
```

Selecting specific columns from a table

You can limit the columns that a SELECT statement retrieves by listing the desired columns immediately after the SELECT keyword. This SELECT statement has the following syntax:

SELECT *column_name* [, *column_name*]…
FROM *table-name*

In the syntax, *column-name-1*, *column-name-2*, and *table-name* should be replaced with the names of the desired columns and table you are querying.

The list of result set columns is called the **select list**. It is separated by commas. There is no comma after the last column in the list, or if there is only one column in the list. Limiting the columns in this way is sometimes called a **projection**.

For example:

```
SELECT Surname, GivenName
FROM Employees;
```

Projections and restrictions

A **projection** is a subset of the columns in a table. A **restriction** (also called **selection**) is a subset of the rows in a table, based on some conditions.

For example, the following SELECT statement retrieves the names and prices of all products in the SQL Anywhere sample database that cost more than $15:

```
SELECT Name, UnitPrice
FROM Products
WHERE UnitPrice > 15
```

This query uses both a projection (`SELECT Name, UnitPrice`) and a restriction (`WHERE UnitPrice > 15`).

Rearranging the order of columns

The order in which you list the column names determines the order in which the columns are displayed. The two following examples show how to specify column order in a display. Both of them find and display the department names and identification numbers from all five of the rows in the Departments table, but in a different order.

```
SELECT DepartmentID, DepartmentName
FROM Departments
```

DepartmentID	DepartmentName
100	R & D
200	Sales
300	Finance
400	Marketing
…	…

```
SELECT DepartmentName, DepartmentID
FROM Departments
```

DepartmentName	DepartmentID
R & D	100
Sales	200
Finance	300
Marketing	400
…	…

Joins

A join links the rows in two or more tables by comparing the values in columns of each table. For example, you might want to select the order item identification numbers and product names for all order items that shipped more than a dozen pieces of merchandise:

```
SELECT SalesOrderItems.ID, Products.Name
FROM Products JOIN SalesOrderItems
WHERE SalesOrderItems.Quantity > 12
```

The Products table and the SalesOrderItems table are joined together based on the foreign key relationship between them.

☞ For more information about using joins, see “Joins: Retrieving Data from Several Tables” on page 321.

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:

SELECT *column-name* [**AS**] *alias*

Providing an alias can produce more readable results. For example, you can change DepartmentName to Department in a listing of departments as follows:

```
SELECT DepartmentName AS Department,
   DepartmentID AS "Identifying Number"
FROM Departments
```

Department	Identifying Number
R & D	100
Sales	200
Finance	300
Marketing	400
…	…

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:

```
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

The SELECT statements you have seen so far produce results that consist solely of data from the tables in the FROM clause. Strings of characters can also be displayed in query results by enclosing them in single quotation marks and separating them from other elements in the select list with commas.

To enclose a quotation mark in a string, you precede it with another quotation mark.

For example:

```
SELECT 'The department''s name is' AS "Prefix",
   DepartmentName AS Department
FROM Departments;
```

Prefix	Department
The department's name is	R & D
The department's name is	Sales
The department's name is	Finance
The department's name is	Marketing
The department's name is	Shipping

Computing values in the SELECT list

The expressions in the select list can be more complicated than just column names or strings. For example, you can perform computations with data from numeric columns in a select list.

Arithmetic operations

To illustrate the numeric operations you can perform in the select list, you start with a listing of the names, quantity in stock, and unit price of products in the SQL Anywhere sample database.

```
SELECT Name, Quantity, UnitPrice
FROM Products;
```

Name	Quantity	UnitPrice
Tee Shirt	28	9
Tee Shirt	54	14
Tee Shirt	75	14
Baseball Cap	112	9
…	…	…

Suppose the practice is to replenish the stock of a product when there are ten items left in stock. The following query lists the number of each product that must be sold before re-ordering:

```
SELECT Name, Quantity - 10
   AS "Sell before reorder"
FROM Products;
```

Name	Sell before reorder
Tee Shirt	18

Name	Sell before reorder
Tee Shirt	44
Tee Shirt	65
Baseball Cap	102
…	…

You can also combine the values in columns. The following query lists the total value of each product in stock:

```
SELECT Name,
   Quantity * UnitPrice AS "Inventory value"
FROM Products;
```

Name	Inventory value
Tee Shirt	252.00
Tee Shirt	756.00
Tee Shirt	1050.00
Baseball Cap	1008.00
…	…

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 avoid misunderstandings, it is recommended that you use parentheses. The following query has the same meaning and gives the same results as the previous one, but some may find it easier to understand:

```
SELECT Name, ( Quantity * UnitPrice ) - 5
FROM Products;
```

☞ For more information on operator precedence, see "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.

The following example illustrates the use of the string concatenation operator in the select list:

```
SELECT EmployeeID, GivenName || ' ' || Surname AS Name
FROM Employees;
```

EmployeeID	Name
102	Fran Whitney
105	Matthew Cobb
129	Philip Chin
148	Julie Jordan
…	…

Date and time operations

Although you can use operators on date and time columns, this typically involves the use of functions. For information on SQL functions, see "UltraLite SQL Function Reference 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:

  ```
  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.

  For a complete list of operators, see "Operators" [*SQL Anywhere Server - SQL Reference*].

- **Functions can be used** In addition to combining columns, you can use a wide range of built-in functions to produce the results you want.

  For example, the following query lists the product names in uppercase:

  ```
  SELECT ID, UCASE( Name )
  FROM Products;
  ```

ID	UCASE(Products.name)
300	TEE SHIRT
301	TEE SHIRT
302	TEE SHIRT
400	BASEBALL CAP
…	…

For a complete list of functions, see “Alphabetical list of functions” [*SQL Anywhere Server - SQL Reference*].

Eliminating duplicate query results

The optional DISTINCT keyword eliminates duplicate rows from the results of a SELECT statement.

If you do not specify DISTINCT, you get all rows, including duplicates. Optionally, you can specify ALL before the select list to get all rows. For compatibility with other implementations of SQL, SQL Anywhere syntax allows the use of ALL to explicitly ask for all rows. ALL is the default.

For example, if you search for all the cities in the Contacts table without DISTINCT, you get 60 rows:

```
SELECT City
FROM Contacts;
```

You can eliminate the duplicate entries using DISTINCT. The following query returns only 16 rows.:

```
SELECT DISTINCT City
FROM Contacts;
```

NULL values are not distinct

The DISTINCT keyword treats NULL values as duplicates of each other. In other words, when DISTINCT is included in a SELECT statement, only one NULL is returned in the results, no matter how many NULL values are encountered.

See also

- “Elimination of unnecessary DISTINCT conditions” on page 477

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 on these features, see "Joins: Retrieving Data from Several Tables" on page 321.

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 not the same as the user ID of the current connection, or 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 save improve readability and 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.

☞ For more information about the FROM clause, see "FROM clause" [*SQL Anywhere Server - SQL Reference*].

Querying from objects other than tables

The most common elements in a FROM clause are table names. 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. In addition to querying from tables and views, you can use derived tables (which are SELECT statements) or stored procedures that return result sets.

For example, the following query operates on the result set of a stored procedure.

```
SELECT *
FROM ShowCustomerProducts( 149 );
```

☞ For more information, see "FROM clause" [*SQL Anywhere Server - SQL Reference*].

The WHERE clause: specifying rows

The WHERE clause in a SELECT statement specifies the search conditions for exactly which rows are retrieved. The general format is:

SELECT *select_list*
FROM *table_list*
WHERE *search-condition*

Search conditions, also called qualifications or predicates, in the WHERE clause include the following:

- **Comparison operators** (=, <, >, and so on) For example, you can list all employees earning more than $50,000:

  ```
  SELECT Surname
  FROM Employees
  WHERE Salary > 50000;
  ```

- **Ranges** (BETWEEN and NOT BETWEEN) For example, you can list all employees earning between $40,000 and $60,000:

  ```
  SELECT Surname
  FROM Employees
  WHERE Salary BETWEEN 40000 AND 60000;
  ```

- **Lists** (IN, NOT IN) For example, you can list all customers in Ontario, Quebec, or Manitoba:

  ```
  SELECT CompanyName , State
  FROM Customers
  WHERE State IN( 'ON', 'PQ', 'MB');
  ```

- **Character matches** (LIKE and NOT LIKE) For example, you can list all customers whose phone numbers start with 415. (The phone number is stored as a string in the database):

  ```
  SELECT CompanyName , Phone
  FROM Customers
  WHERE Phone LIKE '415%';
  ```

- **Unknown values** (IS NULL and IS NOT NULL) For example, you can list all departments with managers:

  ```
  SELECT DepartmentName
  FROM Departments
  WHERE DepartmentHeadID IS NOT NULL;
  ```

- **Combinations** (AND, OR) For example, you can list all employees earning over $50,000 whose first name begins with the letter A.

  ```
  SELECT GivenName, Surname
  FROM Employees
  WHERE Salary > 50000
  AND GivenName like 'A%';
  ```

☞ For more information about 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*

☞ For more information about comparison operators, see “Comparison operators” [*SQL Anywhere Server - SQL Reference*]. For a description of what an expression can consist of, see “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. It is on the Extended Information tab of the database property sheet.

- **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. Trailing blanks are ignored by default in Adaptive Server Enterprise databases.

- **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, which is the default behavior for Adaptive Server Enterprise databases.

Here are some SELECT statements using comparison operators:

```
SELECT *
FROM Products
WHERE Quantity < 20;
SELECT E.Surname, E.GivenName
FROM Employees E
WHERE Surname > 'McBadden';
SELECT ID, Phone
FROM Contacts
WHERE State  != 'CA';
```

The NOT operator

The NOT operator negates an expression. Either of the following two queries will find all Tee shirts and baseball caps that cost $10 or less. However, note the difference in position between the negative logical operator (NOT) and the negative comparison operator (!>).

```
SELECT ID, Name, Quantity
FROM Products
WHERE (name = 'Tee Shirt' OR name = 'BaseBall Cap')
AND NOT UnitPrice > 10;
SELECT ID, Name, Quantity
FROM Products
WHERE (name = 'Tee Shirt' OR name = 'BaseBall Cap')
AND UnitPrice !> 10;
```

Using ranges in the WHERE clause

The BETWEEN keyword specifies an inclusive range, in which the lower value and the upper value 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;
```

Name	UnitPrice
Tee Shirt	14
Tee Shirt	14
Baseball Cap	10
Shorts	15

You can use NOT BETWEEN to find all the rows that are not inside the range.

♦ To list all the products cheaper than $10 or more expensive than $15

• Execute the following query:

```
SELECT Name, UnitPrice
FROM Products
WHERE UnitPrice NOT BETWEEN 10 AND 15;
```

Name	UnitPrice
Tee Shirt	9
Baseball Cap	9
Visor	7
Visor	7

Name	UnitPrice
…	…

Using lists in the WHERE clause

The IN keyword allows you to select values that match any one of a list of values. The expression can be a constant or a column name, and the list can be a set of constants or, more commonly, a subquery.

For example, without IN, if you want a list of the names and states of all the customers who live in Ontario, Manitoba, or Quebec, you can enter this query:

```
SELECT CompanyName, State
FROM Customers
WHERE State = 'ON' OR State = 'MB' OR State = 'PQ';
```

However, you get the same results if you use IN. The items following the IN keyword must be separated by commas and enclosed in parentheses. Put single quotes around character, date, or time values. For example:

```
SELECT CompanyName, State
FROM Customers
WHERE State IN( 'ON', 'MB', 'PQ');
```

Perhaps the most important use for the IN keyword is in nested queries, also called subqueries.

Matching character strings in the WHERE clause

Pattern matching is a versatile way of identifying character data. In SQL, the LIKE keyword is used to search for patterns. Pattern matching employs wildcard characters to match different combinations of characters.

The LIKE keyword indicates that the following character string is a matching pattern. LIKE is used with character data.

The syntax for LIKE is:

expression [**NOT**] **LIKE** *match-expression*

The expression to be matched is compared to a match-expression that can include these special symbols:

Symbols	Meaning
%	Matches any string of 0 or more characters
_	Matches any one character

Symbols	Meaning
[specifier]	The specifier in the brackets may take the following forms: ♦ **Range** A range is of the form *rangespec1-rangespec2*, where *rangespec1* indicates the start of a range of characters, the hyphen indicates a range, and *rangespec2* indicates the end of a range of characters ♦ **Set** A set can be comprised of any discrete set of values, in any order. For example, [a2bR]. Note that the range [a-f], and the sets [abcdef] and [fcbdae] return the same set of values.
[^specifier]	The caret symbol (^) preceding a specifier indicates non-inclusion. [^a-f] means not in the range a-f; [^a2bR] means not a, 2, b, or R.

You can match the column data to constants, variables, or other columns that contain the wildcard characters displayed in the table. When using constants, you should enclose the match strings and character strings in single quotes.

Examples

All the following examples use LIKE with the Surname column in the Contacts table. Queries are of the form:

```
SELECT Surname
FROM Contacts
WHERE Surname LIKE match-expression;
```

The first example would be entered as

```
SELECT Surname
FROM Contacts
WHERE Surname LIKE 'Mc%';
```

Match expression	Description	Returns
'Mc%'	Search for every name that begins with the letters **Mc**	McEvoy
'%er'	Search for every name that ends with **er**	Brier, Miller, Weaver, Rayner
'%en%'	Search for every name containing the letters **en**.	Pettengill, Lencki, Cohen
'_ish'	Search for every four-letter name ending in **ish**.	Fish
'Br[iy][ae]r'	Search for Brier, Bryer, Briar, or Bryar.	Brier
'[M-Z]owell'	Search for all names ending with **owell** that begin with a single letter in the range M to Z.	Powell

Match expression	Description	Returns
'M[^c]%'	Search for all names beginning with M' that do not have c as the second letter	Moore, Mulley, Miller, Masalsky

Wildcards require LIKE

Wildcard characters used without LIKE are interpreted as **string literals** rather than as a pattern: they represent exactly their own values. The following query attempts to find any phone numbers that consist of the four characters 415% only. It does not find phone numbers that start with 415.

```
SELECT Phone
FROM Contacts
WHERE Phone = '415%';
```

☞ For more information on string literals, see "Binary 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 clause:

```
WHERE arrival_time = '9:20'
```

fails to find the value, because the entry holds the date as well as the time. 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`.

☞ For more information, 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.

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):

```
"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. 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 in a column where NULL values are permitted, as specified in the create table statement, in two ways:

- **Default** If no data is entered, and the column has no other default setting, NULL is entered.
- **Explicit entry** You can explicitly enter the value NULL by entering 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 a null value.

For example, the DepartmentHeadID column of the Departments table allows nulls. 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);
```

When NULLs are retrieved

When NULLS are retrieved, displays of query results in Interactive SQL show (null) in the appropriate position:

```
SELECT *
FROM Departments;
```

DepartmentID	DepartmentName	DepartmentHeadID
100	R & D	501
200	Sales	904
300	Finance	1293
400	Marketing	1576
500	Shipping	703
201	Eastern Sales	(null)
202	Western Sales	(null)

Testing a column for NULL

You can use IS NULL in 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 this pattern:

```
WHERE column_name IS [NOT] NULL
```

For example:

```
WHERE advance < $5000
OR advance IS NULL
```

☞ For more information, see “NULL value” [*SQL Anywhere Server - SQL Reference*].

Properties of NULL

The following list expands on the properties of NULL.

- **The difference between FALSE and UNKNOWN** Although neither FALSE nor UNKNOWN returns values, there is an important logical difference between FALSE and UNKNOWN, because the opposite of false ("not false") is true. For example,

  ```
  1 = 2
  ```

  evaluates to false and its opposite,

  ```
  1 != 2
  ```

  evaluates to true. But "not unknown" is still unknown. If null values are included in a comparison, you cannot negate the expression to get the opposite set of rows or the opposite truth value.

- **Substituting a value for NULLs** Use the ISNULL built-in function to substitute a particular value for nulls. The substitution is made only for display purposes; actual column values are not affected. The syntax is:

  ISNULL(*expression*, *value*)

For example, use the following statement to select all the rows from Departments, and display all the null values in column DepartmentHeadID with the value -1.

```
SELECT DepartmentID,
            DepartmentName,
            ISNULL(DepartmentHeadID, -1) as DepartmentHead
FROM Departments
```

- **Expressions that evaluate to NULL** An expression with an arithmetic or bitwise operator evaluates to NULL if any of the operands are null. For example, the following statement evaluates to NULL if column1 is NULL:

  ```
  1 + column1
  ```

- **Concatenating strings and NULL** If you concatenate a string and NULL, the expression evaluates to the string. For example, the following statement returns the string abcdef:

  ```
  SELECT 'abc' || NULL || 'def';
  ```

Connecting conditions with logical operators

The logical operators AND, OR, and NOT are used to connect search conditions in WHERE clauses.

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. It does not find the row for Beth Glassmann.

```
SELECT *
FROM Contacts
WHERE GivenName = 'Beth'
   AND Surname = 'Purcell';
```

Using OR

The OR operator also connects two or more conditions, but it returns results when *any* of the conditions is true. The following query searches for rows containing variants of Elizabeth in the GivenName column.

```
SELECT *
FROM Contacts
WHERE GivenName = 'Beth'
   OR GivenName = 'Liz';
```

Using NOT

The NOT operator negates the expression that follows it. The following query lists all the contacts who do not live in California:

```
SELECT *
FROM Contacts
WHERE NOT State = 'CA';
```

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. For example:

```
SELECT *
FROM Customers
WHERE ( City = 'Newmarket'
        OR City = 'Forest Hill' )
   AND State = 'MN'
;
```

Comparing dates in search conditions

You can use operators other than equals to select a set of rows that satisfy the search condition. The inequality operators (< and >) can be used to compare numbers, dates, and even character strings.

♦ **List all employees born before March 13, 1964**

- In Interactive SQL, execute the following query:

```
SELECT Surname, BirthDate
FROM Employees
WHERE BirthDate < 'March 13, 1964'
ORDER BY BirthDate DESC;
```

Surname	BirthDate
Ahmed	1963-12-12
Dill	1963-07-19
Rebeiro	1963-04-12
Garcia	1963-01-23
Pastor	1962-07-14
…	…

Notes

♦ **Automatic conversion to dates** The SQL Anywhere database server knows that the BirthDate column contains dates, and automatically converts the string `'March 13, 1964'` to a date.

♦ **Ways of specifying dates** There are many ways of specifying dates. The following are all accepted by SQL Anywhere:

```
'March 13, 1964'
'1964/03/13'
'1964-03-13'
```

You can tune the interpretation of dates in queries by setting a database option. Dates in the format *yyyy/mm/dd* or *yyyy-mm-dd* are always recognized unambiguously as dates, regardless of the date_order setting.

For more information on controlling allowable date formats in queries, see “date_order option [compatibility]” [*SQL Anywhere Server - Database Administration*], and “Setting options” [*SQL Anywhere Server - Database Administration*].

- ♦ **Other comparison operators** SQL Anywhere supports several comparison operators.

  For a complete list of available 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". Which employees in the company have names that sound like Brown?

♦ List employees with a last name that sound like Brown

- In Interactive SQL, execute the following query:

```
SELECT Surname, GivenName
FROM Employees
WHERE SOUNDEX( Surname ) = SOUNDEX( 'Brown' );
```

Surname	GivenName
Braun	Jane

The algorithm used by SOUNDEX makes it useful mainly for English-language databases.

☞ For more information, 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 desired columns and table you are querying, and *order-by-column-name* with a column in the table. As before, you can use the asterisk as a short form for all the columns in the table.

♦ List the products in alphabetical order

- In Interactive SQL, execute the following query:

```
SELECT ID, Name, Description
FROM Products
ORDER BY Name;
```

ID	Name	Description
400	Baseball Cap	Cotton Cap
401	Baseball Cap	Wool cap
700	Shorts	Cotton Shorts
600	Sweatshirt	Hooded Sweatshirt
…	…	…

Notes

♦ **The order of clauses is important** The ORDER BY clause must follow the FROM clause and the SELECT clause.

♦ **You can specify either ascending or descending order** The default order is ascending. You can specify a descending order by adding the keyword DESC to the end of the clause, as in the following query:

```
SELECT ID, Quantity
FROM Products
ORDER BY Quantity DESC;
```

ID	Quantity
400	112
700	80
302	75
301	54
600	39
…	…

- **You can order by several columns** The following query sorts first by size (alphabetically), and then by name:

  ```
  SELECT ID, Name, Size
  FROM Products
  ORDER BY Size, Name;
  ```

ID	Name	Size
600	Sweatshirt	Large
601	Sweatshirt	Large
700	Shorts	Medium
301	Tee Shirt	Medium
…	…	…

- **The ORDER BY column does not need to be in the select list** The following query sorts products by unit price, even though the price is not included in the result set

  ```
  SELECT ID, Name, Size
  FROM Products
  ORDER BY UnitPrice;
  ```

ID	Name	Size
500	Visor	One size fits all
501	Visor	One size fits all
300	Tee Shirt	Small
400	Baseball Cap	One size fits all
…	…	…

- **If you do not use an ORDER BY clause, and you execute a query more than once, you may appear to get different results** This is because SQL Anywhere may return the same result set in a different order. In the absence of an ORDER BY clause, SQL Anywhere returns rows in whatever order is most efficient. This means the appearance of result sets may vary depending on when you last accessed the row and other factors. The only way to ensure that rows are returned in a particular order is to use ORDER BY.

Using indexes to improve ORDER BY performance

Sometimes there is more than one possible way for the SQL Anywhere database server to execute a query with an ORDER BY clause. You can use indexes to enable the database server to search the tables more efficiently.

Queries with WHERE and ORDER BY clauses

An example of a query that can be executed in more than one possible way is one that has both a WHERE clause and an ORDER BY clause.

```
SELECT *
FROM Customers
WHERE ID > 300
ORDER BY CompanyName;
```

In this example, SQL Anywhere must decide between two strategies:

1. Go through the entire Customers table in order by company name, checking each row to see if the customer ID is greater than 300.

2. Use the key on the ID column to read only the companies with ID greater than 300. The results would then need to be sorted by company name.

If there are very few ID values greater than 300, the second strategy is better because only a few rows are scanned and quickly sorted. If most of the ID values are greater than 300, the first strategy is much better because no sorting is necessary.

Solving the problem

Creating a two-column index on ID and CompanyName could solve the example above. SQL Anywhere can use this index to select rows from the table in the correct order. However, keep in mind that indexes take up space in the database file and involve some overhead to keep up to date. Do not create indexes indiscriminately.

☞ For more information, see “Using indexes” on page 251.

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:

```
SELECT COUNT( * )
FROM Employees;
```

COUNT(*)
75

The result set consists of only one column, with title **COUNT(*)**, and one row, which contains the total number of employees.

♦ List the number of employees in the company and the birth dates of the oldest and youngest employee

- In Interactive SQL, execute the following query:

```
SELECT COUNT(*), MIN(BirthDate), MAX(BirthDate)
FROM Employees;
```

COUNT(*)	MIN(Employees.BirthDate)	MAX(Employees.BirthDate)
75	1936-01-02	1973-01-18

The functions COUNT, MIN, and MAX are called **aggregate functions**. Aggregate functions summarize information. Other aggregate functions include statistical functions such as AVG, STDDEV, and VARIANCE. All but COUNT require a parameter. See “UltraLite 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:

```
SELECT SalesRepresentative, count( * )
FROM SalesOrders
GROUP BY SalesRepresentative
ORDER BY SalesRepresentative;
```

SalesRepresentative	count(*)
129	57
195	50
299	114
467	56
…	…

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.

☞ For more information, see "The GROUP BY clause: organizing query results into groups" on page 301.

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.

```
-- This query is incorrect
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;
  ```

SalesRepresentative	orders
299	114
129	57
1142	57
467	56

☞ For more information about the HAVING clause, see “The HAVING clause: selecting groups of data” on page 307.

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 desired result. In general, that means eliminating undesired rows in earlier clauses.

Example

To list all sales reps with more than 55 orders and an ID of more than 1000, enter the following statement.

```
SELECT SalesRepresentative, count( * )
FROM SalesOrders
WHERE SalesRepresentative > 1000
GROUP BY SalesRepresentative
HAVING count( * ) > 55
ORDER BY SalesRepresentative;
```

The following statement produces the same results.

```
SELECT SalesRepresentative, count( * )
FROM SalesOrders
GROUP BY SalesRepresentative
HAVING count( * ) > 55 AND SalesRepresentative > 1000
ORDER BY SalesRepresentative;
```

SQL Anywhere detects that both statements describe the same result set, and so executes each efficiently.

CHAPTER 8

Summarizing, Grouping, and Sorting Query Results

Contents

About this chapter

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.

This chapter describes how to group and sort query results.

Summarizing query results using aggregate functions

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 among the available 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 "UltraLite 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.

☞ For more information about the HAVING clause, see "The HAVING clause: selecting groups of data" on page 307.

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 now apply to the use of outer reference aggregate functions in subqueries:

- The outer reference aggregate function can only appear in subqueries that are in the SELECT list or HAVING clause, and these clauses must be in the immediate outer block.
- Outer reference aggregate functions can only contain one outer column reference.
- Local column references and outer column references cannot be mixed in the same aggregate function.

Some problems related to the new standards can be circumvented by rewriting the aggregate function so that it only includes local references. For example, the subquery `(SELECT MAX(S.y + R.y) FROM S)` contains both a local column reference (S.y) and an outer column reference (R.y), which is now illegal. It can be rewritten as `(SELECT MAX(S.y) + R.y FROM S)`. In the rewrite, the aggregate function has only a local column reference. The same sort of rewrite can be used when an outer reference aggregate function appears in clauses other than SELECT or HAVING.

Example

The following query produced the following results in Adaptive Server Anywhere version 7.

```
SELECT Name, (SELECT SUM(p.Quantity)
              FROM SalesOrderItems)
FROM Products p
```

name	sum(p.Quantity)
Tee shirt	30,716
Tee shirt	59,238

In later versions, the same query produces the error message `SQL Anywhere Error -149: Function or column reference to 'name' must also appear in a GROUP BY.` The reason that the statement is no longer valid is that the outer reference aggregate function `sum (p.Quantity)` is now computed in the outer block. In later versions, the query is semantically equivalent to the following (except that Z does not appear as part of the result set):

```
SELECT Name,
       SUM(p.Quantity) as Z,
       (SELECT Z FROM SalesOrderItems)
FROM Products p
```

Since the outer block now computes an aggregate function, the outer block is treated as a grouped query and column name must appear in a GROUP BY clause to appear in the SELECT list.

Aggregate functions and data types

Some aggregate functions have meaning only for certain kinds of data. For example, you can use SUM and AVG with numeric columns only.

However, you can use MIN to find the lowest value—the one closest to the beginning of the alphabet—in a character column:

```
SELECT MIN(Surname)
FROM  Contacts
```

Using COUNT(*)

The COUNT(*) function does not require an expression as an argument because, by definition, it does not use information about any particular column. The COUNT(*) function finds the total number of rows in a table. This statement finds the total number of employees:

```
SELECT COUNT(*)
FROM Employees
```

COUNT(*) returns the number of rows in the specified table without eliminating duplicates. It counts each row separately, including rows that contain NULL.

Like other aggregate functions, you can combine count(*) with other aggregates in the select list, with where clauses, and so on:

```
SELECT count(*), AVG(UnitPrice)
FROM Products
WHERE  UnitPrice > 10
```

count(*)	AVG(Products.UnitPrice)
5	18.2

Using aggregate functions with DISTINCT

The DISTINCT keyword is optional with SUM, AVG, and COUNT. When you use DISTINCT, duplicate values are eliminated before calculating the sum, average, or count.

For example, to find the number of different cities in which there are contacts, enter:

```
SELECT count(DISTINCT City)
FROM Contacts
```

count(distinct Contacts.City)
16

You can use more than one aggregate function with DISTINCT in a query. Each DISTINCT is evaluated independently. For example:

```
SELECT count( DISTINCT GivenName ) "first names",
       count( DISTINCT Surname ) "last names"
FROM Contacts
```

first names	last names
48	60

Aggregate functions and NULL

Any NULLS in the column on which the aggregate function is operating are ignored for the 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:

```
SELECT COUNT (DISTINCT Name)
FROM Products
WHERE UnitPrice > 50
```

count(DISTINCT Name)
0

```
SELECT AVG(UnitPrice)
FROM Products
WHERE UnitPrice > 50
```

AVG(Products.UnitPrice)
(NULL)

The GROUP BY clause: organizing query results into groups

The GROUP BY clause divides the output of a table into groups. You can GROUP 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.

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**. (Remember that a scalar aggregate is a single value produced by an aggregate function without a GROUP BY clause.)

Example

The following query lists the average price of each kind of product:

```
SELECT Name, AVG(UnitPrice) AS Price
FROM Products
GROUP BY Name
```

name	Price
Tee Shirt	12.333333333
Baseball Cap	9.5
Visor	7
Sweatshirt	24
…	…

The summary values (vector aggregates) produced by SELECT statements with aggregates and a GROUP BY appear as columns in each row of the results. By contrast, the summary values (scalar aggregates) produced by queries with aggregates and no GROUP BY also appear as columns, but with only one row. For example:

```
SELECT AVG(UnitPrice)
FROM Products
```

AVG(Products.UnitPrice)
13.3

Understanding GROUP BY

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

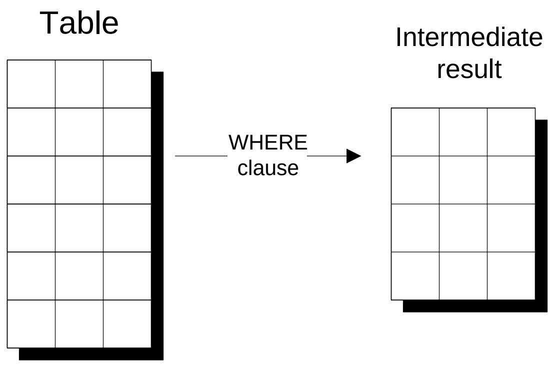
This section uses the ROLLUP sub-clause of the GROUP BY clause in the explanation and example. For more information on the ROLLUP clause, see "Using ROLLUP and CUBE" on page 389.

Consider a single-table query of the following form:

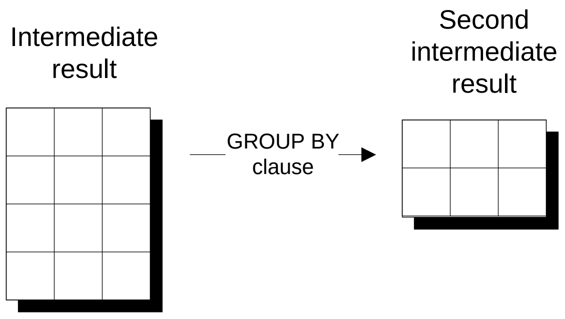
SELECT *select-list*
FROM *table*
WHERE *where-search-condition*
GROUP BY [*group-by-expression* | **ROLLUP** *(group-by-expression)*]
HAVING *having-search-condition*

This query can be thought of as being 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*.

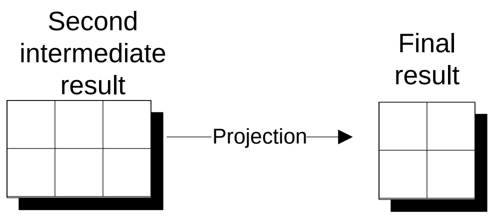


3. **Apply any ROLLUP operation** Subtotal rows computed as part of a ROLLUP operation are added to the result set.

   ☞ For more information, see "Using ROLLUP" on page 390.

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 to nest groups—that is, you can group a table by any combination of expressions.

The following query lists the average price of products, grouped first by name and then by size:

```
SELECT Name, Size, AVG(UnitPrice)
FROM Products
GROUP BY Name, Size
```

name	Size	AVG(Products.UnitPrice)
Tee Shirt	Small	9
Tee Shirt	Medium	14
Tee Shirt	One size fits all	14
Baseball Cap	One size fits all	9.5
…	…	…

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:

```
SELECT  Name, AVG(UnitPrice)
FROM Products
WHERE ID > 400
GROUP BY Name
```

Only the rows with ID values of more than 400 are included in the groups that are used to produce the query results.

Example

The following query illustrates the use of WHERE, GROUP BY, and HAVING clauses in one query:

```
SELECT Name, SUM(Quantity)
FROM Products
WHERE Name LIKE '%shirt%'
GROUP BY Name
HAVING SUM(Quantity) > 100
```

name	SUM(Products.Quantity)
Tee Shirt	157

In this example:

- The WHERE clause includes only rows that have a name including the word *shirt* (Tee Shirt, Sweatshirt).
- The GROUP BY clause collects the rows with a common name.
- The SUM aggregate calculates the total quantity of products available for each group.
- The HAVING clause excludes from the final results the groups whose inventory totals do not exceed 100.

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, while HAVING search conditions often do. The example below is legal:

```
HAVING AVG(UnitPrice) > 20
```

But this example is not legal:

```
WHERE AVG(UnitPrice) > 20
```

Using HAVING with aggregate functions

The following statement is an example of simple use of the HAVING clause with an aggregate function.

To list those products available in more than one size or color, you need a query to group the rows in the Products table by name, but eliminate the groups that include only one distinct product:

```
SELECT Name
FROM Products
GROUP BY Name
HAVING COUNT(*) > 1
```

name
Tee Shirt
Baseball Cap
Visor
Sweatshirt

☞ For information about when you can use aggregate functions in HAVING clauses, see "Where you can use aggregate functions" on page 297.

Using HAVING without aggregate functions

The HAVING clause can also be used without aggregates.

The following query groups the products, and then restricts the result set to only those groups for which the name starts with B.

```
SELECT Name
FROM Products
GROUP BY Name
HAVING Name LIKE 'B%'
```

name
Baseball Cap

More than one condition in HAVING

More than one condition can be included in the HAVING clause. They are combined with the AND, OR, or NOT operators, as in the following example.

To list those products available in more than one size or color, for which one version costs more than $10, you need a query to group the rows in the Products table by name, but eliminate the groups that include only one distinct product, and eliminate those groups for which the maximum unit price is under $10.

```
SELECT Name
FROM Products
GROUP BY Name
HAVING COUNT(*) > 1
AND MAX(UnitPrice) > 10
```

name
Tee Shirt
Sweatshirt

The ORDER BY clause: sorting query results

The ORDER BY clause allows sorting of query results by one or more columns. Each sort can be ascending (ASC) or descending (DESC). If neither is specified, ASC is assumed.

A simple example

The following query returns results ordered by name:

```
SELECT ID, Name
FROM Products
ORDER BY Name
```

ID	name
400	Baseball Cap
401	Baseball Cap
700	Shorts
600	Sweatshirt
…	…

Sorting by more than one column

If you name more than one column in the ORDER BY clause, the sorts are nested.

The following statement sorts the shirts in the Products table first by name in ascending order, then by Quantity (descending) within each name:

```
SELECT ID, Name, Quantity
FROM Products
WHERE Name like '%shirt%'
ORDER BY Name, Quantity DESC
```

ID	name	Quantity
600	Sweatshirt	39
601	Sweatshirt	32
302	Tee Shirt	75
301	Tee Shirt	54
…	…	…

Using the column position

You can use the position number of a column in a select list instead of the column name. Column names and select list numbers can be mixed. Both of the following statements produce the same results as the preceding one.

```
SELECT ID, Name, Quantity
FROM Products
WHERE Name like '%shirt%'
ORDER BY 2, 3 DESC;
SELECT ID, Name, Quantity
FROM Products
WHERE Name like '%shirt%'
ORDER BY 2, Quantity DESC
```

Most versions of SQL require that ORDER BY items appear in the select list, but SQL Anywhere has no such restriction. The following query orders the results by Quantity, although that column does not appear in the select list:

```
SELECT ID, Name
FROM Products
WHERE Name like '%shirt%'
ORDER BY 2, Quantity DESC
```

ORDER BY and NULL

With ORDER BY, NULL sorts before all other values in ascending sort order.

ORDER BY and case sensitivity

The effects of an ORDER BY clause on mixed-case data depend on the database collation and case sensitivity specified when the database is created.

Explicitly limiting the number of rows returned by a query

You can use the FIRST or TOP keywords to limit the number of rows included in the result set of a query. These keywords are for use with queries that include an ORDER BY clause.

Examples

The following query returns information about the employee that appears first when employees are sorted by last name:

```
SELECT FIRST *
FROM Employees
ORDER BY Surname
```

The following query returns the first five employees as sorted by last name:

```
SELECT TOP 5 *
FROM Employees
ORDER BY Surname
```

When you use TOP, you can also use START AT to provide an offset. The following statement lists the fifth and sixth employees sorted in descending order by last name:

```
SELECT TOP 2 START AT 5 *
FROM Employees
ORDER BY Surname DESC
```

FIRST and TOP should be used only in conjunction with an ORDER BY clause to ensure consistent results. Use of FIRST or TOP without an ORDER BY triggers a syntax warning, and will likely yield unpredictable results.

Note
The 'start at' value must be greater than 0. The 'top' value must be greater than 0 when a constant and greater or equal to 0 when a variable.

ORDER BY and GROUP BY

You can use an ORDER BY clause to order the results of a GROUP BY in a particular way.

Example

The following query finds the average price of each product and orders the results by average price:

```
SELECT Name, AVG(UnitPrice)
FROM Products
GROUP BY Name
ORDER BY AVG(UnitPrice)
```

name	AVG(Products.UnitPrice)
Visor	7
Baseball Cap	9.5
Tee Shirt	12.333333333
Shorts	15
…	…

Performing set operations on query results with UNION, INTERSECT, and EXCEPT

The operators described in this section perform set operations on the results of two or more queries. While many of the operations can also be performed using operations in the WHERE clause or HAVING clause, there are some operations that are very difficult to perform in any way other than using these set-based operators. For example:

- When data is held in an unnormalized manner, 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.

  For more information on NULL and set operations, see "Set operators and NULL" on page 315.

For more information, see "EXCEPT operation" [*SQL Anywhere Server - SQL Reference*], "INTERSECT operation" [*SQL Anywhere Server - SQL Reference*], and "UNION operation" [*SQL Anywhere Server - SQL Reference*].

Combining sets with the UNION operation

The UNION operator combines the results of two or more queries into a single result set.

By default, the UNION operator removes duplicate rows from the result set. If you use the ALL option, duplicates are not removed. The columns in the final result set have the same names as the columns in the first result set. Any number of union operators may be used.

By default, a statement containing multiple UNION operators is evaluated from left to right. Parentheses may be used to specify the order of evaluation.

For example, the following two expressions are not equivalent, due to the way that duplicate rows are removed from result sets:

```
x UNION ALL (y UNION z)
(x UNION ALL y) UNION z
```

In the first expression, duplicates are eliminated in the UNION between y and z. In the UNION between that set and x, duplicates are not eliminated. In the second expression, duplicates are included in the union between x and y, but are then eliminated in the subsequent union with z.

Using EXCEPT and INTERSECT

The EXCEPT operation 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 operation 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 operation, both EXCEPT and INTERSECT take the ALL modifier, which prevents the elimination of duplicate rows from the result set.

For more information, see "EXCEPT operation" [*SQL Anywhere Server - SQL Reference*], and "INTERSECT operation" [*SQL Anywhere Server - SQL Reference*].

Rules for set operations

The following rules apply to UNION, EXCEPT, and INTERSECT operations:

- **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:

  ```
  -- This is an example of an invalid statement
  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 operations, 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
  FROM Customers
  ORDER BY 1
  ```

Set operators and NULL

NULL is treated differently by the set operators UNION, EXCEPT, and INTERSECT than it is in search conditions. This difference is one of the main reasons to use set operators.

When comparing rows, set operators treat NULL values as equal to each other. In contrast, when NULL is compared to NULL in a search condition the result is unknown (not true).

One 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:

```
col1 INT,
col2 CHAR(1)
```

The tables and data are set up as follows:

```
CREATE TABLE T1 (col1 INT, col2 CHAR(1));
CREATE TABLE T2 (col1 INT, col2 CHAR(1));
INSERT INTO T1 (col1, col2) VALUES(1, 'a');
INSERT INTO T1 (col1, col2) VALUES(2, 'b');
INSERT INTO T1 (col1) VALUES(3);
INSERT INTO T1 (col1) VALUES(3);
INSERT INTO T1 (col1) VALUES(4);
INSERT INTO T1 (col1) VALUES(4);
INSERT INTO T2 (col1, col2) VALUES(1, 'a');
INSERT INTO T2 (col1, col2) VALUES(2, 'x');
INSERT INTO T2 (col1) VALUES(3);
```

The data in the tables is as follows:

♦ Table T1.

col1	col2
1	a
2	b
3	(NULL)
3	(NULL)
4	(NULL)
4	(NULL)

♦ Table T2

col1	col2
1	a

col1	col2
2	x
3	(NULL)

One query that asks for rows in T1 that also appear in T2 is as follows:

```
SELECT T1.col1, T1.col2
FROM T1 JOIN T2
ON T1.col1 = T2.col1
AND T1.col2 = T2.col2
```

T1.col1	T1.col2
1	a

The row (3, NULL) does not appear in the result set, as the comparison between NULL and NULL is not true. In contrast, approaching the problem using the INTERSECT operator includes a row with NULL:

```
SELECT col1, col2
FROM T1
INTERSECT
SELECT col1, col2
FROM T2
```

col1	col2
1	a
3	(NULL)

The following query uses search conditions to list rows in T1 that do not appear in T2:

```
SELECT col1, col2
FROM T1
WHERE col1 NOT IN (
    SELECT col1
    FROM T2
    WHERE T1.col2 = T2.col2 )
OR col2 NOT IN (
    SELECT col2
    FROM T2
    WHERE T1.col1 = T2.col1 )
```

col1	col2
2	b
3	(NULL)
4	(NULL)
3	(NULL)

col1	col2
4	(NULL)

The NULL-containing rows from T1 are not excluded by the comparison. In contrast, approaching the problem using EXCEPT ALL excludes NULL-containing rows that appear in both tables. In this case, the (3, NULL) row in T2 is identified as the same as the (3, NULL) row in T1.

```
SELECT col1, col2
FROM T1
EXCEPT ALL
SELECT col1, col2
FROM T2
```

col1	col2
2	b
3	(NULL)
4	(NULL)
4	(NULL)

The EXCEPT operator is more restrictive still. It eliminates both (3, NULL) rows from T1 and excludes one of the (4, NULL) rows as a duplicate.

```
SELECT col1, col2
FROM T1
EXCEPT
SELECT col1, col2
FROM T2
```

col1	col2
2	b
4	(NULL)

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.

The results of a standard GROUP BY with vector aggregate functions produce one row with one value per group.

SQL Anywhere and Adaptive Server Enterprise support extensions to HAVING that allow aggregate functions not in the select list and not in the GROUP BY clause.

Compatibility with Adaptive Server Enterprise

Adaptive Server Enterprise supports several extensions to the GROUP BY clause that are not supported in SQL Anywhere. These include the following:

- **Non-grouped columns in the select list** Adaptive Server Enterprise permits column names in the select list that do not appear in the group by clause. For example, the following is valid in Adaptive Server Enterprise:

  ```
  SELECT Name, UnitPrice
  FROM Products
  GROUP BY Name
  ```

  This syntax is not supported in SQL Anywhere.

- **Nested aggregate functions** The following query, which nests a vector aggregate inside a scalar aggregate, is valid in Adaptive Server Enterprise but not in SQL Anywhere:

  ```
  SELECT MAX(AVG(UnitPrice))
  FROM Products
  GROUP BY Name
  ```

- **GROUP BY and ALL** SQL Anywhere does not support the use of ALL in the GROUP BY clause.
- **HAVING with no GROUP BY** SQL Anywhere does not support the use of HAVING with no GROUP BY clause unless all the expressions in the select and having clauses are aggregate functions. For example, the following query is valid in Adaptive Server Enterprise, but is not supported in SQL Anywhere:

```
SELECT UnitPrice
   FROM Products
   HAVING COUNT(*) > 8
```

However, the following statement is valid in SQL Anywhere, because the functions MAX and COUNT are aggregate functions:

```
SELECT MAX(UnitPrice)
FROM Products
HAVING COUNT(*) > 8
```

- **HAVING conditions** Adaptive Server Enterprise supports extensions to HAVING that allow non-aggregate functions not in the select list and not in the GROUP BY clause. Only aggregate functions of this type are allowed in SQL Anywhere.
- **DISTINCT with ORDER BY or GROUP BY** Adaptive Server Enterprise permits the use of columns in the ORDER BY or GROUP BY clause that do not appear in the select list, even in SELECT DISTINCT queries. This can lead to repeated values in the SELECT DISTINCT result set. SQL Anywhere does not support this behavior.
- **Column names in UNIONS** Adaptive Server Enterprise permits the use of columns in the ORDER BY clause in unions of queries. In SQL Anywhere, the ORDER BY clause must use an integer to mark the column by which the results are being ordered.

CHAPTER 9

Joins: Retrieving Data from Several Tables

Contents

About this chapter

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.

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.

Before you start

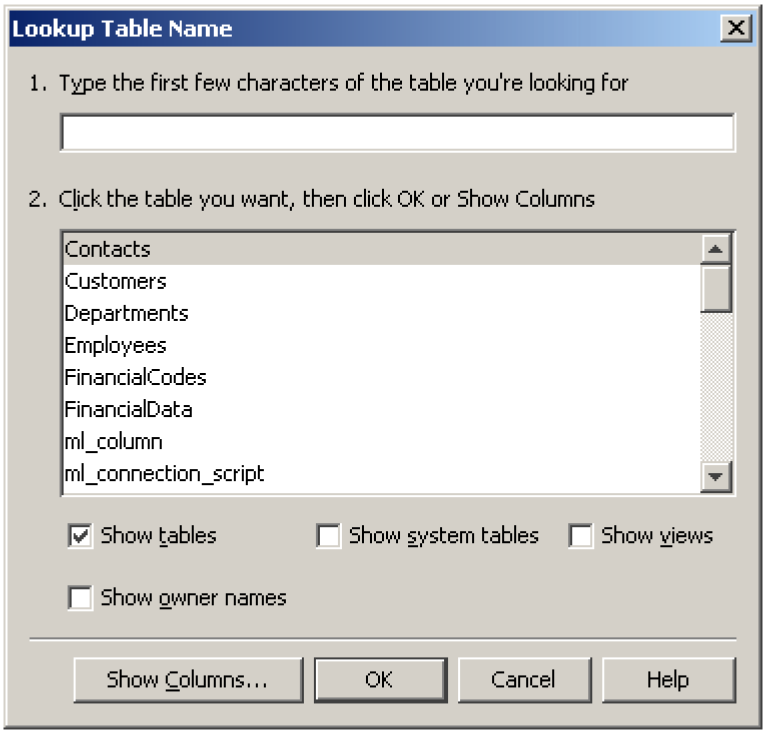
This chapter assumes some knowledge of queries and the syntax of the select statement. Information about queries appears in "Queries: Selecting Data from a Table" on page 263.

Introduction

This chapter describes database queries that look at information in more than one table. To do this, SQL provides the **JOIN** operator. There are several different ways to join tables together in queries, and this chapter describes some of the more important ones.

Displaying a list of tables

In Interactive SQL, you can display a list of tables in the database you are connected to by pressing the F7 key.



Select a table and click Show Columns to see the columns for that table. The Escape key takes you back to the table list, and pressing it again will take you back to the SQL Statements pane. Pressing Enter instead of Escape copies the selected table or column name into the SQL Statements pane at the current cursor position.

Press Escape to leave the list.

☞ For more information on the tables in the SQL Anywhere sample database, see "Tutorial: Using the Sample Database" [*SQL Anywhere Server - Database Administration*].

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.

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.

Joins overview

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.

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 | *view* | *derived table* | *joined table* | **(** *table_expression*, … **)**

table or view:
 [*userid***.**] *table-or-view-name* [[**AS**] *correlation-name*]

derived table:
 (*select-statement* **)** [**AS**] *correlation-name* [**(** *column-name*, … **)**]

joined table:
 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**]

Notes

You cannot use an ON clause with CROSS JOIN.

☞ For the complete syntax, see “FROM clause” [*SQL Anywhere Server - SQL Reference*].

☞ For the syntax of the ON clause, see “Search conditions” [*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** A cross 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 (default)** A join condition is automatically generated based on the foreign key relationships that have been built into the database. Key join is the default when the JOIN keyword is used without specifying a join type and there is no ON clause.
- **NATURAL JOIN** A join condition is automatically generated based on columns having the same name.
- **Join using an ON clause** You specify an explicit join condition. 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. You cannot use an ON clause with a cross join.

Inner and outer joins

Key joins, natural joins and joins with an ON clause may be qualified by specifying INNER, LEFT OUTER, RIGHT OUTER, or FULL OUTER. The default is INNER. When using the keywords LEFT, RIGHT or FULL, the keyword OUTER is optional.

In an inner join, each row in the result satisfies the join condition.

In a left or right outer join, all rows are preserved for one of the tables, and for the other table nulls are returned for rows that do not satisfy the join condition. For example, in a right outer join the right side is preserved and the left side is null-supplying.

In a full outer join, all rows are preserved for both of the tables, and nulls are supplied for rows that do not satisfy the join condition.

Joining two tables

To understand how a simple inner join is computed, consider the following query. It answers the question: which product sizes have been ordered in the same quantity as the quantity in stock?

```
SELECT DISTINCT Name, Size,
    SalesOrderItems.Quantity
FROM Products JOIN SalesOrderItems
ON Products.ID = SalesOrderItems.ProductID
  AND Products.Quantity = SalesOrderItems.Quantity;
```

name	Size	Quantity
Baseball Cap	One size fits all	12
Visor	One size fits all	36

You can interpret the query as follows. Note that this is a conceptual explanation of the processing of this query, used to illustrate the semantics of a query involving a join. It does not represent how SQL Anywhere actually computes the result set.

- Create a cross product of the Products table and SalesOrderItems table. A cross product contains every combination of rows from the two tables.
- Exclude all rows where the product IDs are not identical (because of the join condition `Products.ID = SalesOrderItems.ProductID`).
- Exclude all rows where the quantity is not identical (because of the join condition `Products.Quantity = SalesOrderItems.Quantity`).
- Create a result table with three columns: Products.Name, Products.Size, and SalesOrderItems.Quantity.
- Exclude all duplicate rows (because of the DISTINCT keyword).

☞ For a description of how outer joins are computed, see "Outer joins" on page 334.

Joining more than two tables

With SQL Anywhere, there is no fixed limit on the number of tables you can join.

When joining more than two tables, parentheses are optional. If you do not use parentheses, SQL Anywhere evaluates the statement from left to right. Therefore, `A JOIN B JOIN C` is equivalent to `( A JOIN B ) JOIN C`. Also, the following two SELECT statements are equivalent:

```
SELECT *
FROM A JOIN B JOIN C JOIN D;

SELECT *
FROM ( ( A JOIN B ) JOIN C ) JOIN D;
```

Whenever more than two tables are joined, the join involves table expressions. In the example `A JOIN B JOIN C`, the table expression `A JOIN B` is joined to C. This means, conceptually, that A and B are joined, and then the result is joined to C.

The order of joins is important if the table expression contains outer joins. For example, `A JOIN B LEFT OUTER JOIN C` is interpreted as `(A JOIN B) LEFT OUTER JOIN C`. This means that the table expression `A JOIN B` is joined to C. The table expression `A JOIN B` is preserved and table C is null-supplying.

☞ For more information about outer joins, see "Outer joins" on page 334.

☞ For more information about how SQL Anywhere performs a key join of table expressions, see "Key joins of table expressions" on page 355.

☞ For more information about how SQL Anywhere performs a natural join of table expressions, see "Natural joins of table expressions" on page 350.

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 "Conversion when using comparison operators" [*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 338

♦ "Duplicate correlation names in joins (star joins)" on page 342

♦ "Key joins" on page 352

♦ "Natural joins" on page 348

☞ You can use the REWRITE function to see the ANSI equivalent of a non-ANSI join.

☞ For more information, see "REWRITE function [Miscellaneous]" [*SQL Anywhere Server - SQL Reference*].

Explicit join conditions (the ON clause)

Instead of, or along with, a key or natural join, you can specify a join using an explicit join condition. You specify a join condition by inserting an ON clause immediately after the join. The join condition always refers to the join immediately preceding it. The ON clause applies a restriction to the rows in a join, in much the same way that the WHERE clause applies restrictions to the rows of a query.

The ON clause allows you to construct more useful joins than the CROSS JOIN. For example, you can apply the ON clause to a join of the SalesOrders and Employees table to retrieve only those rows for which the SalesRepresentative in the SalesOrders table is the same as the one in the Employees table in every row of the result. Then each row contains information about an order and the sales representative responsible for it.

For example, in the following query, the first ON clause is used to join SalesOrders to Customers. The second ON clause is used to join the table expression (SalesOrders JOIN Customers) to the base table SalesOrderItems.

```
SELECT *
FROM SalesOrders JOIN Customers
    ON SalesOrders.CustomerID = Customers.ID
  JOIN SalesOrderItems
    ON SalesOrderItems.ID = SalesOrders.ID;
```

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:

```
FROM ( A KEY JOIN B ) JOIN ( C JOIN D ON A.x = C.x )
```

The problem is that the join condition `A.x = C.x` references table A, which is not part of the join it modifies (in this case, `C JOIN D`).

However, as of the ANSI/ISO standard SQL99 and Adaptive Server Anywhere 7.0, there is an exception to this rule: if you use commas between table expressions, an ON condition of a join can reference a table that precedes it syntactically in the FROM clause. Therefore, the following is valid:

```
FROM (A KEY JOIN B) , (C JOIN D ON A.x = C.x)
```

☞ For more information about commas, see “Commas” on page 333.

Example

The following example joins the SalesOrders table with the Employees table. Each row in the result reflects rows in the SalesOrders table where the value of the SalesRepresentative column matched the value of the EmployeeID column of the Employees table.

```
SELECT Employees.Surname,
       SalesOrders.ID,
       SalesOrders.OrderDate
FROM SalesOrders JOIN Employees
   ON SalesOrders.SalesRepresentative = Employees.EmployeeID;
```

Surname	ID	OrderDate
Chin	2008	4/2/2001
Chin	2020	3/4/2001
Chin	2032	7/5/2001
Chin	2044	7/15/2000
Chin	2056	4/15/2001
…	…	…

Following are some notes about this example:

♦ The results of this query contain only 648 rows (one for each row in the SalesOrders table). Of the 48,600 rows in the cross product, only 648 of them have the employee number equal in the two tables.

♦ The ordering of the results has no meaning. You could add an ORDER BY clause to impose a particular order on the query.

♦ The ON clause includes columns that are not included in the final result set.

☞ For additional information, see "Explicit join conditions (the ON clause)" on page 329.

Generated joins and the ON clause

Key joins are the default if the keyword JOIN is used and no join type is specified—unless you use an ON clause. If you use an ON clause with an unspecified JOIN, key join is not the default and no generated join condition is applied.

For example, the following is a key join, because key join is the default when the keyword JOIN is used and there is no ON clause:

```
SELECT *
FROM A JOIN B;
```

The following is a join between table A and table B with the join condition `A.x = B.y`. It is *not* a key join.

```
SELECT *
FROM A JOIN B ON A.x = B.y;
```

If you specify a KEY JOIN or NATURAL JOIN *and* use an ON clause, the final join condition is the conjunction of the generated join condition and the explicit join condition(s). For example, the following statement has two join conditions: one generated because of the key join, and one explicitly stated in the ON clause.

```
SELECT *
FROM A KEY JOIN B ON A.x = B.y;
```

If the join condition generated by the key join is `A.w = B.z`, then the following statement is equivalent:

```
SELECT *
FROM A JOIN B
  ON A.x = B.y
  AND A.w = B.z;
```

☞ For more information about key joins, see “Key joins” on page 352.

Types of explicit join conditions

Most join conditions are based on equality, and so are called **equijoins**. For example,

```
SELECT *
FROM Departments JOIN Employees
   ON Departments.DepartmentID = Employees.DepartmentID;
```

However, you do not have to use equality (=) in a join condition. You can use any search condition, such as conditions containing LIKE, SOUNDEX, BETWEEN, > (greater than), and != (not equal to).

Example

The following example answers the question: For which products has someone ordered more than the quantity in stock?

```
SELECT DISTINCT Products.Name
FROM Products JOIN SalesOrderItems
ON Products.ID = SalesOrderItems.ProductID
   AND SalesOrderItems.Quantity > Products.Quantity;
```

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

Using the WHERE clause for join conditions

Except when using outer joins, you can specify join conditions in the WHERE clause instead of the ON clause. However, you should be aware that there may be semantic differences between the two if the query contains outer joins.

The ON clause is part of the FROM clause, and so is processed before the WHERE clause. This does not make a difference to results except 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 335.

- 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 352.

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.

```
SELECT *
FROM A CROSS JOIN B;
```

The result set from this query includes all columns in A and all columns in B. There is one row in the result set for each combination of a row in A and a row in B. If A has *n* rows and B has *m* rows, the query returns *n* x *m* rows.

Commas

A comma works like a join operator, but is not one. A comma creates a cross product exactly as the keyword CROSS JOIN does. However, join keywords create table expressions, and commas create lists of table expressions.

In the following simple inner join of two tables, a comma and the keywords CROSS JOIN are equivalent:

```
SELECT *
FROM A CROSS JOIN B CROSS JOIN C
WHERE A.x = B.y;
```

and

```
SELECT *
FROM A, B, C
WHERE A.x = B.y;
```

Generally, you can use a comma instead of the keywords CROSS JOIN. The comma syntax is equivalent to cross join syntax, except 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 355.

☞ In the syntax of star joins, commas have a special use. For more information, see “Duplicate correlation names in joins (star joins)” on page 342.

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. The keyword OUTER is optional. These modifiers do not apply to cross joins.

Inner joins

By default, joins are **inner joins**. This means that rows are included in the result set only if they satisfy the join condition.

Example

For example, each row of the result set of the following query contains the information from one Customers row and one SalesOrders row, satisfying the key join condition. If a particular customer has placed no orders, the condition is not satisfied and the result set does not contain the row corresponding to that customer.

```
SELECT GivenName, Surname, OrderDate
FROM Customers KEY INNER JOIN SalesOrders
ORDER BY OrderDate;
```

GivenName	Surname	OrderDate
Hardy	Mums	2000-01-02
Aram	Najarian	2000-01-03
Tommie	Wooten	2000-01-03
Alfredo	Margolis	2000-01-06
…	…	…

Because inner joins and key joins are the defaults, you obtain the same result using the following FROM clause.

```
FROM Customers JOIN SalesOrders
```

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 338.

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.

```
SELECT Surname, OrderDate, City
FROM Customers LEFT OUTER JOIN SalesOrders
   ON Customers.ID = SalesOrders.CustomerID
WHERE Customers.State = 'NY'
ORDER BY OrderDate;
```

Surname	OrderDate	City
Thompson	(NULL)	Bancroft
Reiser	2000-01-22	Rockwood
Clarke	2000-01-27	Rockwood
Mentary	2000-01-30	Rockland
…	…	…

You can interpret the outer join in this statement as follows. Note that this is a conceptual explanation, and does not represent how SQL Anywhere actually computes the result set.

- ♦ Return one row for every sales order placed by a customer. More than one row is returned when the customer placed two or more sales orders, because a row is returned for each sales order. This is the same result as an inner join. The ON condition is used to match customer and sales order rows. The WHERE clause is not used for this step.
- ♦ Include one row for every customer who has not placed any sales orders. This ensures that every row in the Customers table is included. For all 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

```
SELECT *
FROM (A LEFT OUTER JOIN B) RIGHT OUTER JOIN C;
```

Next, you may want to convert the right outer join to a left outer join so that both joins are the same type. To do this, simply reverse the position of the tables in the right outer join, resulting in:

```
SELECT *
FROM C LEFT OUTER JOIN (A LEFT OUTER JOIN B);
```

A is the preserved table and B is the null-supplying table for the nested outer join. C is the preserved table for the first outer join.

You can interpret this join as follows:

- Join A to B, preserving all rows in A.
- Next, join C to the results of the join of A and B, preserving all rows in C.

The join does not have an ON clause, and so is by default a key join. The way SQL Anywhere generates join conditions for this type of join is explained in “Key joins of table expressions that do not contain commas” on page 356.

In addition, the join condition for an outer join must only include tables that have previously been referenced in the FROM clause. This restriction is according to the ANSI/ISO standard, and is enforced to avoid ambiguity. For example, the following two statements are syntactically incorrect, because C is referenced in the join condition before the table itself is referenced.

```
SELECT *
FROM (A LEFT OUTER JOIN B ON B.x = C.x) JOIN C;
```

and

```
SELECT *
FROM A LEFT OUTER JOIN B ON A.x = C.x, C;
```

Outer joins of views and derived tables

Outer joins can also be specified for views and derived tables.

The statement

```
SELECT *
FROM V LEFT OUTER JOIN A ON (V.x = A.x);
```

can be interpreted as follows:

- Compute the view V.
- Join all the rows from the computed view V with A by preserving all the rows from V, using the join condition `V.x = A.x`.

Example

The following example defines a view called V that returns the employee IDs and department names of women who make over $60000.

```
CREATE VIEW V AS
SELECT Employees.EmployeeID, DepartmentName
```

```
    FROM Employees JOIN Departments
      ON Employees.DepartmentID = Departments.DepartmentID
    WHERE Sex = 'F' and Salary > 60000;
```

Next, use this view to add a list of the departments where the women work and the regions where they have sold. The view V is preserved and SalesOrders is null-supplying.

```
SELECT DISTINCT V.EmployeeID, Region, V.DepartmentName
  FROM V LEFT OUTER JOIN SalesOrders
    ON V.EmployeeID = SalesOrders.SalesRepresentative;
```

EmployeeID	Region	DepartmentName
243	(NULL)	R & D
316	(NULL)	R & D
529	(NULL)	R & D
902	Eastern	Sales
…	…	…

Transact-SQL outer joins (*= or =*)

Note
Support for the Transact-SQL outer join operators *= and =* is deprecated and will be removed in a future release.

In accordance with ANSI/ISO SQL standards, SQL Anywhere supports the LEFT OUTER, RIGHT OUTER, and FULL OUTER keywords. For compatibility with Adaptive Server Enterprise 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.ianywhere.com/whitepapers/tsql.html.

In the Transact-SQL dialect, you create outer joins by supplying a comma-separated list of tables in the FROM clause, and using the special operators *= or =* in the WHERE clause. In Adaptive Server Enterprise prior to version 12, the join condition must appear in the WHERE clause (ON was not supported).

Warning: 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 *
               FROM T
               WHERE T.x = S.x )
  ```

Using views with Transact-SQL outer joins

If you define a view with an outer join, and then query the view with a qualification on a column from the null-supplying table of the outer join, the results may not be what you expect. The query returns all rows from the null-supplying table. Rows that do not meet the qualification show a NULL value in the appropriate columns of those rows.

The following rules determine what types of updates you can make to columns through views that contain outer joins:

- INSERT and DELETE statements are not allowed on outer join views.
- UPDATE statements are allowed on outer join views. If the view is defined WITH CHECK option, the update fails if any of the affected columns appears in the WHERE clause in an expression that includes columns from more than one table.

How NULL affects Transact-SQL joins

NULL values in tables or views being joined never match each other in a Transact-SQL outer join. The result of comparing a NULL value with any other NULL value is FALSE.

Specialized joins

This section describes how to create some specialized joins such as self-joins, star joins, and joins using derived tables.

Self-joins

In a **self-join**, a table is joined to itself by referring to the same table using a different correlation name.

Example 1

The following self-join produces a list of pairs of employees. Each employee name appears in combination with every employee name.

```
SELECT a.GivenName, a.Surname,
      b.GivenName, b.Surname
FROM Employees AS a CROSS JOIN Employees AS b;
```

GivenName	Surname	GivenName	Surname
Fran	Whitney	Fran	Whitney
Fran	Whitney	Matthew	Cobb
Fran	Whitney	Philip	Chin
Fran	Whitney	Julie	Jordan
…	…	…	…

Since the Employees table has 75 rows, this join contains 75 x 75 = 5 625 rows. It includes, as well, rows that list each employee with themselves. For example, it contains the row

GivenName	Surname	GivenName	Surname
Fran	Whitney	Fran	Whitney

If you want to exclude rows that contain the same name twice, add the join condition that the employee IDs should not be equal to each other.

```
SELECT a.GivenName, a.Surname,
      b.GivenName, b.Surname
FROM Employees AS a CROSS JOIN Employees AS b
WHERE a.EmployeeID != b.EmployeeID;
```

Without these duplicate rows, the join contains 75 x 74 = 5 550 rows.

This new join contains rows that pair each employee with every other employee, but because each pair of names can appear in two possible orders, each pair appears twice. For example, the result of the above join contains the following two rows.

GivenName	Surname	GivenName	Surname
Matthew	Cobb	Fran	Whitney
Fran	Whitney	Matthew	Cobb

If the order of the names is not important, you can produce a list of the (75 x 74)/2 = 2 775 unique pairs.

```
SELECT a.GivenName, a.Surname,
      b.GivenName, b.Surname
FROM Employees AS a CROSS JOIN Employees AS b
WHERE a.EmployeeID < b.EmployeeID;
```

This statement eliminates duplicate lines by selecting only those rows in which the EmployeeID of employee a is less than that of employee b.

Example 2

The following self-join uses the correlation names report and manager to distinguish two instances of the Employees table, and creates a list of employees and their managers.

```
SELECT report.GivenName, report.Surname,
   manager.GivenName, manager.Surname
FROM Employees AS report JOIN Employees AS manager
   ON (report.ManagerID = manager.EmployeeID)
ORDER BY report.Surname, report.GivenName;
```

This statement produces the result shown partially below. The employee names appear in the two left-hand columns, and the names of their managers are on the right.

GivenName	Surname	GivenName	Surname
Alex	Ahmed	Scott	Evans
Joseph	Barker	Jose	Martinez
Irene	Barletta	Scott	Evans
Jeannette	Bertrand	Jose	Martinez
…	…	…	…

Duplicate correlation names in joins (star joins)

The reason for using duplicate table names is to create a **star join**. In a star join, one table or view is joined to several others.

To create a star join, you use the same table name, view name, or correlation name more than once in the FROM clause. This is an extension to the ANSI/ISO SQL standard. The ability to use duplicate names does not add any additional functionality, but it makes it 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;
```

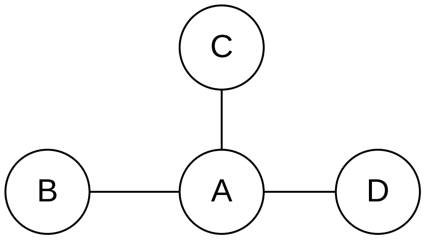
In the next example, table A is joined to three tables: B, C and D.

```
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.



Note You can use duplicate table names only if the extended_join_syntax option is On (the default). For more information, see the "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;
```

GivenName	Surname	OrderDate
Tommie	Wooten	2000-01-03
Michael	Agliori	2000-01-08
Salton	Pepper	2000-01-17
Tommie	Wooten	2000-01-23
…	…	…

Following is the equivalent statement in standard ANSI/ISO syntax:

```
SELECT Customers.GivenName, Customers.Surname,
  SalesOrders.OrderDate
FROM SalesOrders JOIN Customers
  ON SalesOrders.CustomerID =
   Customers.ID
JOIN Employees
  ON SalesOrders.SalesRepresentative =
   Employees.EmployeeID
WHERE Employees.GivenName = 'Rollin'
  AND Employees.Surname = 'Overbey'
ORDER BY SalesOrders.OrderDate;
```

Example 2

This example answers the question: How much of each product has each customer ordered, and who is the manager of the salesperson who took the order?

To answer the question, start by listing the information you need to retrieve. In this case, it is product, quantity, customer name, and manager name. Next, list the tables that hold this information. They are Products, SalesOrderItems, Customers, and Employees. When you look at the structure of the SQL Anywhere sample database (see "Sample database schema" [*SQL Anywhere 10 - Introduction*]), 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.

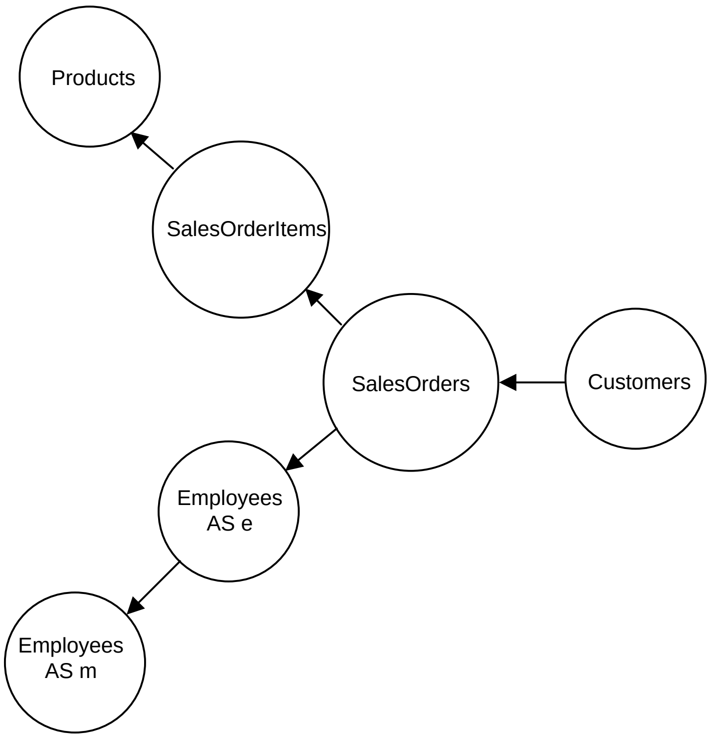
In addition, you need to create a self-join to get the name of the manager, because the Employees table contains ID numbers for managers and the names of all employees, but not a column listing only manager names. For more information, see "Self-joins" on page 341.

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;
```

Given-Name	Name	SUM(SalesOrderItems.Quantity)	GivenName
Sheng	Baseball Cap	240	Moira
Laura	Tee Shirt	192	Moira
Moe	Tee Shirt	192	Moira
Leilani	Sweatshirt	132	Moira
…	…	…	…

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 view.

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 360.

☞ For an explanation of natural joins of derived tables, see "Natural joins of views and derived tables" on page 351.

☞ For an explanation of outer joins of derived tables, see "Outer joins of views and derived tables" on page 337.

Natural joins

When you specify a natural join, SQL Anywhere generates a join condition based on columns with the same name. For this to work in a natural join of base tables, there must be at least one pair of columns with the same name, with one column from each table. If there is no common column name, an error is issued.

If table A and table B have one column name in common, and that column is called x, then

```
SELECT *
FROM A NATURAL JOIN B;
```

is equivalent to the following:

```
SELECT *
FROM A JOIN B
 ON A.x = B.x;
```

If table A and table B have two column names in common, and they are called a and b, then `A NATURAL JOIN B` is equivalent to the following:

```
A JOIN B
 ON A.a = B.a
 AND A.b = B.b;
```

Example 1

For example, you can join the Employees and Departments tables using a natural join because they have a column name in common, the DepartmentID column.

```
SELECT GivenName, Surname, DepartmentName
FROM Employees NATURAL JOIN Departments
ORDER BY DepartmentName, Surname, GivenName;
```

GivenName	Surname	DepartmentName
Janet	Bigelow	Finance
Kristen	Coe	Finance
James	Coleman	Finance
Jo Ann	Davidson	Finance
…	…	…

The following statement is equivalent. It explicitly specifies the join condition that was generated in the previous example.

```
SELECT GivenName, Surname, DepartmentName
FROM Employees JOIN Departments
  ON (Employees.DepartmentID = Departments.DepartmentID)
ORDER BY DepartmentName, Surname, GivenName;
```

Example 2

In Interactive SQL, execute the following query:

```
SELECT Surname, DepartmentName
FROM Employees NATURAL JOIN Departments;
```

Surname	DepartmentName
Whitney	R & D
Cobb	R & D
Breault	R & D
Shishov	R & D
Driscoll	R & D
…	…

SQL Anywhere looks at the two tables and determines that the only column name they have in common is DepartmentID. The following ON CLAUSE is internally generated and used to perform the join:

```
FROM Employees JOIN Departments
   ON Employees.DepartmentID = Departments.DepartmentID
```

NATURAL JOIN is just a shortcut for entering the ON clause; the two queries are identical.

Errors using NATURAL JOIN

The NATURAL JOIN operator can cause problems by equating columns you may not intend to be equated. For example, the following query generates unwanted results:

```
SELECT *
FROM SalesOrders NATURAL JOIN Customers;
```

The result of this query has no rows. SQL Anywhere internally generates the following ON clause:

```
FROM SalesOrders JOIN Customers
   ON SalesOrders.ID = Customers.ID
```

The ID column in the SalesOrders table is an ID number for the *order*. The ID column in the Customers table is an ID number for the *customer*. None of them match. Of course, even if a match were found, it would be a meaningless one.

Natural joins with an ON clause

When you specify a NATURAL JOIN *and* put a join condition in an ON clause, the result is the conjunction of the two join conditions.

For example, the following two queries are equivalent. In the first query, SQL Anywhere generates the join condition `Employees.DepartmentID = Departments.DepartmentID`. The query also contains an explicit join condition.

```
SELECT GivenName, Surname, DepartmentName
FROM Employees NATURAL JOIN Departments
  ON Employees.ManagerID = Departments.DepartmentHeadID;
```

The next query is equivalent. In it, the natural join condition that was generated in the previous query is specified in the ON clause.

```
SELECT GivenName, Surname, DepartmentName
FROM Employees JOIN Departments
  ON Employees.ManagerID = Departments.DepartmentHeadID
   AND Employees.DepartmentID = Departments.DepartmentID;
```

Natural joins of table expressions

When there is a multiple-table expression on at least one side of a natural join, SQL Anywhere generates a join condition by comparing the set of columns for each side of the join operator, and looking for columns that have the same name.

For example, in the statement

```
SELECT *
FROM (A JOIN B) NATURAL JOIN (C JOIN D);
```

there are two table expressions. The column names in the table expression `A JOIN B` are compared to the column names in the table expression `C JOIN D`, and a join condition is generated for each unambiguous pair of matching column names. An *unambiguous pair of matching columns* means that the column name occurs in both table expressions, but does not occur twice in the same table expression.

If there is a pair of ambiguous column names, an error is issued. However, a column name may occur twice in the same table expression, as long as it doesn't also match the name of a column in the other table expression.

Natural joins of lists

When a list of table expressions is on at least one side of a natural join, a separate join condition is generated for each table expression in the list.

Consider the following tables:

♦ table A consists of columns called a, b and c

♦ table B consists of columns called a and d

♦ table C consists of columns called d and c

In this case, the join `(A,B) NATURAL JOIN C` causes SQL Anywhere to generate two join conditions:

```
ON A.c = C.c
 AND B.d = C.d
```

If there is no common column name for A-C or B-C, an error is issued.

If table C consists of columns a, d, and c, then the join `(A,B) NATURAL JOIN C` is invalid. The reason is that column a appears in all three tables, and so the join is ambiguous.

Example

The following example answers the question: for each sale, provide information about what was sold and who sold it.

```
SELECT *
FROM (Employees KEY JOIN SalesOrders)
  NATURAL JOIN (SalesOrderItems KEY JOIN Products);
```

This is equivalent to

```
SELECT *
FROM (Employees KEY JOIN SalesOrders)
  JOIN (SalesOrderItems KEY JOIN Products)
    ON SalesOrders.ID = SalesOrderItems.ID;
```

Natural joins of views and derived tables

An extension to the ANSI/ISO SQL standard is that you can specify views or derived tables on either side of a natural join. In the following statement,

```
SELECT *
FROM View1 NATURAL JOIN View2;
```

the columns in View1 are compared to the columns in View2. If, for example, a column called EmployeeID is found to occur in both views, and there are no other columns that have identical names, then the generated join condition is `(View1.EmployeeID = View2.EmployeeID)`.

Example

The following example illustrates that a view used in a natural join can include expressions, and not just columns, and they are treated the same way in the natural join. First, create the view V with a column called x, as follows:

```
CREATE VIEW V(x) AS
SELECT R.y + 1
FROM R;
```

Next, create a natural join of the view to a derived table. The derived table has a correlation name T with a column called x.

```
SELECT *
FROM V NATURAL JOIN (SELECT P.y FROM P) as T(x);
```

This join is equivalent to the following:

```
SELECT *
FROM V JOIN (SELECT P.y FROM P) as T(x) ON (V.x = T.x);
```

Key joins

Many common joins are between two tables related by a foreign key. The most common join restricts foreign key values to be equal to primary key values. The KEY JOIN operator joins two tables based on a foreign key relationship. In other words, SQL Anywhere generates an ON clause that equates the primary key column from one table with the foreign key column of the other.

When you specify KEY JOIN, SQL Anywhere generates a join condition based on the foreign key relationships in the database. 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 on 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 362.

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” [*SQL Anywhere 10 - Introduction*] for a diagram that includes the role names of all foreign keys in the SQL Anywhere sample database.

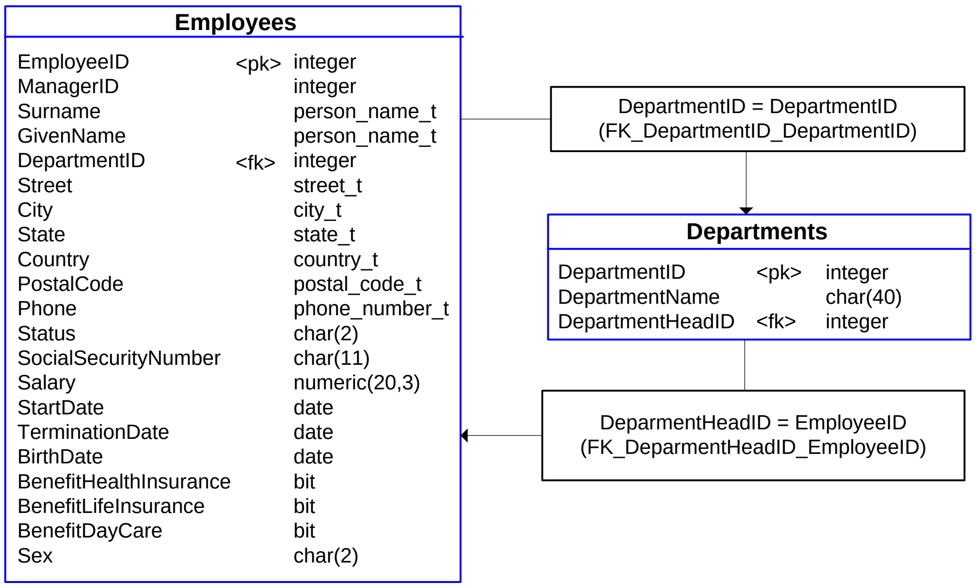
Generating join conditions

SQL Anywhere looks for a foreign key that has the same role name as the correlation name of the primary key table:

- If there is exactly one foreign key with the same name as a table in the join, SQL Anywhere uses it to generate the join condition.
- If there is more than one foreign key with the same name as a table, the join is ambiguous and an error is issued.
- If there is no foreign key with the same name as the table, SQL Anywhere looks for any foreign key relationship, even if the names don't match. If there is more than one foreign key relationship, the join is ambiguous and an error is issued.

Example 1

In the SQL Anywhere sample database, two foreign key relationships are defined between the tables Employees and Departments: the foreign key FK_DepartmentID_DepartmentID in the Employees table references the Departments table; and the foreign key FK_DepartmentHeadID_EmployeeID in the Departments table references the Employees table.



The following query is ambiguous because there are two foreign key relationships and neither has the same role name as the primary key table name. Therefore, attempting this query results in the syntax error `SQLE_AMBIGUOUS_JOIN (-147)`.

```
SELECT Employees.Surname, Departments.DepartmentName
FROM Employees KEY JOIN Departments;
```

Example 2

This query modifies the query in Example 1 by specifying the correlation name FK_DepartmentID_DepartmentID for the Departments table. Now, the foreign key FK_DepartmentID_DepartmentID has the same name as the table it references, and so it is used to define the join condition. The result includes all the employee last names and the departments where they work.

```
SELECT Employees.Surname,
    FK_DepartmentID_DepartmentID.DepartmentName
FROM Employees KEY JOIN Departments
    AS FK_DepartmentID_DepartmentID;
```

The following query is equivalent. It is not necessary to create an alias for the Departments table in this example. The same join condition that was generated above is specified in the ON clause in this query:

```
SELECT Employees.Surname, Departments.DepartmentName
FROM Employees JOIN Departments
   ON Departments.DepartmentID = Employees.DepartmentID;
```

Example 3

If the intent was to list all the employees that are the head of a department, then the foreign key FK_DepartmentHeadID_EmployeeID should be used and Example 1 should be rewritten as follows. This query imposes the use of the foreign key FK_DepartmentHeadID_EmployeeID by specifying the correlation name FK_DepartmentHeadID_EmployeeID for the primary key table Employees.

```
SELECT FK_DepartmentHeadID_EmployeeID.Surname, Departments.DepartmentName
FROM Employees AS FK_DepartmentHeadID_EmployeeID
    KEY JOIN Departments;
```

The following query is equivalent. The join condition that was generated above is specified in the ON clause in this query:

```
SELECT Employees.Surname, Departments.DepartmentName
FROM Employees JOIN Departments
   ON Departments.DepartmentHeadID = Employees.EmployeeID;
```

Example 4

A correlation name is not needed if the foreign key role name is identical to the primary key table name. For example, you can define the foreign key Departments for the Employees table:

```
ALTER TABLE Employees ADD FOREIGN KEY Departments (DepartmentID) REFERENCES
Departments (DepartmentID);
```

Now, this foreign key relationship is the default join condition when a KEY JOIN is specified between the two tables. If the foreign key Departments is defined, then the following query is equivalent to Example 3.

```
SELECT Employees.Surname, Departments.DepartmentName
FROM Employees KEY JOIN Departments;
```

Note

If you try this example in Interactive SQL, you should reverse the change to the SQL Anywhere sample database with the following statement:

```
ALTER TABLE Employees DROP FOREIGN KEY Departments;
```

Key joins of table expressions

SQL Anywhere generates join conditions for the key join of table expressions by examining the foreign key relationship of each pair of tables in the statement.

The following example joins four pairs of tables.

```
SELECT *
FROM (A NATURAL JOIN B) KEY JOIN (C NATURAL JOIN D);
```

The table-pairs are A-C, A-D, B-C and B-D. SQL Anywhere considers the relationship within each pair and then creates a generated join condition for the table expression as a whole. How SQL Anywhere does this depends on whether the table expressions use commas or not. Therefore, the generated join conditions in the following two examples are different. `A JOIN B` is a *table expression that does not contain commas*, and `(A,B)` is a *table expression list*.

```
SELECT *
FROM (A JOIN B) KEY JOIN C;
```

is semantically different from

```
SELECT *
FROM (A,B) KEY JOIN C;
```

The two types of join behavior are explained in the following sections:

♦ "Key joins of table expressions that do not contain commas" on page 356

♦ "Key joins of table expression lists" on page 357

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 353.

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.

☞ For more information, see "Key joins when there are multiple foreign key relationships" on page 353.

Example

The following query returns the names of all salespeople who have sold at least one order to a specific region.

```
SELECT DISTINCT Employees.Surname,
        FK_DepartmentID_DepartmentID.DepartmentName,
        SalesOrders.Region
FROM (SalesOrders, Departments
        AS FK_DepartmentID_DepartmentID)
    KEY JOIN Employees;
```

Surname	DepartmentName	Region
Chin	Sales	Eastern
Chin	Sales	Western
Chin	Sales	Central
…	…	…

This query deals with two pairs of tables: SalesOrders and Employees; and Departments AS FK_DepartmentID_DepartmentID and Employees.

For the pair SalesOrders and Employees, there is no foreign key with the same role name as one of the tables. However, there is a foreign key (FK_SalesRepresentative_EmployeeID) relating the two tables. It is the only foreign key relating the two tables, and so it is used, resulting in the generated join condition `(Employees.EmployeeID = SalesOrders.SalesRepresentative)`.

For the pair Departments AS FK_DepartmentID_DepartmentID and Employees, there is one foreign key that has the same role name as the primary key table. It is FK_DepartmentID_DepartmentID, and it matches the correlation name given to the Departments table in the query. There are no other foreign keys with the same name as the correlation name of the primary key table, so FK_DepartmentID_DepartmentID is used to form the join condition for the table-pair. The join condition that is generated is `(Employees.DepartmentID = FK_DepartmentID_DepartmentID.DepartmentID)`. Note that there is another foreign key relating the two tables, but as it has a different name from either of the tables, it is not a factor.

The final join condition adds together the join condition generated for each table-pair. Therefore, the following query is equivalent:

```
SELECT DISTINCT Employees.Surname,
   Departments.DepartmentName,
   SalesOrders.Region
FROM ( SalesOrders, Departments )
    JOIN Employees
    ON Employees.EmployeeID = SalesOrders.SalesRepresentative
    AND Employees.DepartmentID = Departments.DepartmentID;
```

Key joins of lists and table expressions that do not contain commas

When table expression lists are joined via key join with table expressions that do not contain commas, SQL Anywhere generates a join condition for each table in the table expression list.

For example, the following statement is the key join of a table expression list with a table expression that does not contain commas. This example generates a join condition for table A with table expression `C NATURAL JOIN D`, and for table B with table expression `C NATURAL JOIN D`.

```
SELECT *
FROM (A,B) KEY JOIN (C NATURAL JOIN D);
```

`(A,B)` is a list of table expressions and `C NATURAL JOIN D` is a table expression. SQL Anywhere must therefore generate two join conditions: it generates one join condition for the pairs A-C and A-D, and a second join condition for the pairs B-C and B-D. It does so according to the rules for key joins when there are multiple foreign key relationships:

- For each set of table-pairs, SQL Anywhere looks for a foreign key that has the same role name as the correlation name of one of the primary key tables. If there is exactly one foreign key meeting this criterion, it uses it. If there is more than one, the join is ambiguous and an error is issued.
- For each set of table-pairs, if there is no foreign key with the same name as the correlation name of a table, SQL Anywhere looks for any foreign key relationship between the tables. If there is exactly one relationship, it uses it. If there is more than one, the join is ambiguous and an error is issued.
- For each set of pairs, if there is no foreign key relationship, an error is issued.
- If SQL Anywhere is able to determine exactly one join condition for each set of pairs, it combines the join conditions with the keyword AND.

Example 1

Consider the following join of five tables:

```
((A,B) JOIN (C NATURAL JOIN D) ON A.x = D.y) KEY JOIN E
```

☞ In this case, SQL Anywhere generates a join condition for the key join to E by generating a condition *either* between `(A,B)` and E *or* between `C NATURAL JOIN D` and E. This is as described in "Key joins of table expressions that do not contain commas" on page 356.

☞ 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 357.

☞ 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 356.

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 must be exactly one foreign key relationship between the table-pairs SalesOrders/Employees and SalesOrders/d. There is; it is `SalesOrders.SalesRepresentative = Employees.EmployeeID`.
- There must be exactly one foreign key relationship between the table-pairs FK_DepartmentID_DepartmentID/Employees and FK_DepartmentID_DepartmentID/d. There is; it is `FK_DepartmentID_DepartmentID.DepartmentID = Employees.DepartmentID`.

This example is equivalent to the following. In the following version, it is not necessary to create the correlation name `Departments AS FK_DepartmentID_DepartmentID`, because that was only needed to clarify which of two foreign keys should be used to join Employees and Departments.

```
SELECT DISTINCT Employees.Surname,
   Departments.DepartmentName
FROM (SalesOrders, Departments)
    JOIN (Employees JOIN Departments AS d
       ON Employees.EmployeeID = d.DepartmentHeadID)
    ON SalesOrders.SalesRepresentative = Employees.EmployeeID
       AND Departments.DepartmentID = Employees.DepartmentID;
```

Key joins of views and derived tables

When you include a view or derived table in a key join, SQL Anywhere follows the same basic procedure as with tables, but with these differences:

- For each key join, SQL Anywhere considers the pairs of tables in the FROM clause of the query and the view, and generates *one* join condition for the set of all pairs, regardless of whether the FROM clause in the view contains commas or join keywords.
- SQL Anywhere joins the tables based on the foreign key that has the same role name as the correlation name of the view or derived table.
- When you include a view or derived table in a key join, the view or derived table definition cannot contain UNION, INTERSECT, EXCEPT, ORDER BY, DISTINCT, GROUP BY, aggregate functions, window functions, TOP, FIRST, START AT, or FOR XML. If it contains any of these items, an error is returned. In addition, the derived table cannot be defined as a recursive table expression.

  A derived table works identically to a view. The only difference is that instead of referencing a predefined view, the definition for the table is included in the statement.

  For information about recursive table expressions, see "Recursive common table expressions" on page 372, and "Recursive Table algorithm" on page 522.

Example 1

For example, in the following statement, View1 is a view.

```
SELECT *
FROM View1 KEY JOIN B;
```

The definition of View1 can be any of the following and result in the same join condition to B. (The result set will differ, but the join conditions will be identical.)

```
SELECT *
FROM C CROSS JOIN D;
```

or

```
SELECT *
FROM C,D;
```

or

```
SELECT *
FROM C JOIN D ON (C.x = D.y);
```

In each case, to generate a join condition for the key join of View1 and B, SQL Anywhere considers the table-pairs C-B and D-B, and generates a single join condition. It generates the join condition based on the rules for multiple foreign key relationships described in "Key joins of table expressions" on page 355, 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.

```
SELECT *
FROM View1 KEY JOIN B;
```

is equivalent to

```
SELECT *
FROM View1 JOIN B ON B.y = View1.z;
```

☞ For more information, see "Key joins when there are multiple foreign key relationships" on page 353.

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);
```

This is equivalent to

```
SELECT *
FROM V JOIN (SalesOrders,
   Departments FK_DepartmentID_DepartmentID)
ON (V.EmployeeID = SalesOrders.SalesRepresentative
AND V.DepartmentID =
    FK_DepartmentID_DepartmentID.DepartmentID);
```

Rules describing the operation of key joins

The following rules summarize the information provided above.

Rule 1: key join of two tables

This rule applies to `A KEY JOIN B`, where A and B are base or temporary tables.

1. Find all foreign keys from A referencing B.

   If there exists a foreign key whose role name is the correlation name of table B, then mark it as a preferred foreign key.

2. Find all foreign keys from B referencing A.

   If there exists a foreign key whose role name is the correlation name of table A, then mark it as a preferred foreign key.

3. If there is more than one preferred key, the join is ambiguous. The syntax error `SQLE_AMBIGUOUS_JOIN (-147)` is issued.

4. If there is a single preferred key, then this foreign key is chosen to define the generated join condition for this KEY JOIN expression.

5. If there is no preferred key, then other foreign keys between A and B are used:

   - If there is more than one foreign key between A and B, then the join is ambiguous. The syntax error `SQLE_AMBIGUOUS_JOIN (-147)` is issued.

- ♦ If there is a single foreign key, then this foreign key is chosen to define the generated join condition for this KEY JOIN expression.
- ♦ If there is no foreign key, then the join is invalid and an error is generated.

Rule 2: key join of table expressions that do not contain commas

This rule applies to `A KEY JOIN B`, where A and B are table expressions that do not contain commas.

1. For each pair of tables; one from expression A and one from expression B, list all foreign keys, and mark all preferred foreign keys between the tables. The rule for determining a preferred foreign key is given in Rule 1, above.
2. If there is more than one preferred key, then the join is ambiguous. The syntax error `SQLE_AMBIGUOUS_JOIN (-147)` is issued.
3. If there is a single preferred key, then this foreign key is chosen to define the generated join condition for this KEY JOIN expression.
4. If there is no preferred key, then other foreign keys between pairs of tables are used:
   - ♦ If there is more than one foreign key, then the join is ambiguous. The syntax error `SQLE_AMBIGUOUS_JOIN (-147)` is issued.
   - ♦ If there is a single foreign key, then this foreign key is chosen to define the generated join condition for this KEY JOIN expression.
   - ♦ If there is no foreign key, then the join is invalid and an error is generated.

Rule 3: key join of table expression lists

This rule applies to `(A1, A2, ...) KEY JOIN ( B1, B2, ...)` where A1, B1, and so on are table expressions that do not contain commas.

1. For each pair of table expressions Ai and Bj, find a unique generated join condition for the table expression `(Ai KEY JOIN Bj)` by applying Rule 1 or 2. If any KEY JOIN for a pair of table expressions is ambiguous by Rule 1 or 2, a syntax error is generated.
2. The generated join condition for this KEY JOIN expression is the conjunction of the join conditions found in step 1.

Rule 4: key join of lists and table expressions that do not contain commas

This rule applies to `(A1, A2, ...) KEY JOIN ( B1, B2, ...)` where A1, B1, and so on are table expressions that may contain commas.

1. For each pair of table expressions Ai and Bj, find a unique generated join condition for the table expression `(Ai KEY JOIN Bj)` by applying Rule 1, 2, or 3. If any KEY JOIN for a pair of table expressions is ambiguous by Rule 1, 2, or 3, then a syntax error is generated.
2. The generated join condition for this KEY JOIN expression is the conjunction of the join conditions found in step 1.

CHAPTER 10

Common Table Expressions

Contents

About this chapter

The WITH prefix to the SELECT statement affords you the opportunity to define common table expressions. These can be used like temporary views within your query. This chapter describes how to use them.

About 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.

☞ For information about recursive queries, see "Recursive common table expressions" on page 372.

For example, consider the problem of determining which department has the most employees. The Employees table in the SQL Anywhere sample database lists all the employees in a fictional company and specifies in which department each works. The following query lists the department ID codes and the total number of employees in each department.

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

This query can be used to extract the department with the most employees as follows:

```
SELECT DepartmentID, n
FROM ( SELECT DepartmentID, COUNT(*) AS n
       FROM Employees GROUP BY DepartmentID ) AS a
WHERE a.n =
  ( SELECT MAX( n )
    FROM ( SELECT DepartmentID, COUNT(*) AS n
           FROM Employees GROUP BY DepartmentID ) AS b );
```

While this statement provides the correct result, it has some disadvantages. The first disadvantage is that the repeated subquery makes this statement 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.

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

SELECT DepartmentID, n
FROM CountEmployees
WHERE n = ( SELECT MAX( n )
            FROM CountEmployees );
```

The disadvantage of this approach is that some overhead is required, as the engine 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.

```
WITH CountEmployees( DepartmentID, n ) AS
  ( SELECT DepartmentID, COUNT(*) AS n
    FROM Employees GROUP BY DepartmentID )
SELECT DepartmentID, n
FROM CountEmployees
WHERE n = ( SELECT MAX( n )
            FROM CountEmployees );
```

Changing the query to search for the department with the fewest employees demonstrates that such queries may return multiple rows.

```
WITH CountEmployees( DepartmentID, n ) AS
  ( SELECT DepartmentID, count(*) AS n
    FROM Employees GROUP BY DepartmentID )
SELECT DepartmentID, n
FROM CountEmployees
WHERE n = ( SELECT MIN( n )
            FROM CountEmployees );
```

In the SQL Anywhere sample database, two departments share the minimum number of employees, which is 9.

Multiple correlation names

Just as when using tables, you can give different correlation names to multiple instances of a common table expression. Doing so permits you to join a common table expression to itself. For example, the query below produces pairs of departments that have the same number of employees, although there are only two departments with the same number of employees in the SQL Anywhere sample database.

```
WITH CountEmployees( DepartmentID, n ) AS
    ( SELECT DepartmentID, count(*) AS n
      FROM Employees GROUP BY DepartmentID )
SELECT a.DepartmentID, a.n, b.DepartmentID, b.n
FROM CountEmployees AS a JOIN CountEmployees AS b
ON a.n = b.n AND a.DepartmentID < b.DepartmentID;
```

Multiple table expressions

A single WITH clause may define more than one common table expression. These definitions must be separated by commas. The following example lists the department that has the smallest payroll and the department that has the largest number of employees.

```
WITH
  CountEmployees( DepartmentID, n ) AS
    ( SELECT DepartmentID, count(*) AS n
      FROM Employees GROUP BY DepartmentID ),
  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.

```
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.

```
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.

```
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.

```
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 )
```

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.

```
SELECT GivenName || ' ' || Surname AS sales_rep_name,
       SalesRepresentative AS sales_rep_id,
       SUM( p.UnitPrice * i.Quantity ) AS total_sales
```

```
FROM Employees LEFT OUTER JOIN SalesOrders AS o
              INNER JOIN SalesOrderItems AS i
              INNER JOIN Products AS p
WHERE OrderDate BETWEEN '2000-01-01' AND '2001-12-31'
                  AND OrderDate < '2001-01-01'
GROUP BY SalesRepresentative, GivenName, Surname;
```

The above query is the basis of the common table expression that appears in the following procedure. The ID number of the sales representative and the year in question are incoming parameters. As this 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
                    OrderDate <= 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 technique within procedures. The following example declares a procedure that takes the year in question as an argument.

```
CREATE PROCEDURE sales_by_third ( IN y INTEGER )
BEGIN
  WITH thirds ( q_name, q_start, q_end ) AS
  ( SELECT 'T1', YMD( y, 01, 01), YMD( y, 04, 30 ) UNION
    SELECT 'T2', YMD( y, 05, 01), YMD( y, 08, 31 ) UNION
    SELECT 'T3', YMD( y, 09, 01), YMD( y, 12, 31 ) )
  SELECT q_name,
         SalesRepresentative,
         count(*) AS num_orders,
         SUM( p.UnitPrice * i.Quantity ) AS total_sales
  FROM thirds LEFT OUTER JOIN SalesOrders AS o
    ON OrderDate BETWEEN q_start and q_end
            KEY JOIN SalesOrderItems AS i
            KEY JOIN Products AS p
  GROUP BY q_name, SalesRepresentative
  ORDER BY q_name, SalesRepresentative;
END;

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 can either limit the number of levels of recursion or you can provide no limit. 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 will be excluded, but can introduce infinite recursion should the graph contain 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 “About common table expressions” on page 366.

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,        -- initial subquery
           GivenName, Surname, 0
    FROM Employees AS e
```

```
      WHERE ManagerID = EmployeeID )
    UNION ALL
    ( SELECT e.EmployeeID, e.ManagerID,    -- recursive subquery
             e.GivenName, e.Surname, m.mgmt_level + 1
      FROM Employees AS e JOIN manager AS m
       ON   e.ManagerID =  m.EmployeeID
        AND e.ManagerID <> e.EmployeeID
        AND m.mgmt_level < 20 ) )
  SELECT * FROM manager
  ORDER BY mgmt_level, Surname, GivenName;
```

The condition within the recursive query that restricts the management level to less than 20 is an important precaution. It prevents infinite recursion in the event that the table data contains a cycle.

The max_recursive_iterations option

The option max_recursive_iterations 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 terminate, 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

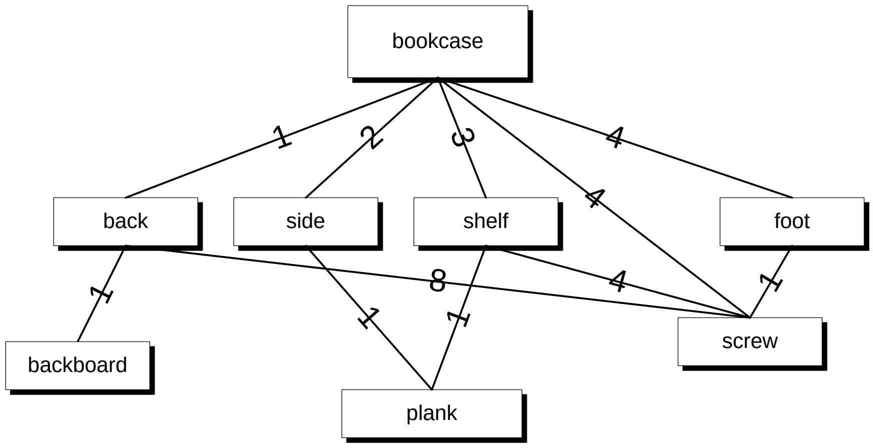
The following restrictions apply to recursive common table expressions.

- 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.

component	**subcomponent**	**quantity**
bookcase	back	1
bookcase	side	2
bookcase	shelf	3
bookcase	foot	4
bookcase	screw	4
back	backboard	1
back	screw	8
side	plank	1

component	subcomponent	quantity
shelf	plank	1
shelf	screw	4

The following statements create the bookcase table and insert the data shown in the above table.

```
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.

```
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.

```
WITH RECURSIVE parts ( component, subcomponent, quantity ) AS
( SELECT component, subcomponent, quantity
  FROM bookcase WHERE component = 'bookcase'
    UNION ALL
  SELECT b.component, b.subcomponent, p.quantity * b.quantity
  FROM parts p JOIN bookcase b ON p.subcomponent = b.component )
SELECT subcomponent, SUM( quantity ) AS quantity
FROM parts
WHERE subcomponent NOT IN ( SELECT component FROM bookcase )
GROUP BY subcomponent
ORDER BY subcomponent;
```

The results of this query are shown below.

subcomponent	quantity
backboard	1
foot	4
plank	5

subcomponent	quantity
screw	24

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:

```
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 results of this query are identical to those of the previous query.

Data type declarations in recursive common table expressions

The data types of the columns in the temporary view are defined by those of the initial subquery. The data types of the columns from the recursive subquery must match. The database server automatically attempts to convert the values returned by the recursive subquery to match those of the initial query. If this is not possible, or if information may be lost in the conversion, an error is generated.

In general, explicit casts are often required when the initial subquery returns a literal value or NULL. Explicit casts may also be required when the initial subquery selects values from different columns than the recursive subquery.

Casts may be required if the columns of the initial subquery do not have the same domains as those of the recursive subquery. Casts must always be applied to NULL values in the initial subquery.

For example, the bookshelf parts explosion sample works correctly because the initial subquery returns rows from the bookcase table, and thus inherits the data types of the selected columns.

☞ For more information, see "Parts explosion problems" on page 374.

If this query is rewritten as follows, explicit casts are required.

```
WITH RECURSIVE parts ( component, subcomponent, quantity ) AS
( SELECT NULL, 'bookcase', 1         -- ERROR! Wrong domains!
    UNION ALL
  SELECT b.component, b.subcomponent,
         p.quantity * b.quantity
  FROM parts p JOIN bookcase b
    ON p.subcomponent = b.component )
SELECT * FROM parts
ORDER BY component, subcomponent;
```

Without casts, 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 short integer, but the data type of the quantity column is INT.

No cast is required for the second column because this column of the initial query is already a string.

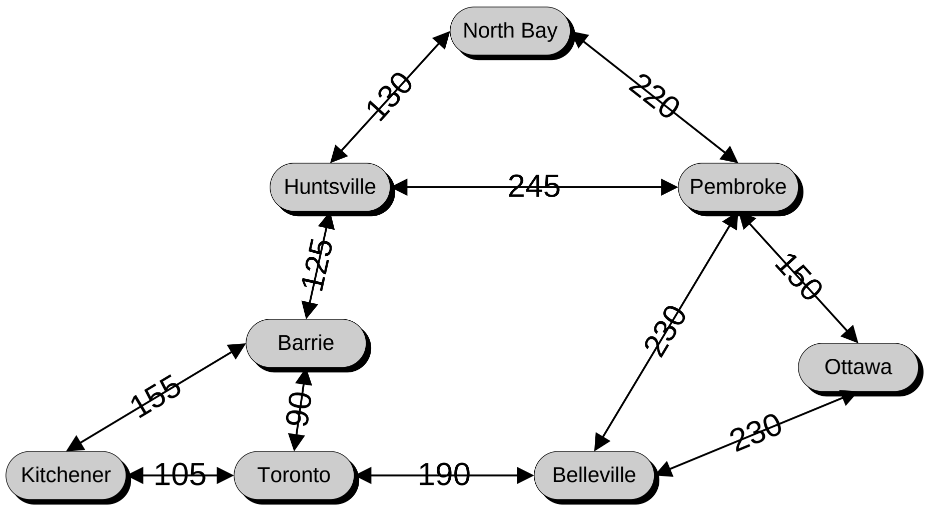
Casting the data types in the initial subquery allows the query to behave as intended:

```
WITH RECURSIVE parts ( component, subcomponent, quantity ) AS
( SELECT CAST( NULL AS VARCHAR ),   -- CASTs must be used
         'bookcase',               -- to declare the
         CAST( 1   AS INT )         -- correct datatypes
      UNION ALL
  SELECT b.component, b.subcomponent,
         p.quantity * b.quantity
  FROM parts p JOIN bookcase b
    ON p.subcomponent = b.component )
SELECT * FROM parts
ORDER BY component, subcomponent;
```

Least distance problem

You can use recursive common table expressions to find desirable paths on a directed graph. Each row in a database table represents a directed edge. Each row specifies an origin, a destination, and a cost of traveling from the origin to the destination. Depending on the problem, the cost may represent distance, travel time, or some other measure. Recursion permits you to explore possible routes through this graph. From the set of possible routes, you can then select the ones that interest you.

For example, consider the problem of finding a desirable way to drive between the cities of Kitchener and Pembroke. There are quite a few possible routes, each of which takes you through a different set of intermediate cities. The goal is to find the shortest routes, and to compare them to reasonable alternatives.



First, define a table to represent the edges of this graph and insert one row for each edge. Since all the edges of this graph 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.

```
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 will start 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 are terminated 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 desired destination.

In the current example, no path should return to Kitchener and all paths should be terminated if they reach Pembroke.

It is particularly important to guarantee that recursive queries will terminate 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 termination by limiting the maximum number of segments in any given route to seven.

Since the point of the example query is to select a practical route, the main query selects only those routes that are less than 50 percent longer than the shortest route.

```
  WITH RECURSIVE
      trip ( route, destination, previous, distance, segments ) AS
  ( SELECT CAST( origin || ', ' || destination AS VARCHAR(256) ),
           destination, origin, distance, 1
    FROM travel
    WHERE origin = 'Kitchener'
      UNION ALL
    SELECT route || ', ' || v.destination,
           v.destination,            -- current endpoint
           v.origin,                 -- previous endpoint
           t.distance + v.distance,  -- total distance
           segments + 1              -- total number of segments
    FROM trip t JOIN travel v ON t.destination = v.origin
    WHERE v.destination <> 'Kitchener'  -- Don't return to start
      AND v.destination <> t.previous   -- Prevent backtracking
      AND v.origin      <> 'Pembroke'   -- Stop at the end
      AND segments                      -- TERMINATE RECURSION!
            < ( SELECT count(*)/2 FROM travel ) )
  SELECT route, distance, segments FROM trip
  WHERE destination = 'Pembroke' AND
        distance < 1.5 * ( SELECT MIN( distance )
                           FROM trip
                           WHERE destination = 'Pembroke' )
  ORDER BY distance, segments, route;
```

When run with against the above data set, this statement yields the following results.

route	distance	segments
Kitchener, Barrie, Huntsville, Pembroke	525	3
Kitchener, Toronto, Belleville, Pembroke	525	3
Kitchener, Toronto, Barrie, Huntsville, Pembroke	565	4
Kitchener, Barrie, Huntsville, North Bay, Pembroke	630	4
Kitchener, Barrie, Toronto, Belleville, Pembroke	665	4
Kitchener, Toronto, Barrie, Huntsville, North Bay, Pembroke	670	5
Kitchener, Toronto, Belleville, Ottawa, Pembroke	675	4

Using multiple recursive common table expressions

A recursive query may include multiple recursive queries, as long as they are disjoint. It may also include a mix of recursive and non-recursive common table expressions. The RECURSIVE keyword must be present if at least one of the common table expressions is recursive.

For example, the following query—which returns the same result as the previous query—uses a second, non-recursive common table expression to select the length of the shortest route. The definition of the second common table expression is separated from the definition of the first by a comma.

```
WITH RECURSIVE
  trip ( route, destination, previous, distance, segments ) AS
    ( SELECT CAST( origin || ', ' || destination AS VARCHAR(256) ),
             destination, origin, distance, 1
      FROM travel
      WHERE origin = 'Kitchener'
      UNION ALL
      SELECT route || ', ' || v.destination,
             v.destination,
             v.origin,
             t.distance + v.distance,
             segments + 1
      FROM trip t JOIN travel v ON t.destination = v.origin
      WHERE v.destination <> 'Kitchener'
        AND v.destination <> t.previous
        AND v.origin      <> 'Pembroke'
        AND segments
              < ( SELECT count(*)/2 FROM travel ) ),
  shortest ( distance ) AS                   -- Additional,
    ( SELECT MIN(distance)                   -- non-recursive
      FROM trip                              -- common table
      WHERE destination = 'Pembroke' )       -- expression
SELECT route, distance, segments FROM trip
WHERE destination = 'Pembroke' AND
      distance < 1.5 * ( SELECT distance FROM shortest )
ORDER BY distance, segments, route;
```

Like non-recursive common table expressions, recursive expressions, when used within stored procedures, may contain references to local variables or procedure parameters. For example, the best_routes procedure, defined below, identifies the shortest routes between the two named cities.

```
CREATE PROCEDURE best_routes (
   IN initial VARCHAR(10),
   IN final   VARCHAR(10)
)
BEGIN
  WITH RECURSIVE
      trip ( route, destination, previous, distance, segments ) AS
  ( SELECT CAST( origin || ', ' || destination AS VARCHAR(256) ),
           destination, origin, distance, 1
    FROM travel
    WHERE origin = initial
      UNION ALL
    SELECT route || ', ' || v.destination,
           v.destination,            -- current endpoint
           v.origin,                 -- previous endpoint
           t.distance + v.distance,  -- total distance
           segments + 1              -- total number of segments
    FROM trip t JOIN travel v ON t.destination = v.origin
    WHERE v.destination <> initial     -- Don't return to start
```

```
         AND v.destination <> t.previous  -- Prevent backtracking
         AND v.origin      <> final       -- Stop at the end
         AND segments                     -- TERMINATE RECURSION!
               < ( SELECT count(*)/2 FROM travel ) )
     SELECT route, distance, segments FROM trip
     WHERE destination = final AND
           distance < 1.4 * ( SELECT MIN( distance )
                              FROM trip
                              WHERE destination = final )
     ORDER BY distance, segments, route;
   END;

   CALL best_routes ( 'Pembroke', 'Kitchener' );
```

CHAPTER 11

OLAP Support

Contents

About this chapter

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. This chapter describes the SQL extensions and functions offered by SQL Anywhere to support OLAP.

Overview of OLAP functionality

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 386.

  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 395.

- **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 401.

- **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 417.

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*]
- ♦ "Clustered Hash Group By algorithm" on page 519
- ♦ "CREATE INDEX statement" [*SQL Anywhere Server - SQL Reference*]
- ♦ "CREATE TABLE statement" [*SQL Anywhere Server - SQL Reference*]
- ♦ "ALTER TABLE statement" [*SQL Anywhere Server - SQL Reference*]

GROUP BY clause extensions

The standard GROUP BY clause of a SELECT statement allows you to group rows in the result set according the grouping expressions you supply. For example, if you specify `GROUP BY columnA, columnB`, the rows are grouped by combinations of unique values from columnA and columnB. In the standard GROUP BY clause, the groups reflect the evaluation of the combination of all specified GROUP BY expressions .

However, you may want to specify different groupings or subgroupings of the result set. For example, you may want to your results to show your data grouped by unique values of columnA and columnB, and then regrouped again by unique values of columnC. You can achieve this result using the GROUPING SETS extension to the GROUP BY clause.

GROUP BY GROUPING SETS

The GROUPING SETS clause is an extension to the GROUP BY clause of a SELECT statement. The GROUPING SETS clause allows you to group your results multiple ways, without having to use multiple SELECT statements to do so. This means you can reduce response time and improve performance.

For example, the following two queries statements are semantically equivalent. However, the second query defines the grouping criteria more efficiently using a GROUP BY GROUPING SETS clause.

Multiple groupings using multiple SELECT statements:

```
SELECT NULL, NULL, NULL, COUNT( * ) AS Cnt
FROM Customers
WHERE State IN ( 'MB' , 'KS' )
   UNION ALL
SELECT City, State, NULL, COUNT( * ) AS Cnt
FROM Customers
WHERE State IN ( 'MB' , 'KS' )
GROUP BY City, State
   UNION ALL
SELECT NULL, NULL, CompanyName, COUNT( * ) AS Cnt
FROM Customers
WHERE State IN ( 'MB' , 'KS' )
GROUP BY CompanyName;
```

Multiple groupings using GROUPING SETS:

```
SELECT City, State, CompanyName, COUNT( * ) AS Cnt
FROM Customers
WHERE State IN ( 'MB' , 'KS' )
GROUP BY GROUPING SETS( ( City, State ), ( CompanyName ) , ( ) );
```

Both methods produce the same results, shown below:

	City	State	CompanyName	Cnt
1	(NULL)	(NULL)	(NULL)	8
2	(NULL)	(NULL)	Cooper Inc.	1
3	(NULL)	(NULL)	Westend Dealers	1

	City	State	CompanyName	Cnt
4	(NULL)	(NULL)	Toto's Active Wear	1
5	(NULL)	(NULL)	North Land Trading	1
6	(NULL)	(NULL)	The Ultimate	1
7	(NULL)	(NULL)	Molly's	1
8	(NULL)	(NULL)	Overland Army Navy	1
9	(NULL)	(NULL)	Out of Town Sports	1
10	'Pembroke'	'MB'	(NULL)	4
11	'Petersburg'	'KS'	(NULL)	1
12	'Drayton'	'KS'	(NULL)	3

Rows 2–9 are the rows generated by grouping over CompanyName, rows 10–12 are rows generated by grouping over the combination of City and State, and row 1 is the grand total represented by the empty grouping set, specified using a pair of matched parentheses (). The empty grouping set represents a single partition of all 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 didn't 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 393.

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:

```
SELECT Year( OrderDate ) AS Year,
        Quarter( OrderDate ) AS Quarter,
        COUNT (*) AS Orders
FROM SalesOrders
GROUP BY GROUPING SETS ( ( Year, Quarter ), ( Year ) )
ORDER BY Year, Quarter;
```

This query returns the following results:

	Year	Quarter	Orders
1	2000	(NULL)	380
2	2000	1	87
3	2000	2	77
4	2000	3	91
5	2000	4	125
6	2001	(NULL)	268
7	2001	1	139
8	2001	2	119
9	2001	3	10

Rows 1 and 6 are subtotals of orders for Year 2000 and Year 2001, respectively. Rows 2–5 and rows 7–9 are the detail rows for the subtotal rows. That is, they show the total orders per quarter, per year.

There is no grand total for all quarters in all years in the result set. To do that, the query must include the empty grouping specification **()** in the GROUPING SETS specification.

Specifying an empty grouping specification

If you use an empty GROUPING SETS specification **()** in the GROUP BY clause, this results in a grand total row for all things that are being totaled in the results. With a grand total row, all values for all grouping expressions contain placeholder NULLs. You can use the GROUPING function to distinguish placeholder NULLs from actual NULLs resulting from the evaluation of values in the underlying data for the row. See "Detecting placeholder NULLs using the GROUPING function" on page 393.

Specifying duplicate grouping sets

You can specify duplicate grouping specifications in a GROUPING SETS clause. In this case, the result of the SELECT statement contains identical rows.

The following query includes duplicate groupings:

```
SELECT City, COUNT( * ) AS Cnt
FROM Customers
WHERE State IN ( 'MB' , 'KS' )
GROUP BY GROUPING SETS( ( City ), ( City ) );
```

This query returns the following results. Note that as a result of the duplicate groupings, rows 1–3 are identical to rows 4–6:

	City	Cnt
1	'Drayton'	3
2	'Petersburg'	1

	City	Cnt
3	'Pembroke'	4
4	'Drayton'	3
5	'Petersburg'	1
6	'Pembroke'	4

Practicing good form

Grouping syntax is interpreted differently for a GROUP BY GROUPING SETS clause than it is for a simple GROUP BY clause. For example, GROUP BY (X, Y) returns results grouped by distinct combinations of X and Y values. However, GROUP BY GROUPING SETS (X, Y) specifies two individual grouping sets, and the result of the two groupings are unioned together. That is, results are grouped by (X), and then unioned to the same results grouped by (Y).

For good form, and to avoid any ambiguity 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:

```
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 393.

Using ROLLUP

A common requirement of many applications is to compute subtotals of the grouping attributes from left-to-right, in sequence. This pattern is referred to as a hierarchy because the introduction of additional subtotal calculations produces additional rows with finer granularity of detail. In SQL Anywhere, you can specify a hierarchy of grouping attributes using the ROLLUP keyword to specify a ROLLUP clause.

A query using a ROLLUP clause produces a hierarchical series of grouping sets, as follows. If the ROLLUP clause contains *n* GROUP BY expressions of the form (X_1,X_2, . . . , X_n) then the ROLLUP clause generates *n* + 1 grouping sets as:

$\{(), (X_1), (X_1,X_2), (X_1,X_2,X_3), . . . , (X_1,X_2,X_3, . . . , X_n)\}.$

Example

The following query summarizes the sales orders by year and quarter, and returns the result set shown in the table below:

```
SELECT QUARTER( OrderDate ) AS Quarter,
       YEAR( OrderDate ) AS Year,
       COUNT( * ) AS Orders,
       GROUPING( Quarter ) AS GQ,
       GROUPING( Year ) AS GY
FROM SalesOrders
GROUP BY ROLLUP( Year, Quarter )
ORDER BY Year, Quarter;
```

This query returns the following results:

	Quarter	Year	Orders	GQ	GY
1	(NULL)	(NULL)	648	1	1
2	(NULL)	2000	380	1	0
3	1	2000	87	0	0
4	2	2000	77	0	0
5	3	2000	91	0	0
6	4	2000	125	0	0
7	(NULL)	2001	268	1	0
8	1	2001	139	0	0
9	2	2001	119	0	0
10	3	2001	10	0	0

The first row in a result set shows the grand total (648) of all orders, for all quarters, for both years.

Row 2 shows total orders (380) for year 2000, while rows 3–6 show the order subtotals, by quarter, for the same year. Likewise, row 7 shows total Orders (268) for year 2001, while rows 8–10 show the subtotals, by quarter, for the same year.

Note how the values returned by GROUPING function can be used to differentiate subtotal rows from the row that contains the grand total. For rows 2 and 7, the presence of NULL in the quarter column, and the value of 1 in the GQ column (Grouping by Quarter), indicate that the row is a totaling of orders in all quarters (per year).

Likewise, in row 1, the presence of NULL in the Quarter and Year columns, plus the presence of a 1 in the GQ and GY columns, indicate that the row is a totaling of orders for all quarters and for all years.

☞ For more information about the syntax for the ROLLUP clause, see "GROUP BY clause" [*SQL Anywhere Server - SQL Reference*].

Support for 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 $(X_1,X_2, \ldots,X_n)$ in a CUBE clause, then CUBE generates 2^n grouping sets as:

$\{(), (X_1), (X_1,X_2), (X_1,X_2,X_3), ... , (X_1,X_2,X_3, . . .,X_n),$
$(X_2), (X_2,X_3), (X_2,X_3,X_4), ... , (X_2,X_3,X_4, ... , X_n), ... , (X_n)\}.$

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:

	Quarter	Year	Orders	GQ	GY
1	(NULL)	(NULL)	648	1	1
2	1	(NULL)	226	0	1
3	2	(NULL)	196	0	1
4	3	(NULL)	101	0	1
5	4	(NULL)	125	0	1
6	(NULL)	2000	380	1	0
7	1	2000	87	0	0
8	2	2000	77	0	0
9	3	2000	91	0	0
10	4	2000	125	0	0
11	(NULL)	2001	268	1	0
12	1	2001	139	0	0
13	2	2001	119	0	0
14	3	2000	10	0	0

The first row in the result set shows the grand total (648) of all orders, for all quarters, for years 2000 and 2001 combined.

Rows 2–5 summarize sales orders by calendar quarter in any year.

Rows 6 and 11 show total Orders for years 2000, and 2001, respectively.

Rows 7–10 and rows 12–14 show the quarterly totals for years 2000, and 2001, respectively.

Note how the values returned by the GROUPING function can be used to differentiate subtotal rows from the row that contains the grand total. For rows 6 and 11, the presence of NULL in the Quarter column, and the value of 1 in the GQ column (Grouping by Quarter), indicate that the row is a totaling of Orders in all quarters for the year.

Note
The result set generated through the use of CUBE can be very large because CUBE generates an exponential number of grouping sets. For this reason, SQL Anywhere does not permit a GROUP BY clause to contain more than 64 GROUP BY expressions. If a statement exceeds this limit, it fails with SQLCODE -944 (SQLSTATE 42WA1).

☞ For more information about the syntax for the CUBE clause, see "GROUP BY clause" [*SQL Anywhere Server - SQL Reference*].

Support for 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 LEFT OUTER JOIN SalesOrders
   ON Employees.EmployeeID = SalesOrders.SalesRepresentative
WHERE Employees.Sex IN ( 'F' )
   AND Employees.State IN ( 'TX' , 'NY' )
GROUP BY GROUPING SETS ( ( Year, Employee ), ( Year ), ( ) )
ORDER BY Year, Employee;
```

This query returns the following results:

	Employees	Year	Orders	GE	GY
1	(NULL)	(NULL)	54	1	1
2	(NULL)	(NULL)	0	1	0

	Employees	Year	Orders	GE	GY
3	102	(NULL)	0	0	0
4	390	(NULL)	0	0	0
5	1062	(NULL)	0	0	0
6	1090	(NULL)	0	0	0
7	1507	(NULL)	0	0	0
8	(NULL)	2000	34	1	0
9	667	2000	34	0	0
10	(NULL)	2001	20	1	0
11	667	2001	20	0	0

In this example, row 1 represents the grand total of orders (54) because the empty grouping set **()** was specified. Notice that GE and GY both contain a 1 to indicate that the NULLs in the Employees and Year columns are placeholder NULLs for Employees and Year columns, respectively.

Row 2 is a subtotal row. The 1 in the GE column indicates that the NULL in the Employees column is a placeholder NULL. The 0 in the GY column indicates that the NULL in the Year column is the result of evaluating the underlying data, and not a placeholder NULL; in this case, this row represents those employees who have no orders.

Rows 3–7 show the total number of orders, per employee, where the Year was NULL. That is, these are the female employees that live in Texas and New York who have no orders. These are the detail rows for row 2. That is, row 2 is a totaling of rows 3–7.

Row 8 is a subtotal row showing the number of orders for all employees combined, in the year 2000. Row 9 is the single detail row for row 8.

Row 10 is a subtotal row showing the number of orders for all employees combined, in the year 2001. Row 11 is the single detail row for row 10.

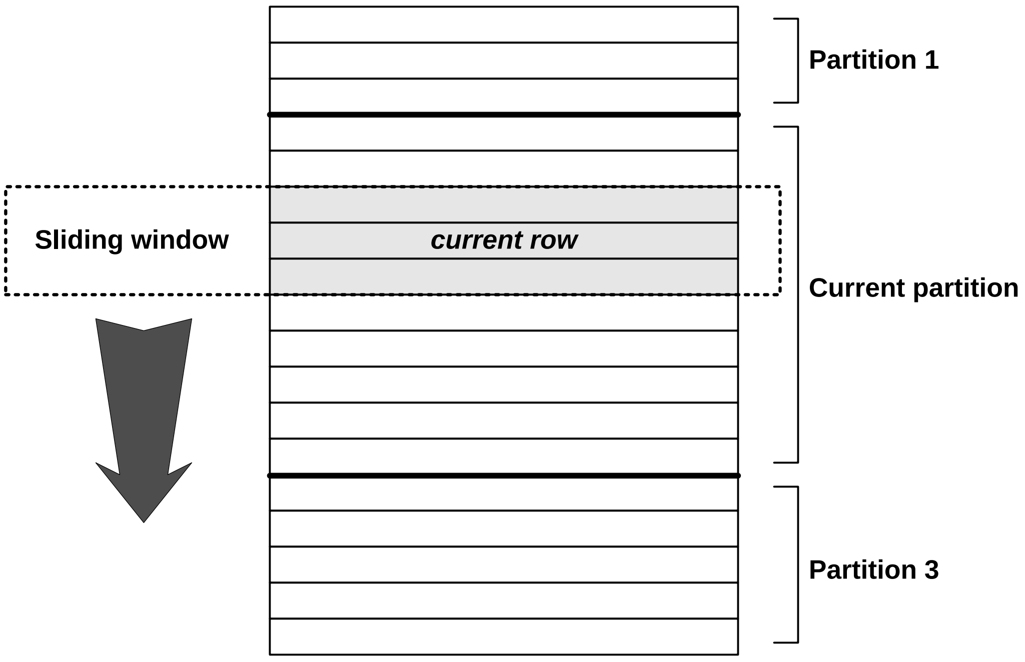
☞ For more information on the syntax of the GROUPING function, see "GROUPING function [Aggregate]" [*SQL Anywhere Server - SQL Reference*].

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 on window aggregate functions, see "Window aggregate functions" on page 401.

♦ **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 on window ranking functions, see "Window ranking functions" on page 417.

♦ **Row numbering functions** Row numbering functions uniquely number the rows in a partition. In SQL Anywhere, the ROW_NUMBER function is ANSI standard-compliant function that permits much of the same functionality as the SQL Anywhere NUMBER(*) function.

☞ For more information on using this function in the window context, see "ROW_NUMBER function" on page 425.

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. 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):

```
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:

	ID	Description	Quantity	ShipDate	Cumulative_qty
1	301	V-neck	24	2001-07-16	24
2	302	Crew Neck	60	2001-07-02	60
3	302	Crew Neck	36	2001-07-13	96
4	400	Cotton Cap	48	2001-07-05	48
5	400	Cotton Cap	24	2001-07-19	72
6	401	Wool Cap	48	2001-07-09	48
7	500	Cloth Visor	12	2001-07-22	12
8	501	Plastic Visor	60	2001-07-07	60
9	501	Plastic Visor	12	2001-07-12	72
10	501	Plastic Visor	12	2001-07-22	84
11	601	Zipped Sweatshirt	60	2001-07-19	60
12	700	Cotton Shorts	24	2001-07-26	24

In this example, the computation of the SUM window function occurs after the join of the two tables and the application of the query's WHERE clause. The query is processed as follows:

1. Partition (group) the input rows based on the value ProductID.
2. Within each partition, sort the rows based on the value of ShipDate.
3. For each row in the partition, evaluate the SUM function over the values in Quantity, using a sliding window consisting of the first (sorted) row of each partition, up to and including the current row.

WINDOW clause 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.

```
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:

    ♦ AVG
    ♦ COUNT
    ♦ MAX
    ♦ MIN
    ♦ SUM

♦ **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
    ♦ REGR_R2
    ♦ REGR_SLOPE
    ♦ REGR_SXX
    ♦ REGR_SXY
    ♦ REGR_SYY

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:

- **AVG function** Returns the average, for a set of rows, of a numeric expression or of a set unique values.
- **COUNT function** Returns the number of rows that qualify for the specified expression.
- **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.
- **SUM function** Returns the total of the specified expression for each group of rows.

☞ To review the mathematical formulas represented by these functions, see "Mathematical formulas for the aggregate functions" on page 425.

The following example shows the SUM function used as a window function. The query returns a result set that partitions the data by DepartmentID, and then provides a cumulative summary (Sum_Salary) of employees' salaries, starting with the employee who has been at the company the longest. The result set includes only those employees who reside in California, Utah, New York, or Arizona. The column Sum_Salary provides the cumulative total of employees' salaries.

```
SELECT DepartmentID, Surname, StartDate, Salary,
SUM( Salary ) OVER ( PARTITION BY DepartmentID
    ORDER BY StartDate
    RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW )
AS "Sum_Salary"
FROM Employees
WHERE State IN ( 'CA', 'UT', 'NY', 'AZ' )
   AND DepartmentID IN ( '100', '200' )
ORDER BY DepartmentID, StartDate;
```

The table that follows represents the result set from the query. The result set is partitioned by DepartmentID.

	DepartmentID	Surname	StartDate	Salary	Sum_Salary
1	100	Whitney	1984-08-28	45700.00	45700.00
2	100	Cobb	1985-01-01	62000.00	107700.00
3	100	Shishov	1986-06-07	72995.00	180695.00
4	100	Driscoll	1986-07-01	48023.69	228718.69
5	100	Guevara	1986-10-14	42998.00	271716.69
6	100	Wang	1988-09-29	68400.00	340116.69
7	100	Soo	1990-07-31	39075.00	379191.69
8	100	Diaz	1990-08-19	54900.00	434091.69
9	200	Overbey	1987-02-19	39300.00	39300.00

	DepartmentID	Surname	StartDate	Salary	Sum_Salary
10	200	Martel	1989-10-16	55700.00	95000.00
11	200	Savarino	1989-11-07	72300.00	167300.00
12	200	Clark	1990-07-21	45000.00	212300.00
13	200	Goggin	1990-08-05	37900.00	250200.00

For DepartmentID 100, the cumulative total of salaries from employees in California, Utah, New York, and Arizona is $434,091.69 and the cumulative total for employees in department 200 is $250,200.00.

☞ For more information on the exact syntax of the SUM function, see "SUM function [Aggregate]" [*SQL Anywhere Server - SQL Reference*].

Computing deltas between adjacent rows

Using two windows—one window over the current row, the other over the previous row—you can compute deltas, or changes, between adjacent rows. For example, the following query computes the delta (Delta) between the salary for one employee and the previous employee in the results:

```
SELECT EmployeeID AS EmployeeNumber,
   Surname AS LastName,
   SUM( Salary ) OVER ( ORDER BY BirthDate
      ROWS BETWEEN CURRENT ROW AND CURRENT ROW )
      AS CurrentRow,
   SUM( Salary ) OVER ( ORDER BY BirthDate
      ROWS BETWEEN 1 PRECEDING AND 1 PRECEDING )
      AS PreviousRow,
   ( CurrentRow - PreviousRow ) AS Delta
FROM Employees
WHERE State IN ( 'NY' );
```

	EmployeeNumber	LastName	CurrentRow	PreviousRow	Delta
1	913	Martel	55700.000	(NULL)	(NULL)
2	1062	Blaikie	54900.000	55700.000	-800.000
3	249	Guevara	42998.000	54900.000	-11902.000
4	390	Davidson	57090.000	42998.000	14092.000
5	102	Whitney	45700.000	57090.000	-11390.000
6	1507	Wetherby	35745.000	45700.000	-9955.000
7	1751	Ahmed	34992.000	35745.000	-753.000
8	1157	Soo	39075.000	34992.000	4083.000

Note that SUM is performed only on the current row for the CurrentRow window because the window size was set to `ROWS BETWEEN CURRENT ROW AND CURRENT ROW`. Likewise, SUM is performed only over the previous row for the PreviousRow window, because the window size was set to `ROWS BETWEEN`

`1 PRECEDING AND 1 PRECEDING`. 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.

Computing a moving average

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.

```
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;
```

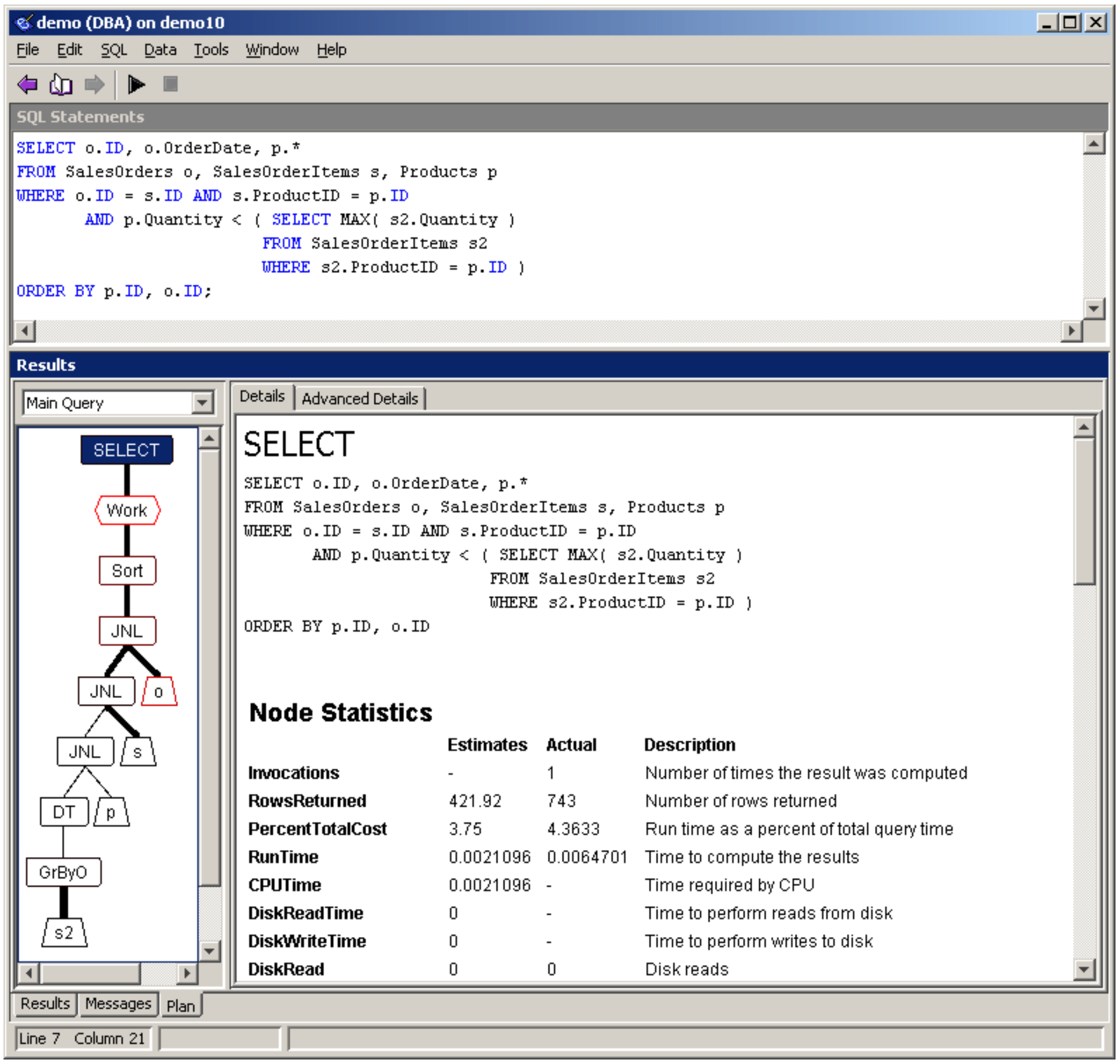
☞ For more information on the syntax for the AVG function, see "AVG function [Aggregate]" [*SQL Anywhere Server - SQL Reference*].

Eliminating correlated subqueries

In some situations, you may need the ability to compare a particular column value with a maximum or minimum value. Often you form these queries as nested queries involving a correlated attribute (also known as an outer reference). As an example, consider the following query, which lists all orders, including product information, where the product quantity-on-hand cannot cover the maximum single order for that product:

```
SELECT o.ID, o.OrderDate, p.*
FROM SalesOrders o, SalesOrderItems s, Products p
WHERE o.ID = s.ID AND s.ProductID = p.ID
       AND p.Quantity < ( SELECT MAX( s2.Quantity )
                          FROM SalesOrderItems s2
                          WHERE s2.ProductID = p.ID )
ORDER BY p.ID, o.ID;
```

The graphical plan for this query is displayed on the Plan tab, in the Results pane of Interactive SQL, 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.



```
SELECT o.ID, o.OrderDate, p.*
FROM SalesOrders o, SalesOrderItems s, Products p
WHERE o.ID = s.ID AND s.ProductID = p.ID
       AND p.Quantity < ( SELECT MAX( s2.Quantity )
                          FROM SalesOrderItems s2
                          WHERE s2.ProductID = p.ID )
ORDER BY p.ID, o.ID;
```

SELECT

```
SELECT o.ID, o.OrderDate, p.*
FROM SalesOrders o, SalesOrderItems s, Products p
WHERE o.ID = s.ID AND s.ProductID = p.ID
       AND p.Quantity < ( SELECT MAX( s2.Quantity )
                          FROM SalesOrderItems s2
                          WHERE s2.ProductID = p.ID )
ORDER BY p.ID, o.ID
```

Node Statistics

	Estimates	Actual	Description
Invocations	-	1	Number of times the result was computed
RowsReturned	421.92	743	Number of rows returned
PercentTotalCost	3.75	4.3633	Run time as a percent of total query time
RunTime	0.0021096	0.0064701	Time to compute the results
CPUTime	0.0021096	-	Time required by CPU
DiskReadTime	0	-	Time to perform reads from disk
DiskWriteTime	0	-	Time to perform writes to disk
DiskRead	0	0	Disk reads

Rather than relying on the optimizer to transform the correlated subquery into a join with a derived table—which can only be done for straightforward cases due to the complexity of the semantic analysis—you can form such queries using a window function:

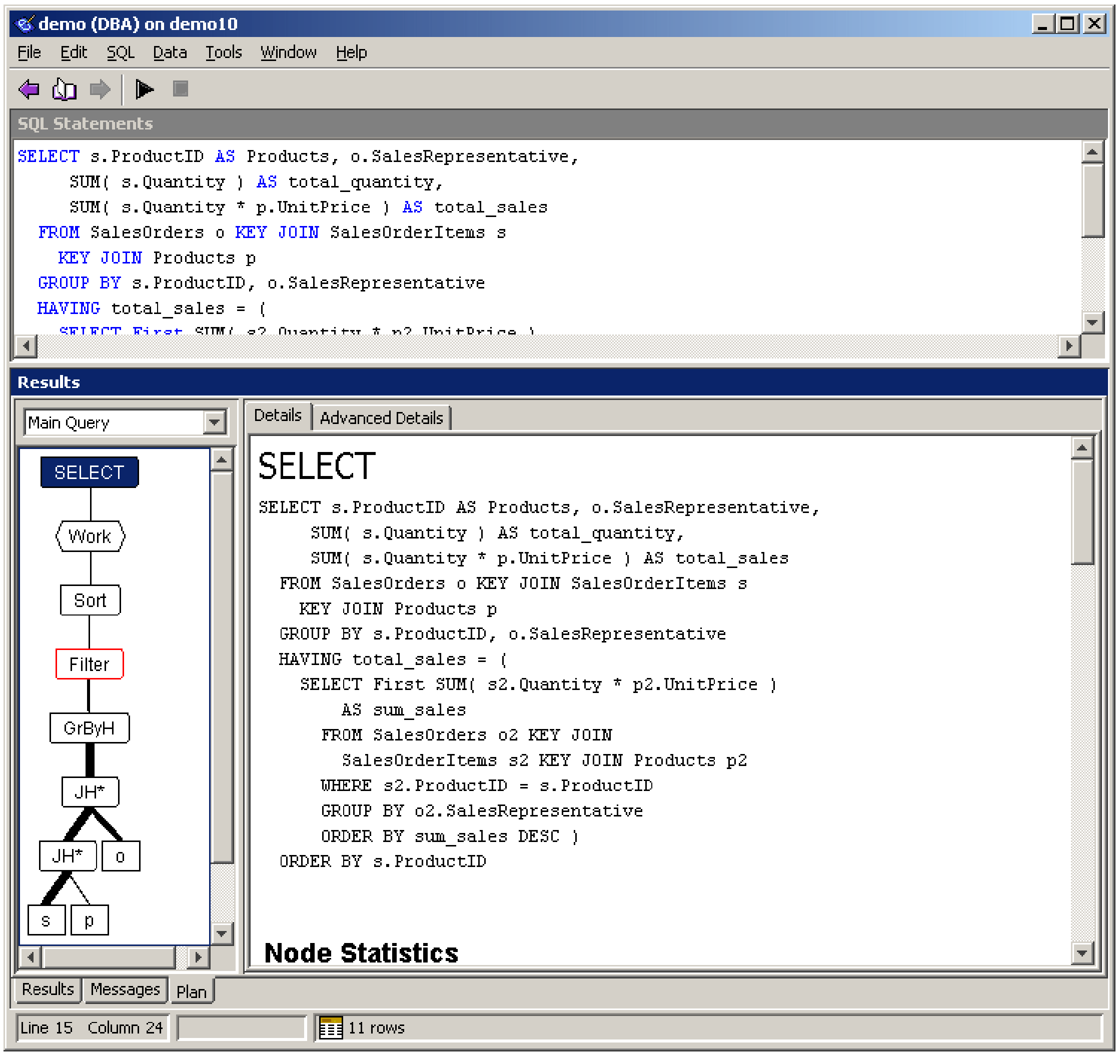
```
SELECT order_qty.ID, o.OrderDate, p.*
  FROM ( SELECT s.ID, s.ProductID,
           MAX( s.Quantity ) OVER (
             PARTITION BY s.ProductID
             ORDER BY s.ProductID )
           AS max_q
           FROM SalesOrderItems s )
  AS order_qty, Products p, SalesOrders o
  WHERE p.ID = ProductID
    AND o.ID = order_qty.ID
    AND p.Quantity < max_q
  ORDER BY p.ID, o.ID;
```

☞ For more information on the full syntax of the MAX and MIN functions, see “MAX function [Aggregate]” [*SQL Anywhere Server - SQL Reference*], and “MIN function [Aggregate]” [*SQL Anywhere Server - SQL Reference*].

Complex analytics

Consider the following query, which lists the top salespeople (defined by total sales) for each product in the database:

```
SELECT s.ProductID AS Products, o.SalesRepresentative,
     SUM( s.Quantity ) AS total_quantity,
     SUM( s.Quantity * p.UnitPrice ) AS total_sales
  FROM SalesOrders o KEY JOIN SalesOrderItems s
    KEY JOIN Products p
  GROUP BY s.ProductID, o.SalesRepresentative
  HAVING total_sales = (
    SELECT First SUM( s2.Quantity * p2.UnitPrice )
        AS sum_sales
      FROM SalesOrders o2 KEY JOIN
        SalesOrderItems s2 KEY JOIN Products p2
      WHERE s2.ProductID = s.ProductID
      GROUP BY o2.SalesRepresentative
      ORDER BY sum_sales DESC )
  ORDER BY s.ProductID;
```



```
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
```

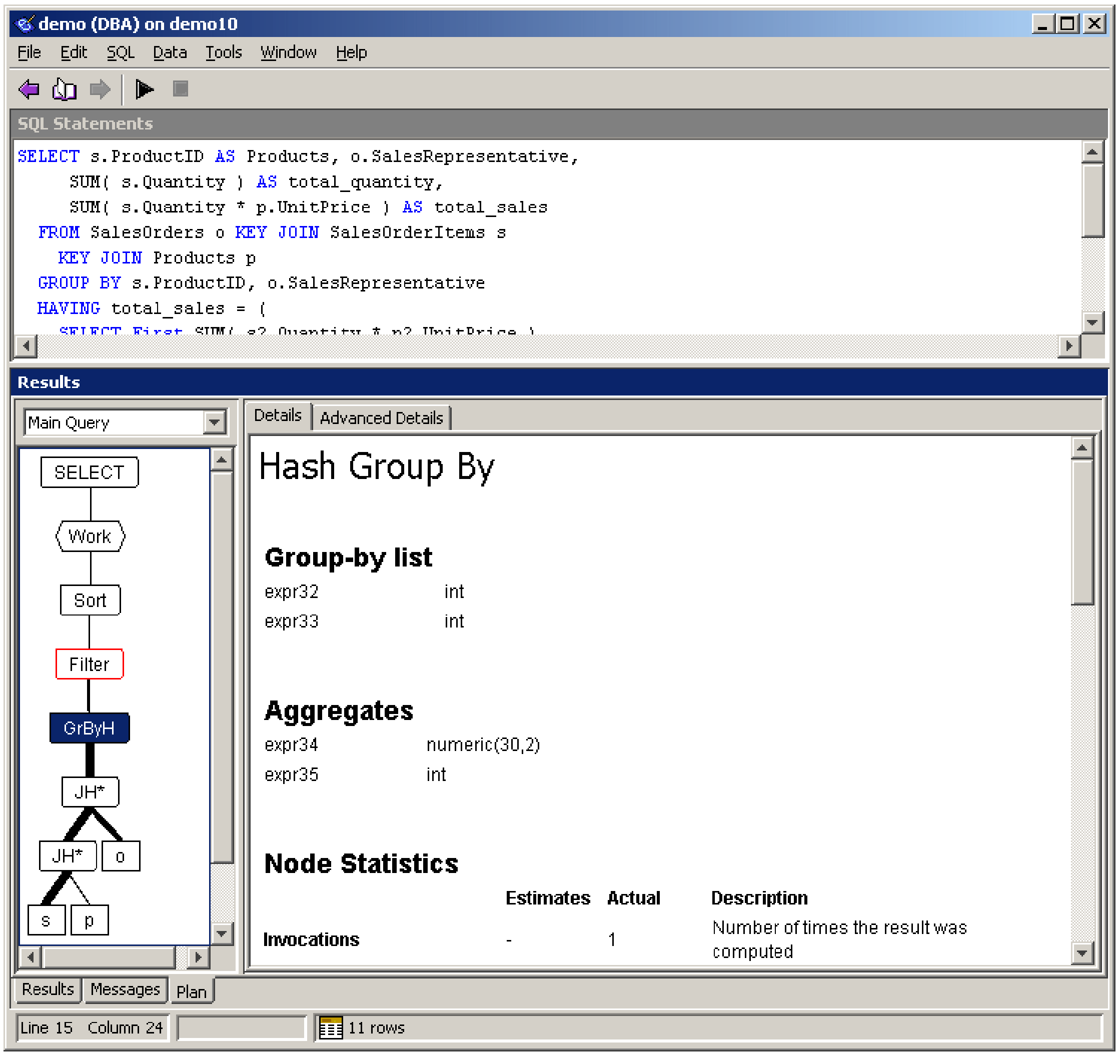
Node Statistics

This query returns the result:

	Products	SalesRepresentative	total_quantity	total_sales
1	300	299	660	5940.00
2	301	299	516	7224.00
3	302	299	336	4704.00
4	400	299	458	4122.00
5	401	902	360	3600.00
6	500	949	360	2520.00

	Products	SalesRepresentative	total_quantity	total_sales
7	501	690	360	2520.00
8	501	949	360	2520.00
9	600	299	612	14688.00
10	601	299	336	15264.00
11	700	299	1008	15120.00

The original query is formed using a correlated subquery that determines the highest sales for any particular product, as ProductID is the subquery's correlated outer reference. Using a nested query, however, is often an expensive option, as in this case. This is because the subquery involves not only a GROUP BY clause, but also an ORDER BY clause within the GROUP BY clause. This makes it impossible for the query optimizer to rewrite this nested query as a join while retaining the same semantics.



```
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 )
```

Hash Group By

Group-by list

expr32	int
expr33	int

Aggregates

expr34	numeric(30,2)
expr35	int

Node Statistics

	Estimates	Actual	Description
Invocations	-	1	Number of times the result was computed

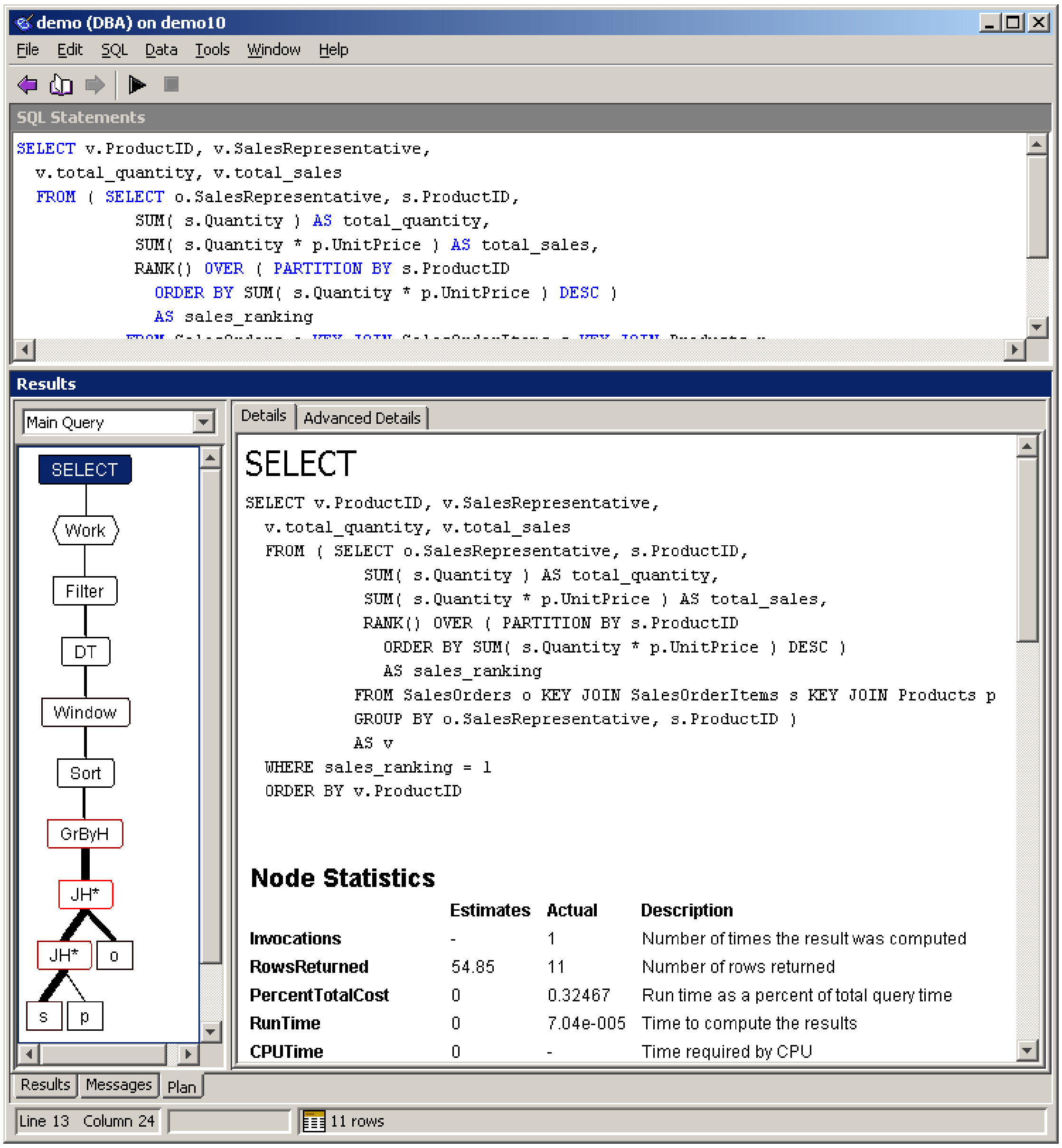
Line 15 Column 24 11 rows

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.

However, a rewrite of the query, using a ranking function, computes the identical result much more efficiently:

```
SELECT v.ProductID, v.SalesRepresentative,
  v.total_quantity, v.total_sales
  FROM ( SELECT o.SalesRepresentative, s.ProductID,
              SUM( s.Quantity ) AS total_quantity,
              SUM( s.Quantity * p.UnitPrice ) AS total_sales,
              RANK() OVER ( PARTITION BY s.ProductID
                ORDER BY SUM( s.Quantity * p.UnitPrice ) DESC )
                AS sales_ranking
              FROM SalesOrders o KEY JOIN SalesOrderItems s KEY JOIN Products p
              GROUP BY o.SalesRepresentative, s.ProductID )
              AS v
WHERE sales_ranking = 1
ORDER BY v.ProductID;
```

This rewritten query results in a simpler plan:



```
SELECT v.ProductID, v.SalesRepresentative,
  v.total_quantity, v.total_sales
  FROM ( SELECT o.SalesRepresentative, s.ProductID,
            SUM( s.Quantity ) AS total_quantity,
            SUM( s.Quantity * p.UnitPrice ) AS total_sales,
            RANK() OVER ( PARTITION BY s.ProductID
              ORDER BY SUM( s.Quantity * p.UnitPrice ) DESC )
              AS sales_ranking
           FROM SalesOrders o KEY JOIN SalesOrderItems s KEY JOIN Products p
           GROUP BY o.SalesRepresentative, s.ProductID )
           AS v
  WHERE sales_ranking = 1
  ORDER BY v.ProductID
```

SELECT

Node Statistics

	Estimates	Actual	Description
Invocations	-	1	Number of times the result was computed
RowsReturned	54.85	11	Number of rows returned
PercentTotalCost	0	0.32467	Run time as a percent of total query time
RunTime	0	7.04e-005	Time to compute the results
CPUTime	0	-	Time required by CPU

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 SalesRepresentatives 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.

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 425.

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 sample consisting of a numeric expression, as a DOUBLE.

Example 1

The following query returns a result set that shows the employees whose salary is one standard deviation greater than the average salary of their department. Standard deviation is a measure of how much the data varies from the mean.

```
SELECT *
FROM ( SELECT
    Surname AS Employee,
    DepartmentID AS Department,
    CAST( Salary as DECIMAL( 10, 2 ) )
        AS Salary,
    CAST( AVG( Salary )
        OVER ( PARTITION BY DepartmentID ) AS DECIMAL ( 10, 2 ) )
        AS Average,
    CAST( STDDEV_POP( Salary )
        OVER ( PARTITION BY DepartmentID ) AS DECIMAL ( 10, 2 ) )
        AS StandardDeviation
    FROM Employees
    GROUP BY Department, Employee, Salary )
    AS DerivedTable
WHERE Salary > Average + StandardDeviation
ORDER BY Department, Salary, Employee;
```

The table that follows represents the result set from the query. Every department has at least one employee whose salary significantly deviates from the mean.

	Employee	DepartmentID	Salary	Average	StandardDeviation
1	Lull	100	87900.00	58736.28	16829.60
2	Scheffield	100	87900.00	58736.28	16829.60
3	Scott	100	96300.00	58736.28	16829.60
4	Sterling	200	64900.00	48390.95	13869.60
5	Savarino	200	72300.00	48390.95	13869.60
6	Kelly	200	87500.00	48390.95	13869.60
7	Shea	300	138948.00	59500.00	30752.40
8	Blaikie	400	54900.00	43640.67	11194.02
9	Morris	400	61300.00	43640.67	11194.02
10	Evans	400	68940.00	43640.67	11194.02
11	Martinez	500	55500.00	33752.20	9084.50

Employee Scott earns $96,300.00, while the departmental average is $58,736.28. The standard deviation for that department is $16,829.00, which means that salaries less than $75,565.88 (`58736.28 + 16829.60`

`= 75565.88`) fall within one standard deviation of the mean. At $96,300.00, employee Scott is well above that figure.

This example assumes that Surname and Salary are unique for each employee, which isn't necessarily true. To ensure uniqueness, you could add EmployeeID to the GROUP BY clause.

Example 2

The following statement lists the average and variance in the number of items per order in different time periods:

```
SELECT YEAR( ShipDate ) AS Year,
    QUARTER( ShipDate ) AS Quarter,
    AVG( Quantity ) AS Average,
    STDDEV_POP( Quantity ) AS Variance
FROM SalesOrderItems
GROUP BY Year, Quarter
ORDER BY Year, Quarter;
```

This query returns the following result:

Year	Quarter	Average	Variance
2000	1	25.775148	14.2794...
2000	2	27.050847	15.0270...
...	...	...	...

☞ For more information on the syntax for this function, see "STDDEV_SAMP function [Aggregate]" [*SQL Anywhere Server - SQL Reference*].

STDDEV_SAMP function

This function computes the standard deviation of a population consisting of a numeric expression, as a DOUBLE. For example, the following statement returns the average and variance in the number of items per order in different quarters:

```
SELECT YEAR( ShipDate ) AS Year,
    QUARTER( ShipDate ) AS Quarter,
    AVG( Quantity ) AS Average,
    STDDEV_SAMP( Quantity ) AS Variance
FROM SalesOrderItems
GROUP BY Year, Quarter
ORDER BY Year, Quarter;
```

This query returns the following result:

Year	Quarter	Average	Variance
2000	1	25.775148	14.3218...
2000	2	27.050847	15.0696...
...	...	...	...

☞ For more information on the syntax for this function, see "STDDEV_POP function [Aggregate]" [*SQL Anywhere Server - SQL Reference*].

VARIANCE function

This function is an alias for the VAR_SAMP function. See "VAR_SAMP function [Aggregate]" [*SQL Anywhere Server - SQL Reference*].

VAR_POP function

This function computes the statistical variance of a population consisting of a numeric expression, as a DOUBLE. For example, the following statement lists the average and variance in the number of items per order in different time periods:

```
SELECT YEAR( ShipDate ) AS Year,
    QUARTER( ShipDate ) AS Quarter,
    AVG( Quantity ) AS Average,
    VAR_POP( quantity ) AS Variance
FROM SalesOrderItems
GROUP BY Year, Quarter
ORDER BY Year, Quarter;
```

This query returns the following result:

Year	Quarter	Average	Variance
2000	1	25.775148	203.9021...
2000	2	27.050847	225.8109...
...	...	...	...

If VAR_POP is computed for a single row, then it returns the value 0.

☞ For more information on the syntax for this function, see "VAR_POP function [Aggregate]" [*SQL Anywhere Server - SQL Reference*].

VAR_SAMP function

This function computes the statistical variance of a sample consisting of a numeric expression, as a DOUBLE.

For example, the following statement lists the average and variance in the number of items per order in different time periods:

```
SELECT YEAR( ShipDate ) AS Year,
    QUARTER( ShipDate ) AS Quarter,
    AVG( Quantity ) AS Average,
    VAR_SAMP( Quantity ) AS Variance
FROM SalesOrderItems
GROUP BY Year, Quarter
ORDER BY Year, Quarter;
```

This query returns the following result:

Year	Quarter	Average	Variance
2000	1	25.775148	205.1158...
2000	2	27.050847	227.0939...
...	...	...	...

If VAR_SAMP is computed for a single row, then it returns NULL.

☞ For more information on 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 on the mathematical formulas represented by these functions see "Mathematical formulas for the aggregate functions" on page 425.

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 on 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 on 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 on 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 on 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 on 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 on 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 on 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 on 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 on 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 on 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 on 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 on 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:

	Description	Quantity	UnitPrice	Rank
1	Zipped Sweatshirt	32	24.00	1
2	Hooded Sweatshirt	39	24.00	1
3	Cotton Shorts	80	15.00	3

Note that rows 1 and 2 have the same value for Unit Price, and therefore also have the same rank. This is called a tie.

With the RANK function, the rank value jumps after a tie. For example, the rank value for row 3 has jumped to three instead of 2. This is different from the DENSE_RANK function, where no jumping occurs after a tie. See "DENSE_RANK function" on page 419.

Example 2

The following SQL query finds the male and female employees from Utah and ranks them in descending order according to salary.

```
SELECT Surname, Salary, Sex,
     RANK() OVER ( ORDER BY Salary DESC ) "Rank"
     FROM Employees
WHERE State IN ( 'UT' );
```

The table that follows represents the result set from the query:

	Surname	Salary	Sex	Rank
1	Shishov	72995.00	F	1
2	Wang	68400.00	M	2
3	Cobb	62000.00	M	3
4	Morris	61300.00	M	4
5	Diaz	54900.00	M	5
6	Driscoll	48023.69	M	6
7	Hildebrand	45829.00	F	7
8	Goggin	37900.00	M	8
9	Rebeiro	34576.00	M	9
10	Bigelow	31200.00	F	10
11	Lynch	24903.00	M	11

Example 3

You can partition your data to provide different results. Using the query from Example 2, you can change the data by partitioning it by gender. The following example ranks employees in descending order by salary and partitions by gender.

```
SELECT Surname, Salary, Sex,
     RANK () OVER ( PARTITION BY Sex
     ORDER BY Salary DESC ) "Rank"
     FROM Employees
WHERE State IN ( 'UT' );
```

The table that follows represents the result set from the query:

	Surname	Salary	Sex	Rank
1	Wang	68400.00	M	1
2	Cobb	62000.00	M	2
3	Morris	61300.00	M	3
4	Diaz	54900.00	M	4
5	Driscoll	48023.69	M	5
6	Goggin	37900.00	M	6
7	Rebeiro	34576.00	M	7
8	Lynch	24903.00	M	8
9	Shishov	72995.00	F	1
10	Hildebrand	45829.00	F	2
11	Bigelow	31200.00	F	3

☞ For more information on 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. If the ORDER BY clause includes multiple expressions, the second and subsequent expressions are used to break ties if the first expression has the same value in adjacent rows. NULL values are sorted before any other value (in ascending sequence).

The DENSE_RANK function returns a series of ranks that are monotonically increasing with no gaps, or jumps in rank value. The term dense is used because there are no jumps in rank value (unlike the RANK function).

As the window moves down the input rows, the rank is calculated for the expression specified in the window's ORDER BY clause. If the ORDER BY clause includes multiple expressions, the second and subsequent expressions are used to break ties if the first expression has the same value in adjacent rows. NULL values are sorted before any other value (in ascending sequence).

Example 1

The following query determines the three most expensive products in the database. A descending sort sequence is specified for the window so that the most expensive products have the lowest rank (rankings start at 1).

```
SELECT Top 3 *
  FROM ( SELECT Description, Quantity, UnitPrice,
    DENSE_RANK() OVER ( ORDER BY UnitPrice DESC ) AS Rank
    FROM Products ) AS DT
  ORDER BY Rank;
```

This query returns the following result:

	Description	Quantity	UnitPrice	Rank
1	Hooded Sweatshirt	39	24.00	1
2	Zipped Sweatshirt	32	24.00	1
3	Cotton Shorts	80	15.00	2

Note that rows 1 and 2 have the same value for Unit Price, and therefore also have the same rank. This is called a tie.

With the DENSE_RANK function, there is no jump in the rank value after a tie. For example, the rank value for row 3 is 2. This is different from the RANK function, where a jump in rank values occurs after a tie. See “RANK function” on page 417.

Example 2

Because windows are evaluated after a query's GROUP BY clause, you can specify complex requests that determine rankings based on the value of an aggregate function.

The following query produces the top three salespeople in each region by their total sales within that region, along with the total sales for each region:

```
SELECT *
  FROM ( SELECT o.SalesRepresentative, o.Region,
           SUM( s.Quantity * p.UnitPrice ) AS total_sales,
           DENSE_RANK() OVER ( PARTITION BY o.Region,
             GROUPING( o.SalesRepresentative )
             ORDER BY total_sales DESC ) AS sales_rank
         FROM Products p, SalesOrderItems s, SalesOrders o
         WHERE p.ID = s.ProductID AND s.ID = o.ID
         GROUP BY GROUPING SETS( ( o.SalesRepresentative, o.Region ),
            o.Region ) ) AS DT
  WHERE sales_rank <= 3
  ORDER BY Region, sales_rank;
```

This query returns the following result:

	SalesRepresentative	Region	total_sales	sales_rank
1	299	Canada	9312.00	1
2	(NULL)	Canada	24768.00	1
3	1596	Canada	3564.00	2
4	856	Canada	2724.00	3

	SalesRepresentative	Region	total_sales	sales_rank
5	299	Central	32592.00	1
6	(NULL)	Central	134568.00	1
7	856	Central	14652.00	2
8	467	Central	14352.00	3
9	299	Eastern	21678.00	1
10	(NULL)	Eastern	142038.00	1
11	902	Eastern	15096.00	2
12	690	Eastern	14808.00	3
13	1142	South	6912.00	1
14	(NULL)	South	45262.00	1
15	667	South	6480.00	2
16	949	South	5782.00	3
17	299	Western	5640.00	1
18	(NULL)	Western	37632.00	1
19	1596	Western	5076.00	2
20	667	Western	4068.00	3

This query combines multiple groupings through the use of GROUPING SETS. 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 on 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.

```
SELECT DepartmentID, Surname, Salary,
    CUME_DIST() OVER ( PARTITION BY DepartmentID
      ORDER BY Salary DESC ) "Rank"
  FROM Employees
  WHERE State IN ( 'CA' );
```

This query returns the following result:

DepartmentID	Surname	Salary	Rank
200	Savarino	72300.00	0.333333333333333
200	Clark	45000.00	0.666666666666667
200	Overbey	39300.00	1

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:

Description	Quantity	UnitPrice	CDist
Wool cap	12	10.00	0.5

☞ For more information on 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:

	DepartmentID	Surname	Salary	Sex	PctRank
1	200	Martel	55700.000	M	0.0
2	100	Guevara	42998.000	M	0.333333333
3	100	Soo	39075.000	M	0.666666667
4	400	Ahmed	34992.000	M	1.0
5	300	Davidson	57090.000	F	0.0
6	400	Blaikie	54900.000	F	0.333333333
7	100	Whitney	45700.000	F	0.666666667
8	400	Wetherby	35745.000	F	1.0

Since the input is partitioned by gender (Sex), PERCENT_RANK is evaluated separately for males and females.

Example 2

The following example returns a list of female employees in Utah and Arizona and ranks them in descending order according to salary. Here, the PERCENT_RANK function is used to provide a cumulative total in descending order.

```
SELECT Surname, Salary,
     PERCENT_RANK () OVER ( ORDER BY Salary DESC ) "Rank"
     FROM Employees
WHERE State IN ( 'UT', 'AZ' ) AND Sex IN ( 'F' );
```

This query returns the following results:

	Surname	Salary	Rank
1	Shishov	72995.00	0
2	Jordan	51432.00	0.25
3	Hildebrand	45829.00	0.5
4	Bigelow	31200.00	0.75
5	Bertrand	29800.00	1

Using PERCENT_RANK to find top and bottom percentiles

You can use PERCENT_RANK to find the top or bottom percentiles in the data set. In the following example, the query returns male employees whose salary is in the top five percent of the data set.

```
SELECT *
FROM ( SELECT Surname, Salary,
       PERCENT_RANK () OVER ( ORDER BY Salary DESC ) "Rank"
       FROM Employees
       WHERE Sex IN ( 'M' )   )
       AS DerivedTable ( Surname, Salary, Percent )
WHERE Percent < 0.05;
```

This query returns the following results:

	Surname	Salary	Percent
1	Scott	96300.00	0
2	Sheffield	87900.00	0.025
3	Lull	87900.00	0.025

☞ For more information on 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:

Description	Quantity	RowNum
Tank Top	28	1
V-neck	54	2
Crew Neck	75	3

As with the ranking functions, ROW_NUMBER requires an ORDER BY clause.

As well, ROW_NUMBER can return non-deterministic results when the window's ORDER BY clause is over non-unique expressions; row order is unpredictable 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 on 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

Function	Symbol	Formula
SUM(X)		$\sum_{i=1}^{n} x_i$
MAX(X)		$x_i : x_i \geq x_j, i \neq j \; \forall \, i, j \in n$
MIN(X)		$x_i : x_i \leq x_j, i \neq j \; \forall \, i, j \in n$
AVG(X)	$\bar{x}$	$\frac{\sum x_i}{n}$
COUNT(*)		n
VAR_SAMP(X)	s_x^2	$\frac{\sum (x_i - \bar{x})^2}{(n-1)}$
VAR_POP(X)	σ_x^2	$\frac{\sum (x_i - \bar{x})^2}{n}$
VARIANCE(X)		identical to VAR_SAMP(X)
STDDEV_SAMP(X)	s_x	$\sqrt{\frac{\sum (x_i - \bar{x})^2}{(n-1)}}$
STDDEV_POP(X)	σ_x	$\sqrt{\frac{\sum (x_i - \bar{x})^2}{n}}$
STDDEV(X)		identical to STDDEV_SAMP(X)

Statistical aggregate functions

COVAR_SAMP(Y,X)	*Co-variance*	$s_{xy} = \frac{\sum xy - \frac{(\sum x)(\sum y)}{n}}{(n-1)}$
COVAR_POP(Y,X)	*Co-variance*	$\sigma_{xy} = \frac{\sum xy - \frac{(\sum x)(\sum y)}{n}}{n}$
CORR(Y,X)	*Correlation Coefficient*	$r = \frac{\sum xy - \frac{1}{n}(\sum x)(\sum y)}{(n-1) s_x s_y}$
REGR_AVGX(Y,X)	*Independent mean*	$\bar{x}$
REGR_AVGY(Y,X)	*Dependent mean*	$\bar{y}$
REGR_SLOPE(Y,X)	*Regression Slope*	$b = r \frac{s_y}{s_x}$
REGR_INTERCEPT(Y,X)	*Regression Intercept*	$a = \bar{y} - b\bar{x}$
REGR_R2(Y,X)	*'Goodness-of-fit'*	r^2
REGR_COUNT(Y,X)	*Sample size*	n (non-null (Y, X) pairs)
REGR_SXX(Y,X)	*Sum of squares* (x)	$\sum x^2 - \frac{(\sum x)^2}{n}$
REGR_SYY(Y,X)	*Sum of squares* (y)	$\sum y^2 - \frac{(\sum y)^2}{n}$
REGR_SXY(Y,X)	*Sum of products*	$\sum xy - \frac{(\sum y)(\sum x)}{n}$

CHAPTER 12

Using Subqueries

Contents

About this chapter

This chapter shows how to use the results of one query as part of another SELECT statement. This is a useful tool in building more complex and informative queries.

When you create a query, you use WHERE and HAVING clauses to restrict the rows that the query returns. Sometimes, the rows you select depend on information stored in more than one table. A subquery in the WHERE or HAVING clause allows you to select rows from one table according to specifications obtained from another table. Additional ways to do this can be found in "Joins: Retrieving Data from Several Tables" on page 321.

Before your start

This chapter assumes some knowledge of queries and the syntax of the SELECT statement. Information about queries appears in "Queries: Selecting Data from a Table" on page 263.

Introduction to subqueries

A relational database allows you to store related data in more than one table. You can extract data from related tables using **subqueries**—queries that appear in another query's WHERE clause or HAVING clause. Subqueries make some queries easier to write than joins, and there are queries that cannot be written without using subqueries.

Subqueries use the results of one query as part of another query. This section illustrates a situation where subqueries can be used by building a query that lists order items for products that are low in stock.

There are two queries involved in producing this list. This section first describes them separately, and then shows the single query that produces the same result.

For example, 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.

In general, only the simplest questions can be answered using only one table. For example, if the 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, if "nearly out of stock" depends on how many items of each type the typical customer orders, the number "50" will have to be replaced by a value obtained from the SalesOrderItems table.

Structure of the subquery

A subquery is structured like a regular query, and appears in the main query's SELECT, FROM, WHERE, or HAVING clause. Continuing with the previous example, you can use a subquery to select the average number of items that a customer orders, and then use that figure in the main query to find products that are nearly out of stock. The following query finds the names and descriptions of the products which number less than twice the average number of items of each type that a customer orders.

```
SELECT Name, Description
FROM Products WHERE Quantity <  2 * (
   SELECT AVG( Quantity )
   FROM SalesOrderItems
   );
```

In the WHERE clause, subqueries help select the rows from the tables listed in the FROM clause that appear in the query results. In the HAVING clause, they help select the row groups, as specified by the main query's GROUP BY clause, that appear in the query results.

Simple examples

Following are three simple examples of using subqueries.

♦ List all products for which there are less than 20 items in stock

- In Interactive SQL, execute the following statement:

```
SELECT ID, Description, Quantity
FROM Products
WHERE Quantity < 20;
```

ID	Description	Quantity
401	Wool cap	12

The query shows that only wool caps are low in stock.

♦ **List all order items for wool caps**

- In Interactive SQL, execute the following statement:

```
SELECT *
FROM SalesOrderItems
WHERE ProductID = 401
ORDER BY ShipDate DESC;
```

ID	LineID	ProductID	Quantity	ShipDate
2082	1	401	48	7/9/2001
2053	1	401	60	6/30/2001
2125	2	401	36	6/28/2001
2027	1	401	12	6/17/2001
…	…	…	…	…

This two-step process of identifying items low in stock and identifying orders for those items can be combined into a single query using subqueries.

♦ **List all order items for products that are low in stock**

- In Interactive SQL, execute the following statement:

```
SELECT *
FROM SalesOrderItems
WHERE ProductID IN
   (  SELECT ID
       FROM Products
       WHERE Quantity < 20 )
ORDER BY ShipDate DESC;
```

ID	LineID	ProductID	Quantity	ShipDate
2082	1	401	48	7/9/2001
2053	1	401	60	6/30/2001
2125	2	401	36	6/28/2001

ID	LineID	ProductID	Quantity	ShipDate
2027	1	401	12	6/17/2001
…	…	…	…	…

The subquery in the statement is the phrase enclosed in parentheses:

```
( SELECT ID
    FROM Products
    WHERE Quantity < 20 );
```

The subquery makes a list of all values in the ID column in the Products table, satisfying the WHERE clause search condition.

The subquery 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.

Single-row and multiple-row subqueries

There are constraints on the number of rows and columns that a subquery can return. If you use IN, ANY, or ALL, the subquery may return several rows, but only one column. If you use other operators, the subquery must return a single value.

A multiple-row subquery

Two tables in the SQL Anywhere sample database contain financial results data. The FinancialCodes table is a table holding the different codes for financial data and their meaning. To list the revenue items from the FinancialData table, execute the following query:

```
SELECT *
FROM FinancialData
WHERE FinancialData.Code IN
    ( SELECT FinancialCodes.Code
        FROM FinancialCodes
        WHERE type = 'revenue' );
```

Year	Quarter	Code	Amount
1999	Q1	r1	1023
1999	Q2	r1	2033
1999	Q3	r1	2998
1999	Q4	r1	3014
2000	Q1	r1	3114

This example uses qualifiers to clearly identify the table to which the Code column in each reference belongs. In this particular example, the qualifiers could have been omitted.

Two other keywords can be used as qualifiers for operators to allow them to work with multiple rows: ANY and ALL.

The following query is identical to the successful query above:

```
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' );
```

A common error using subqueries

In general, subquery result sets are restricted to a single column. The following example does not make sense because SQL Anywhere does not know which column from FinancialCodes to compare to the FinancialData.Code column.

```
-- this query returns an error
SELECT *
FROM FinancialData
WHERE FinancialData.Code IN
    ( SELECT FinancialCodes.Code, FinancialCodes.type
      FROM FinancialCodes
      WHERE type = 'revenue' );
```

Single-row subqueries

While subqueries used with an IN condition may return a set of rows, a subquery used with a comparison operator must return only one row. For example the following command results in an error since the subquery returns two rows:

```
-- this query returns an error
SELECT *
FROM FinancialData
WHERE FinancialData.Code =
    ( SELECT FinancialCodes.Code
            FROM FinancialCodes
            WHERE type = 'revenue' );
```

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;
```

ID	OrderDate	CompanyName
2488	1/15/2001	North Land Trading
2513	2/05/2001	The Hat Company
2518	2/10/2001	Sports Replay
2049	2/17/2001	Cooper Inc.
...	...	...

Using a subquery

The following statement obtains the same results using a subquery instead of a join:

```
SELECT SalesOrders.ID,
   SalesOrders.OrderDate,
   (  SELECT CompanyName FROM Customers
       WHERE Customers.ID = SalesOrders.CustomerID )
FROM SalesOrders
WHERE OrderDate > '2001/01/01'
ORDER BY OrderDate;
```

The subquery refers to the CustomerID column in the SalesOrders table even though the SalesOrders table is not part of the subquery. Instead, the SalesOrders.CustomerID column refers to the SalesOrders table in the main body of the statement. This is called an **outer reference**. Any subquery that contains an outer reference is called a **correlated subquery**.

A subquery can be used instead of a join whenever only one column is required from the other table. (Recall that subqueries can only return one column.) In this example, you only needed the CompanyName column, so the join could be changed into a subquery.

Using an outer join

To list all customers in Washington state, together with their most recent order ID, execute the following query:

```
SELECT  CompanyName, State,
   ( SELECT MAX( ID )
       FROM SalesOrders
      WHERE SalesOrders.CustomerID = Customers.ID )
FROM Customers
WHERE State = 'WA';
```

CompanyName	State	MAX(SalesOrders.ID)
Custom Designs	WA	2547
It's a Hit!	WA	(NULL)

The It's a Hit! company placed no orders, and the subquery returns NULL for this customer. Companies who have not placed an order are not listed when inner joins are used.

You could also specify an outer join explicitly. In this case, a GROUP BY clause is also required.

```
SELECT CompanyName, State,
   MAX( SalesOrders.ID )
FROM Customers
   KEY LEFT OUTER JOIN SalesOrders
WHERE State = 'WA'
GROUP BY CompanyName, State;
```

Using subqueries in the WHERE clause

Subqueries in the WHERE clause work as part of the row selection process. You use a subquery in the WHERE clause when the criteria you use to select rows depend on the results of another table.

Example

Find the products whose in-stock quantities are less than double the average ordered quantity.

```
SELECT Name, Description
FROM Products WHERE Quantity <  2 * (
   SELECT AVG( Quantity )
   FROM SalesOrderItems );
```

This is a two-step query: first, find the average number of items requested per order; and then find which products in stock number less than double that quantity.

The query in two steps

The Quantity column of the SalesOrderItems table stores the *number* of items requested per item type, customer, and order. The subquery is

```
SELECT AVG( Quantity )
FROM SalesOrderItems;
```

It returns the average quantity of items in the SalesOrderItems table, which is 25.851413.

The next query returns the names and descriptions of the items whose in-stock quantities are less than twice the previously-extracted value.

```
SELECT Name, Description
FROM Products
WHERE Quantity < 2*25.851413;
```

Using a subquery combines the two steps into a single operation.

Purpose of a subquery in the WHERE clause

A subquery in the WHERE clause is part of a search condition. The chapter “Queries: Selecting Data from a Table” on page 263 describes simple search conditions you can use in the WHERE clause.

Subqueries in the HAVING clause

Although you usually use subqueries as search conditions in the WHERE clause, sometimes you can also use them in the HAVING clause of a query. When a subquery appears in the HAVING clause, like any expression in the HAVING clause, it is used as part of the row group selection.

Here is a request that lends itself naturally to a query with a subquery in the HAVING clause: "Which products' average in-stock quantity is more than double the average number of each item ordered per customer?"

Example

```
SELECT Name, AVG( Quantity )
FROM Products
GROUP BY Name
HAVING AVG( Quantity ) > 2* (
   SELECT AVG( Quantity )
   FROM SalesOrderItems
 );
```

name	AVG(Products.Quantity)
Baseball Cap	62
Shorts	80

The query executes as follows:

- The subquery calculates the average quantity of items in the SalesOrderItems table.
- The main query then goes through the Products table, calculating the average quantity per product, grouping by product name.
- The HAVING clause then checks if each average quantity is more than double the quantity found by the subquery. If so, the main query returns that row group; otherwise, it doesn't.
- The SELECT clause produces one summary row for each group, displaying the name of each product and its in-stock average quantity.

You can also use outer references in a HAVING clause, as shown in the following example, a slight variation on the one above.

Example

This example finds the product ID numbers and line ID numbers of those products whose average ordered quantities is more than half the in-stock quantities of those products.

```
SELECT ProductID, LineID
FROM SalesOrderItems
GROUP BY ProductID, LineID
HAVING 2* AVG( Quantity ) > (
   SELECT Quantity
   FROM Products
   WHERE Products.ID = SalesOrderItems.ProductID );
```

ProductID	LineID
601	3
601	2
601	1
600	2
…	…

In this example, the subquery must produce the in-stock quantity of the product corresponding to the row group being tested by the HAVING clause. The subquery selects records for that particular product, using the outer reference SalesOrderItems.ProductID**.**

A subquery with a comparison returns a single value

This query uses the comparison >, suggesting that the subquery must return exactly one value. In this case, it does. Since the ID field of the Products table is a primary key, there is only one record in the Products table corresponding to any particular product ID.

Subquery tests

The chapter "Queries: Selecting Data from a Table" on page 263 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.
- **Quantified comparison test** Compares the value of an expression to each of the set of values produced by a subquery.
- **Subquery set membership test** Checks if the value of an expression matches one of the set of values produced by a subquery.
- **Existence test** Checks if the subquery produces any rows.

Subquery comparison test

The subquery comparison test (=, <>, <. <=, >, >=) is a modified version of the simple comparison test. The only difference between the two is that in the former, the expression following the operator is a subquery. This test is used to compare a value from a row in the main query to a *single* value produced by the subquery.

Example

This query contains an example of a subquery comparison test:

```
SELECT Name, Description, Quantity
FROM Products
WHERE Quantity <  2 * (
   SELECT AVG( Quantity )
   FROM SalesOrderItems );
```

name	Description	Quantity
Tee Shirt	Tank Top	28
Baseball Cap	Wool cap	12
Visor	Cloth Visor	36
Visor	Plastic Visor	28
…	…	…

The following subquery retrieves a single value—the average quantity of items of each type per customer's order—from the SalesOrderItems table.

```
SELECT AVG( Quantity )
FROM SalesOrderItems;
```

Then the main query compares the quantity of each in-stock item to that value.

A subquery in a comparison test returns one value

A subquery in a comparison test must return exactly one value. Consider this query, whose subquery extracts two columns from the SalesOrderItems table:

```
SELECT Name, Description, Quantity
FROM Products
WHERE Quantity <  2 * (
   SELECT AVG( Quantity ), MAX( Quantity )
   FROM SalesOrderItems);
```

It returns the error `Subquery allowed only one select list item.`

Quantified comparison tests with ANY and ALL

The quantified comparison test has two categories, the ALL test and the ANY test:

The ANY test

The ANY test, used in conjunction with one of the SQL comparison operators (=, <>, <, <=, >, >=), compares a single value to the column of data values produced by the subquery. To perform the test, SQL uses the specified comparison operator to compare the test value to each data value in the column. If *any* of the comparisons yields a TRUE result, the ANY test returns TRUE.

A subquery used with ANY must return a single column.

Example

Find the order and customer IDs of those orders placed after the first product of the order #2005 was shipped.

```
SELECT ID, CustomerID
FROM SalesOrders
WHERE OrderDate > ANY (
   SELECT ShipDate
   FROM SalesOrderItems
   WHERE ID=2005 );
```

ID	CustomerID
2006	105
2007	106
2008	107
2009	108
…	…

In executing this query, the main query tests the order dates for each order against the shipping dates of *every* product of the order #2005. If an order date is greater than the shipping date for *one* shipment of order #2005, then that ID and customer ID from the SalesOrders table are part of the result set. The ANY test is 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 NULL. 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.

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 );
```

ID	CustomerID
2002	102
2003	103
2004	104
2005	101
…	…

In executing this query, the main query tests the order dates for each order against the shipping dates of *every* product of order #2001. If an order date is greater than the shipping date for *every* shipment of order #2001, then the ID and customer ID from the SalesOrders table are part of the result set. The ALL test is 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)
  ```

  For more information about this test, see "Quantified comparison test" on page 452.

Testing set membership with IN conditions

You can use the subquery set membership test to compare a value from the main query to more than one value in the subquery.

The subquery set membership test compares a single data value for each row in the main query to the single column of data values produced by the subquery. If the data value from the main query matches *one* of the data values in the column, the subquery returns TRUE.

Example

Select the names of the employees who head the Shipping or Finance departments:

```
SELECT GivenName, Surname
FROM Employees
WHERE EmployeeID IN (
   SELECT DepartmentHeadID
   FROM Departments
   WHERE ( DepartmentName='Finance' OR
          DepartmentName = 'Shipping' ) );
```

GivenName	Surname
Mary Anne	Shea
Jose	Martinez

The subquery in this example extracts from the Departments table the ID numbers that correspond to the heads of the Shipping and Finance departments. The main query then returns the names of the employees whose ID numbers match one of the two found by the subquery.

```
SELECT DepartmentHeadID
FROM Departments
WHERE ( DepartmentName='Finance' OR
       DepartmentName = 'Shipping' );
```

Set membership test is equivalent to =ANY test

The subquery set membership test is equivalent to the =ANY test. The following query is equivalent to the query from the above example.

```
SELECT GivenName, Surname
FROM Employees
WHERE EmployeeID = ANY (
   SELECT DepartmentHeadID
   FROM Departments
   WHERE ( DepartmentName='Finance' OR
          DepartmentName = 'Shipping' ) );
```

Negation of the set membership test

You can also use the subquery set membership test to extract those rows whose column values are not equal to any of those produced by a subquery. To negate a set membership test, insert the word NOT in front of the keyword IN.

Example

The subquery in this query returns the first and last names of the employees that are not heads of the Finance or Shipping departments.

```
SELECT GivenName, Surname
FROM Employees
WHERE EmployeeID NOT IN (
   SELECT DepartmentHeadID
   FROM Departments
   WHERE ( DepartmentName='Finance' OR
          DepartmentName = 'Shipping' ) );
```

Existence test

Subqueries used in the subquery comparison test and set membership test both return data values from the subquery table. Sometimes, however, you may be more concerned with whether the subquery returns *any* results, rather than *which* results. The existence test (EXISTS) checks whether a subquery produces any rows of query results. If the subquery produces one or more rows of results, the EXISTS test returns TRUE. Otherwise, it returns FALSE.

Example

Here is an example of a request expressed using a subquery: "Which customers placed orders after July 13, 2001?"

```
SELECT GivenName, Surname
FROM Customers
WHERE EXISTS (
   SELECT *
   FROM SalesOrders
   WHERE ( OrderDate > '2001-07-13' ) AND
         ( Customers.ID = SalesOrders.CustomerID ) );
```

GivenName	Surname
Almen	de Joie
Grover	Pendelton
Ling Ling	Andrews
Bubba	Murphy

Explanation of the existence test

Here, for each row in the Customers table, the subquery checks if that customer ID corresponds to one that has placed an order after July 13, 2001. If it does, the query extracts the first and last names of that customer from the main table.

The EXISTS test does not use the results of the subquery; it just checks if the subquery produces any rows. So the existence test applied to the following two subqueries return the same results. These are subqueries and cannot be processed on their own, because they refer to the Customers table which is part of the main query, but not part of the subquery.

☞ For more information, see “Correlated subqueries” on page 450.

```
SELECT *
FROM SalesOrders
WHERE ( OrderDate > '2001-07-13' ) AND
      ( Customers.ID = SalesOrders.CustomerID )

SELECT OrderDate
FROM 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.

Outer references

Within the body of a subquery, it is often necessary to refer to the value of a column in the active row of the main query. Consider the following query:

```
SELECT Name, Description
FROM Products
WHERE Quantity < 2 * (
   SELECT AVG( Quantity )
   FROM SalesOrderItems
   WHERE Products.ID = SalesOrderItems.ProductID );
```

This 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 SQL 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.

Description of an outer reference

The Products.ID column in this subquery is an example of an outer reference. A subquery that uses an outer reference is a correlated subquery. An outer reference is a column name that does not refer to any of the columns in any of the tables in the FROM clause of the subquery. Instead, the column name refers to a column of a table specified in the FROM clause of the main query. As the above example shows, the value of a column in an outer reference comes from the row currently being tested by the main query.

Subqueries and joins

The query optimizer automatically rewrites as joins many of the queries that make use of subqueries.

Example

Consider the request, "When did Mrs. Clarke and Suresh place their orders, and by which sales representatives?" It can be answered with the following query:

```
SELECT OrderDate, SalesRepresentative
FROM SalesOrders
WHERE CustomerID IN (
   SELECT ID
   FROM Customers
   WHERE Surname = 'Clarke' OR GivenName = 'Suresh' );
```

OrderDate	SalesRepresentative
2001-01-05	1596
2000-01-27	667
2000-11-11	467
2001-02-04	195
…	…

The subquery yields a list of customer IDs that correspond to the two customers whose names are listed in the WHERE clause, and the main query finds the order dates and sales representatives corresponding to those two people's orders.

Replacing a subquery with a join

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.

Some joins cannot be written as subqueries

Both of these queries find the correct order dates and sales representatives, and neither is more right than the other. Many people will find the subquery form more natural, because the request doesn't ask for any information about customer IDs, and because it might seem odd to join the SalesOrders and Customers tables together to answer the question.

If, however, the request changes to include some information from the Customers table, the subquery form no longer works. For example, the request "When did Mrs. Clarke and Suresh place their orders, and by which representatives, and what are their full names?", it is necessary to include the Customers table in the main WHERE clause:

```
SELECT GivenName, Surname, OrderDate, SalesRepresentative
FROM SalesOrders, Customers
WHERE CustomerID=Customers.ID AND
  ( Surname = 'Clarke' OR GivenName = 'Suresh' );
```

GivenName	Surname	OrderDate	SalesRepresentative
Belinda	Clarke	2001-01-05	1596
Belinda	Clarke	2000-01-27	667
Belinda	Clarke	2000-11-11	467
Belinda	Clarke	2001-02-04	195
…	…	…	…

Some subqueries cannot be written as joins

Similarly, there are cases where a subquery will work but a join will not. For example:

```
SELECT Name, Description, Quantity
FROM Products
WHERE Quantity <  2 * (
   SELECT AVG( Quantity )
   FROM SalesOrderItems );
```

name	Description	Quantity
Tee Shirt	Tank Top	28
Baseball Cap	Wool cap	12
Visor	Cloth Visor	36
…	…	…

In this case, the inner query is a summary query and the outer query is not, so there is no way to combine the two queries by a simple join.

For more information on joins, see “Joins: Retrieving Data from Several Tables” on page 321.

Nested subqueries

As you have seen, subqueries usually appear in the HAVING clause or the WHERE clause of a query. A subquery may itself contain a WHERE clause and/or a HAVING clause, and, consequently, a subquery may appear in another subquery. Subqueries inside other subqueries are called **nested subqueries**.

Examples

List the order IDs and line IDs of those orders shipped on the same day when any item in the fees department was ordered.

```
SELECT ID, LineID
FROM SalesOrderItems
WHERE ShipDate = ANY (
   SELECT OrderDate
   FROM SalesOrders
   WHERE FinancialCode IN (
      SELECT Code
      FROM FinancialCodes
      WHERE ( Description = 'Fees' ) ) );
```

ID	LineID
2001	1
2001	2
2001	3
2002	1
…	…

Explanation of the nested subqueries

- 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 );
```

Nested subqueries can also have more than three levels. Though there is no maximum number of levels, queries with three or more levels take considerably longer to run than do smaller queries.

How subqueries work

Understanding which queries are valid and which ones aren't can be complicated when a query contains a subquery. Similarly, figuring out what a multi-level query does can also be very involved, and it helps to understand how SQL Anywhere processes subqueries. For general information about processing queries, see "Summarizing, Grouping, and Sorting Query Results" on page 295.

Correlated subqueries

In a simple query, the database server evaluates and processes the query's WHERE clause once for each row of the query. Sometimes, though, the subquery returns only one result, making it unnecessary for the database server to evaluate it more than once for the entire result set.

Uncorrelated subqueries

Consider this query:

```
SELECT Name, Description
FROM Products
WHERE Quantity <  2 * (
   SELECT AVG( Quantity )
   FROM SalesOrderItems );
```

In this example, 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.

Correlated subqueries

When a subquery contains an outer reference, you cannot use this shortcut. For instance, the subquery in the following query returns a value dependent upon the active row in the Products table. Such a subquery is called a correlated subquery. In these cases, the subquery might return a different value for each row of the outer query, making it necessary for the database server to perform more than one evaluation.

```
SELECT Name, Description
FROM Products
WHERE Quantity < 2 * (
   SELECT AVG( Quantity )
   FROM SalesOrderItems
   WHERE Products.ID=SalesOrderItems.ProductID );
```

Converting subqueries in the WHERE clause to joins

The SQL Anywhere query optimizer converts some multi-level queries to use joins. 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.

Example

The question "When did Mrs. Clarke and Suresh place their orders, and by which sales representatives?" can be written as a two-level query:

```
SELECT OrderDate, SalesRepresentative
FROM SalesOrders
WHERE CustomerID IN (
   SELECT ID
   FROM Customers
   WHERE Surname = 'Clarke' OR GivenName = 'Suresh' );
```

An alternate, and equally correct, way to write the query uses joins:

```
SELECT GivenName, Surname, OrderDate, SalesRepresentative
FROM SalesOrders, Customers
WHERE CustomerID=Customers.ID AND
  ( Surname = 'Clarke' OR GivenName = 'Suresh' );
```

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. 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*

Whether a subquery can be converted to a join depends on a number of factors, such as the type of operator and the structures of the query and of the subquery.

Comparison operators

A subquery that follows a comparison operator (=, <>, <, <=, >, >=) must satisfy certain conditions if it is to be converted into a join. For example, subqueries that follow comparison operators are valid only if they return exactly one value for each row of the main query. In addition to this criterion, a subquery is converted to a join only if the subquery:

- does not contain a GROUP BY clause
- does not contain the keyword DISTINCT
- is not a UNION query
- is not an aggregate query

Example

Suppose the request "When were Suresh's products ordered, and by which sales representative?" were phrased as the subquery

```
SELECT OrderDate, SalesRepresentative
FROM SalesOrders
WHERE CustomerID = (
   SELECT ID
```

```
      FROM Customers
      WHERE GivenName = 'Suresh' );
```

This query satisfies the criteria, and therefore, it would be converted to a query using a join:

```
SELECT OrderDate, SalesRepresentative
FROM SalesOrders, Customers
WHERE CustomerID=Customers.ID AND
  ( Surname = 'Clarke' OR GivenName = 'Suresh' );
```

However, the request, "Find the products whose in-stock quantities are less than double the average ordered quantity" cannot be converted to a join, as the subquery contains the AVG aggregate function:

```
SELECT Name, Description
FROM Products
WHERE Quantity <  2 * (
   SELECT AVG( Quantity )
   FROM SalesOrderItems );
```

Quantified comparison test

A subquery that follows one of the keywords ALL, ANY and SOME is converted into a join only if it satisfies certain criteria.

- 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:

NOT *column-name operator* **ANY (** *subquery-expression* **)**

column-name inverse-operator **ALL (** *subquery-expression* **)**

These expressions are generally equivalent as well:

NOT *column-name operator* **ALL (** *subquery-expression* **)**

column-name inverse-operator **ANY (** *subquery-expression* **)**

where *inverse-operator* is obtained by negating *operator*, as shown in the table below:

operator	inverse-operator
=	<>
<	=>
>	=<
=<	>
=>	<
<>	=

Set membership test

A query containing a subquery that follows the keyword IN is converted into a join 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 query:

```
SELECT GivenName, Surname
FROM Employees
WHERE EmployeeID IN (
   SELECT DepartmentHeadID
   FROM Departments
   WHERE ( DepartmentName ='Finance' OR
          DepartmentName = 'Shipping' ) );
```

would be converted to a joined query, as it satisfies the conditions. 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 can't be converted

```
SELECT GivenName, Surname
FROM Employees
WHERE EmployeeID IN (
   SELECT DepartmentHeadID
   FROM Departments
   WHERE ( DepartmentName='Finance' OR
          DepartmentName = 'Shipping' )
   UNION
   SELECT CustomerID
   FROM SalesOrders);
```

Similarly, the request "Find the names of employees who are not department heads" is formulated as the negated subquery

```
SELECT GivenName, Surname
FROM Employees
  WHERE NOT EmployeeID IN (
   SELECT DepartmentHeadID
   FROM Departments
   WHERE ( DepartmentName='Finance' OR
          DepartmentName = 'Shipping' ) );
```

and would not be converted.

The conditions that must be fulfilled for a subquery that follows the IN keyword and the ANY keyword to be converted to a join are identical. This is not a coincidence, and the reason for this is that the expression

A query with an IN operator can be converted to one with an ANY operator

WHERE *column-name* **IN(** *subquery-expression* **)**

is logically equivalent to the expression

WHERE *column-name* = **ANY(** *subquery-expression* **)**

So the query

```
SELECT GivenName, Surname
FROM Employees
WHERE EmployeeID IN (
   SELECT DepartmentHeadID
   FROM Departments
   WHERE ( DepartmentName='Finance' OR
          DepartmentName = 'Shipping' ) );
```

is equivalent to the query

```
SELECT GivenName, Surname
FROM Employees
WHERE EmployeeID = ANY (
   SELECT DepartmentHeadID
   FROM Departments
   WHERE ( DepartmentName='Finance' OR
          DepartmentName = 'Shipping' ) );
```

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.

Existence test

A subquery that follows the keyword EXISTS is converted to a join only if it satisfies the following two conditions:

- The main query does not contain a GROUP BY clause, and is not an aggregate query, or the subquery returns exactly one value.
- The conjunct 'EXISTS (subquery)' is not negated.
- The subquery is correlated; that is, it contains an outer reference.

Example

The request, "Which customers placed orders after July 13, 2001?", which can be formulated by a query whose non-negated subquery contains the outer reference **Customers.ID** = **SalesOrders.CustomerID**, can be represented with the following join:

```
SELECT GivenName, Surname
FROM Customers
WHERE EXISTS (
   SELECT *
   FROM SalesOrders
   WHERE ( OrderDate > '2001-07-13' ) AND
         ( Customers.ID = SalesOrders.CustomerID ) );
```

The EXISTS keyword tells the database server to check for empty result sets. When using inner joins, the database server automatically displays only the rows where there is data from all of the tables in the FROM clause. So, this query returns the same rows as does the one with the subquery:

```
SELECT DISTINCT GivenName, Surname
FROM Customers, SalesOrders
WHERE ( SalesOrders.OrderDate > '2001-07-13' ) AND
      ( Customers.ID = SalesOrders.CustomerID );
```

CHAPTER 13

Adding, Changing, and Deleting Data

Contents

About this chapter

This chapter describes how to modify the data in a database.

Most of the chapter is devoted to the INSERT, UPDATE, and DELETE statements, as well as statements for bulk loading and unloading.

Data modification statements

The statements you use to add, change, or delete data are called **data modification** statements. The most common such statements include:

- **Insert** adds new rows to a table
- **Update** changes existing rows in a table
- **Delete** 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 common statements, the LOAD TABLE and TRUNCATE TABLE statements are especially useful for bulk loading and deleting of data.

Sometimes, the data modification statements are collectively known as the **data modification language** (DML) part of SQL.

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 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.

☞ For more information about transactions, see "Using Transactions and Isolation Levels" on page 111.

Grouping changes into 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: database 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.

Options in Interactive SQL

Interactive SQL provides you with two options that let you control when and how transactions end:

- ♦ If you set the option auto_commit to On, Interactive SQL automatically commits your results following every successful statement and automatically performs a ROLLBACK after each failed statement.
- ♦ 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.

Using a data source in Interactive SQL
By default, ODBC operates in autocommit mode. Even if you have set the auto_commit option to Off in Interactive SQL, 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 further information, see "Transact-SQL Compatibility" on page 564.

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.

☞ For more information on Interactive SQL options, see “Interactive SQL options” [*SQL Anywhere Server - Database Administration*].

Use COMMIT with care
When trying the examples in this tutorial, be careful not to COMMIT any changes until you are sure that you want to change the database permanently.
For more information on the COMMIT statement, see “COMMIT statement” [*SQL Anywhere Server - SQL Reference*].

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.

The ROLLBACK statement undoes all changes you have made to the database since the last time you made changes permanent.

☞ For more information, 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.

☞ For a detailed description of data recovery mechanisms, see “Backup and Data Recovery” [*SQL Anywhere Server - Database Administration*].

Integrity checking

SQL Anywhere automatically checks for some common errors in your data.

Inserting duplicate data

For example, suppose you attempt to create a department, but supply a DepartmentID value that is already in use:

To do this, enter the command:

```
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:

```
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 rep ID. This kind of validity checking is called **referential integrity** checking as it maintains the integrity of references among the tables in the database.

☞ For more information on primary and foreign keys, see “Relations between tables” [*SQL Anywhere 10 - Introduction*].

Errors 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.

Example: DELETE errors

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.

```
primary key for row in table 'Departments' is referenced in another
table
```

To remove the R&D department, you need to first get rid of all employees in that department:

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

You can now delete the R&D department.

You should cancel these changes to the database (for future use) by entering a ROLLBACK statement:

```
ROLLBACK;
```

All changes made since the last successful COMMIT WORK are undone. If you have not done a COMMIT, then all changes since you started Interactive SQL are undone.

Example: UPDATE errors

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 after the COMMIT WORK is complete

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 the COMMIT WORK is done. This is important if you want to change the value of a referenced primary key; for example, deleting the R&D department in the Employees and Departments tables. To make these deletions, the database has to be inconsistent in between the changes. In this case, you must configure the database to check only on commits.

☞ For more information, 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.

☞ For more information, see the chapter "Ensuring Data Integrity" on page 89.

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 the "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' );
```

Execute a ROLLBACK statement to undo the insert.

The DepartmentHeadID column has no default, but can allow NULL. A NULL is assigned to that column.

The order in which you list the column names must match the order in which you list the values. The following example produces the same results as the previous one:

```
INSERT INTO Departments ( DepartmentName, DepartmentID )
VALUES ( 'Western Sales', 703 );
```

Execute a ROLLBACK statement to undo the insert.

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:

Value being inserted	Nullable	Not nullable	Nullable, with DEFAULT	Not nullable, with DEFAULT	Not nullable, with DEFAULT AUTOINCREMENT
<none>	NULL	SQL error	DEFAULT value	DEFAULT value	DEFAULT value
NULL	NULL	SQL error	NULL	SQL error	DEFAULT value
specified value	specified value	specified value	specified value	specified value	specified value

By default, columns allow NULL values unless you explicitly state NOT NULL in the column definition when creating a table. You can alter this default using the allow_nulls_by_default option. You can also alter whether a specific column allows NULLs using the ALTER TABLE statement. See "allow_nulls_by_default option [compatibility]" [*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.

☞ For more information on constraints, see "Using table and column constraints" on page 99.

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.

☞ For more information about defaults, see "Using column defaults" on page 93.

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 their CREATE TABLE statements, you do not need to specify column names in either table. Suppose you have a table named NewProducts that contains some rows of product information in the same format as in the Products table. To add to Products all the rows in NewProducts:

```
INSERT Products
SELECT *
FROM NewProducts;
```

You can use expressions in a SELECT statement inside an INSERT statement.

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.

☞ For more information about adding INSERT statements to applications, 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.
For more information, 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

```
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.

☞ For more information, 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;
```

You should roll back your changes when you have completed the example:

```
ROLLBACK;
```

Tip
You can also delete rows from database tables from the Interactive SQL result set.
For more information, see "Editing result sets in Interactive SQL" [*SQL Anywhere Server - Database Administration*].

Deleting all rows from a table

You can use the TRUNCATE TABLE statement as a fast method of deleting all the rows in a table. It is faster than a DELETE statement with no conditions, because the DELETE logs each change, while TRUNCATE does not record individual rows deleted.

The table definition for a table emptied with the TRUNCATE TABLE statement remains in the database, along with its indexes and other associated objects, unless you execute a DROP TABLE statement.

You cannot use TRUNCATE TABLE if another table has rows that reference it through a referential integrity constraint. Delete the rows from the foreign table, or truncate the foreign table and then truncate the primary table.

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.

CHAPTER 14

Query Optimization and Execution

Contents

About this chapter

This chapter describes the steps a statement goes through starting with the annotation phase and ending with its execution. It documents the assumptions that underlie the design of the optimizer, and discusses selectivity estimation, cost estimation, and the other steps of optimization. It also describes different execution algorithms used to compute the result of a query.

Although UPDATE, INSERT, and DELETE statements must also be optimized and executed, the focus of this chapter is on SELECT queries. The processing of these other commands follows similar principles.

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.

Following are the phases of query processing in SQL Anywhere:

- **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.
- **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 on the Plan tab in Interactive SQL (you must have one of the Graphical Plan choices selected in Tools ▶ Options ▶ Plan). 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 execution plans" on page 529.
- **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. See "Queries that bypass optimization" on page 475.

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 "Execution plan caching" on page 505.

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 left-hand condition will be evaluated first. If the truth of the entire condition can then be determined, the compiler guarantees that the right-hand condition will not be evaluated.

This arrangement lets you combine conditions that would otherwise require two nested IF statements into one. For example, in C you can test whether a pointer is NULL before you use it as follows. The nested conditions in the first statement can be replaced using the syntax shown in the second statement below:

```
if ( X != NULL ) {
   if ( X->var != 0 ) {
      ... statements ...
   }
}

if ( X != NULL && X->var != 0 ) {
      ... statements ...
}
```

Unlike C, SQL has no such rules concerning execution order. SQL Anywhere is free to rearrange the order of such conditions as it sees fit. The original and reordered forms are semantically equivalent because the SQL language specification makes no distinction between one order or another. In particular, query optimizers are 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:

- Elimination of unnecessary DISTINCT conditions
- unnesting subqueries

- predicate pushdown in UNION or GROUPed views and derived tables
- optimization of OR and IN-list predicates
- optimization of LIKE predicates
- conversion of outer joins to inner joins
- elimination of outer joins and inner joins
- discovery of exploitable conditions through predicate inference
- elimination of unnecessary case translation
- rewriting subqueries as EXISTS predicates

Elimination of unnecessary DISTINCT conditions

Sometimes a DISTINCT condition is unnecessary. For example, the properties of one or more column in your result may contain a UNIQUE condition, either explicitly or implicitly, because it is a primary key.

Examples

The DISTINCT keyword in the following command is unnecessary because the Products table contains the primary key p.ID, which is part of the result set.

```
SELECT DISTINCT p.ID, p.Quantity
FROM Products p;
```

```
Products<seq>
```

The database server executes the semantically-equivalent query:

```
SELECT p.ID, p.Quantity
FROM Products p;
```

Similarly, the result of the following query contains the primary keys of both tables, so each row in the result must be distinct. 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';
```

```
Work[ HF[ c<seq> ] *JH o<seq> ]
```

Unnesting 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's WHERE clause. In general, subquery unnesting 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 non-correlated subquery, or a subquery with more than one table in the FROM clause, is flattened if it can be decided, based on the query semantics, that the subquery returns at most one row.

Examples

The subquery in the following example can match at most one row for each row in the outer block. Because it can match at most one row, SQL Anywhere recognizes that it can convert it to an inner join.

```
SELECT s.*
FROM SalesOrderItems s
WHERE EXISTS
    ( SELECT *
      FROM Products p
      WHERE s.ProductID = p.ID
         AND p.ID = 300 AND p.Quantity > 20);
```

Following conversion, this same statement is expressed internally using join syntax:

```
SELECT s.*
FROM Products p JOIN SalesOrderItems s
   ON p.ID = s.ProductID
WHERE p.ID = 300 AND p.Quantity > 20;
```

```
p<Products> JNL s<FK_ProductID_ID>
```

Similarly, the following query contains a conjunctive EXISTS predicate in the subquery. This subquery can match more than one row.

```
SELECT p.*
FROM Products p
WHERE EXISTS
    ( SELECT *
      FROM SalesOrderItems s
      WHERE s.ProductID = p.ID
         AND s.ID = 2001);
```

SQL Anywhere converts this query to an inner join, with a DISTINCT in the SELECT-list.

```
SELECT DISTINCT p.*
FROM Products p JOIN SalesOrderItems s
   ON p.ID = s.ProductID
WHERE s.ID = 2001;
```

```
Work[ DistH[ s<FK_ID_ID> JNL p<Products> ] ]
```

SQL Anywhere can also eliminate subqueries in comparisons when the subquery matches at most one row for each row in the outer block. Such is the case in the following query.

```
SELECT *
FROM Products p
WHERE p.ID =
    ( SELECT s.ProductID
      FROM SalesOrderItems s
      WHERE s.ID = 2001
         AND s.LineID = 1 );
```

SQL Anywhere rewrites this query as follows:

```
SELECT p.*
FROM Products p, SalesOrderItems s
```

```
WHERE p.ID = s.ProductID
   AND s.ID = 2001
   AND s.LineID = 1;
```

```
s<SalesOrderItems> JNL p<Products>
```

The DUMMY table is treated as a special table when subquery unnesting 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 pushdown 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 pushdown is performed for a predicate if, and only if, the predicate refers exclusively to the columns of a single view or derived table. A join predicate, for example, is not pushed down into the view.

Example

Suppose you have the view ProductSummary defined as follows:

```
CREATE VIEW ProductSummary( ID,
        NumberOfOrders,
        TotalQuantity) AS
SELECT ProductID, count(*), sum( Quantity )
FROM SalesOrderItems
GROUP BY ProductID;
```

For each product ordered, the ProductSummary view returns a count of the number of orders that include it, and the sum of the quantities ordered over all of the orders. Now consider the following query over this view:

```
SELECT *
FROM ProductSummary
WHERE ID = 300;
```

The query restricts the output to only the row for which the value in the ID column is 300. This query, and the query in the definition of the view could be combined into the following, semantically-equivalent, SELECT statement:

```
SELECT ProductID, COUNT(*), SUM( Quantity )
FROM SalesOrderItems
GROUP BY ProductID
HAVING ProductID = 300;
```

An unsophisticated execution plan for this query would involve computing the aggregates for each product, and then restricting the result to only the single row for product ID 300. However, the HAVING predicate on the ProductID column can be pushed into the query's WHERE clause since it is a grouping column, yielding the following:

```
SELECT ProductID, COUNT(*), SUM( Quantity )
FROM SalesOrderItems
WHERE ProductID = 300
GROUP BY ProductID;
```

This SELECT statement significantly reduces the computation required. If this predicate is sufficiently selective, the optimizer could now use an index on ProductID to retrieve only those rows for product 300, rather than sequentially scanning the SalesOrderItems table.

The same optimization is also used for views involving UNION or UNION ALL.

Optimization of OR and IN-list predicates

The optimizer supports a special optimization for exploiting IN predicates on indexed columns. This optimization also applies equally to multiple predicates on the same indexed column that are ORed together, since the two are semantically equivalent. To enable the optimization, the IN-list must contain only constants.

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:

```
SELECT *
FROM SalesOrders
WHERE SalesRepresentative = 902 OR SalesRepresentative = 195;
```

This query is semantically equivalent to:

```
SELECT *
FROM SalesOrders
WHERE SalesRepresentative IN (195, 902);
```

The optimizer estimates the combined selectivity of the IN-list predicate to be low enough to warrant indexed retrieval. 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:

```
SalesOrders<FK_SalesRepresentative_EmployeeID>
```

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 can be converted to an inner join when a null-intolerant predicate on the null-supplying table is present in the query's 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.

Example

For example, consider the query that is intended to list all products and their orders for larger quantities; the LEFT OUTER JOIN is intended to ensure 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';
```

While it is rare for this optimization to apply to straightforward outer join queries, it can often apply when a query refers to one or more views that are written using outer joins. The query's WHERE clause may include conditions that restrict the output of the view such that all null-supplying rows from one or more table expressions would be eliminated, hence making this optimization applicable.

Elimination of unecessary inner and outer joins

The join elimination rewrite optimization reduces the join degree of the query by eliminating tables from the query when it is safe to do so. Typically, this optimization is applied for inner joins defined as primary key to foreign key joins, or primary key to primary key joins. The join elimination optimization can also be applied to tables used in outer joins, although the conditions for which the optimization is valid are much more complex.

This optimization does not eliminate tables that are updatable using UPDATE or DELETE WHERE CURRENT, even when it is correct to do so. This can negatively impact performance of the query. However, if your query is for read only, you can specify FOR READ ONLY in the SELECT statement, to ensure that the join eliminations are performed. Note that the tables appearing in subqueries or nested derived tables are inherently non-updatable, even though the tables in the main query block are updatable.

To summarize, there are three main categories of joins for which this rewrite optimization applies:

- The join is a primary key to foreign key join, and only primary key columns from the primary table are referenced in the query. In this case, the primary key table is eliminated if it is not updatable.
- The join is a primary key to primary key join between two instances of the same table. In this case, one of the tables is eliminated if it is not updatable.
- The join is an outer join and the null-supplying table expression has the following properties:
  - the null-supplying table expression returns at most one row for each row of the preserved side of the outer join
  - no expression produced by the null-supplying table expression is needed in the rest of the query beyond the outer join.

Example

For example, in the query below, the join is a primary key to foreign key join and the primary key table, Products, can be eliminated:

```
SELECT s.ID, s.LineID, p.ID
FROM SalesOrderItems s KEY JOIN Products p
FOR READ ONLY;
```

The query would be rewritten as:

```
SELECT s.ID, s.LineID, s.ProductID
FROM SalesOrderItems s
WHERE s.ProductID IS NOT NULL
FOR READ ONLY;
```

The second query is semantically equivalent to the first because any row from the SalesOrderItems table that has a NULL foreign key to Products does not appear in the result.

In the following query, the OUTER JOIN can be eliminated given that the null-supplying table expression cannot produce more than one row for any row of the preserved side and none of the columns from Products is used above the LEFT OUTER JOIN.

```
SELECT s.ID, s.LineID
FROM SalesOrderItems s LEFT OUTER JOIN Products p ON p.ID = s.ProductID
WHERE s.Quantity > 5
FOR READ ONLY;
```

The query is rewritten as:

```
SELECT s.ID, s.LineID
FROM SalesOrderItems s
WHERE s.Quantity > 5
FOR READ ONLY;
```

Discovery of exploitable conditions through predicate inference

An efficient access strategy for virtually any query relies on the presence of sargable conditions in the WHERE, ON, and HAVING clauses. Indexed retrieval is possible only by exploiting sargable conditions as matching predicates. In addition, hash, merge, and block-nested loops join can only be used when an equijoin condition is present. For these reasons, SQL Anywhere does detailed analysis of the search conditions in the original query text to discover simplified or implied conditions that can be exploited by the optimizer.

As a preprocessing step, several simplifications are made to predicates in the original statement once view expansion and merging have taken place. For example:

- `X = X` is rewritten as `X IS NOT NULL` if X is nullable; otherwise, the predicate is eliminated.
- `ISNULL(X,X)` is rewritten as 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 ANDed together. Each term is either made up of a single atomic condition, or a set of conditions ORed together.

Converting an arbitrary condition into CNF may yield an expression of similar complexity, but with a much larger set of conditions. SQL Anywhere recognizes this situation, and refrains from naively converting the condition into CNF. Instead, SQL Anywhere analyzes the original expression for exploitable predicates that are implied by the original search condition, and ANDs these inferred conditions to the query. Complete normalization is also avoided if this requires duplication of an expensive predicate (for example, a quantified subquery predicate). However, the algorithm merges IN-list predicates together whenever feasible.

Once the search condition has either been completely normalized or the exploitable conditions have been found, the optimizer performs transitivity analysis to discover transitive equality conditions, primarily transitive join conditions and conditions with a constant. In doing so, the optimizer increases its degrees of freedom when performing join enumeration during its cost-based optimization phase, since these transitive conditions may permit additional alternative join orders.

Example

Suppose the original query is as follows:

```
SELECT e.Surname, s.ID, s.OrderDate
FROM SalesOrders s, Employees e
WHERE
  ( e.EmployeeID = s.SalesRepresentative AND
    ( s.SalesRepresentative = 142 OR
      s.SalesRepresentative = 1596 )
  ) OR (
     e.EmployeeID = s.SalesRepresentative AND
     s.CustomerID = 667 );
```

This query has no conjunctive equijoin condition; 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:

```
SELECT e.Surname, s.ID, s.OrderDate
FROM SalesOrders s, Employees e
WHERE
   e.EmployeeID = s.SalesRepresentative AND
   ( s.SalesRepresentative = 142 OR
     s.SalesRepresentative = 1596 OR
     s.CustomerID = 667 );
```

This query can now be efficiently optimized as an inner join query.

Elimination of unnecessary case translation

By default, SQL Anywhere databases support case-insensitive string comparisons. Occasionally the optimizer may encounter queries where the user is explicitly forcing text conversion through the use of the UPPER, UCASE, LOWER, or LCASE built-in functions when such conversion is unnecessary. SQL Anywhere automatically eliminates this unnecessary conversion when the database's collation sequence permits it. An extra benefit of eliminating the case translations in the predicates is the transformation of some of these predicates into sargable predicates, which can be used for indexed retrieval of the corresponding table.

Example

Consider the following query:

```
SELECT *
FROM Customers
WHERE UPPER(Surname) = 'SMITH';
```

On a case insensitive database, this query is rewritten internally as follows, so that the optimizer can consider using an index on Customers.Surname:

```
SELECT *
FROM Customers
WHERE Surname = 'SMITH';
```

Rewriting subqueries as EXISTS predicates

The assumptions that underlie the design of SQL Anywhere require that it conserves memory and that by default it returns the first few results of a cursor as quickly as possible. In keeping with these objectives, SQL Anywhere rewrites all set-operation subqueries, such as IN, ANY, or SOME predicates, as EXISTS or NOT EXISTS predicates, if such rewriting is semantically correct. By doing so, SQL Anywhere avoids creating unnecessary work tables and may more easily identify a suitable index through which to access a table.

Non-correlated and correlated subqueries

Non-correlated 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 non-correlated subquery. It selects information about all the customers who did not place an order on January 1, 2001.

```
SELECT *
FROM Customers c
WHERE c.ID NOT IN
   (  SELECT o.CustomerID
      FROM SalesOrders o
      WHERE o.OrderDate = '2001-01-01' );
```

One possible way to evaluate this query is to create a work table of all customers in the SalesOrder table who placed orders on January 1, 2001, and then query the Customers table and extract one row for each customer listed in the work table.

However, SQL Anywhere avoids materializing results as work tables. It also gives preference to plans that return the first few rows of a result most quickly. 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.

```
SELECT *
FROM Customers c
WHERE NOT EXISTS
   (  SELECT *
      FROM SalesOrders o
      WHERE o.OrderDate = '2000-01-01'
         AND o.CustomerID = c.ID );
```

This query is semantically equivalent to the one above, but when expressed in this new syntax, several advantages become 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 on subquery caching is located in “Subquery and function caching” on page 524.

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 dialog (available from the Tools menu). See "Graphical plans" on page 223, and "Reading execution plans" on page 529.

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 489.

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 ProcCall algorithm. See "SYSPROCEDURE system view" [*SQL Anywhere Server - SQL Reference*], and "ProcCall algorithm" on page 527.

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 the following:

- "sa_get_histogram system procedure" [*SQL Anywhere Server - SQL Reference*]
- "The Histogram utility" [*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.

```
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 489.

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 489, and “Working with indexes” on page 80.

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 489.

Indexes can be used to satisfy a predicate

Often, SQL Anywhere can evaluate sargable predicates with the aid of an index. Using an index, the optimizer speeds access to data and reduces the amount of information read and processed from base tables. For example, when optimization_goal is set to first-row, the optimizer tries to use indexes to satisfy ORDER BY and GROUP BY clauses. See "optimization_goal option [database]" [*SQL Anywhere Server - Database Administration*].

When the optimizer cannot find a suitable index, it resorts to a sequential table scan, which can be expensive. An index can improve performance dramatically when joining tables. Add indexes to tables or rewrite queries wherever doing so facilitates the efficient processing of common requests.

To find out if your query performance can be improved by creating new indexes, use the Application Profiling wizard found in Application Profiling mode in Sybase Central.

☞ For more information on profiling and predicate analysis, see "Application profiling" on page 188, and "Predicate analysis" on page 492.

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.

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 52.

Examples

In each of these examples, attributes *x* and *y* are each columns of a single table. Attribute *z* is contained in a separate table. Assume that an index exists for each of these attributes.

Sargable	**Non-sargable**
x = 10	*x* < > 10
x IS NULL	*x* IS NOT NULL
x > 25	*x* = 4 OR *y* = 5
x = *z*	*x* = *y*
x IN (4, 5, 6)	*x* NOT IN (4, 5, 6)
x LIKE 'pat%'	*x* LIKE '%tern'
x = 20 - 2	*x* + 2 = 20

Sometimes it may not be obvious whether a predicate is sargable. In these cases, you may be able to rewrite the predicate so it is sargable. For each example, you could rewrite the predicate *x* LIKE 'pat%' using the fact that u is the next letter in the alphabet after t: *x* >= 'pat' and *x* < 'pau'. In this form, an index on attribute *x* is helpful in locating values in the restricted range. Fortunately, SQL Anywhere makes this particular transformation for you automatically.

A sargable predicate used for indexed retrieval on a table is a **matching** predicate. A WHERE clause can have 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 483.

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 74.
- the materialized view is enabled for use in optimization. See “Enabling and disabling optimizer use of a materialized view” on page 76.
- the materialized view has been initialized. See “Initializing materialized views” on page 72.
- the materialized view meets all of the optimizer requirements for consideration. See “Requirements for View Matching algorithm” on page 496.
- the values of some critical options used to create the materialized views match the options for the connection executing the query. See “Restrictions when managing materialized views” on page 69.
- 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 77.

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 when managing materialized views" on page 69.

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 529.

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 tracing level" on page 202.

☞ For more information on the enumeration phase of optimization, see "Phases of query processing" on page 474.

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
- LEFT OUTER JOIN
- RIGHT OUTER 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 529.
- "Phases of query processing" on page 474
- "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 497).

  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 "Execution plan caching" on page 505.

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 502.

- ♦ 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 when managing materialized views" on page 69.
- ♦ 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 77.
- ♦ All the tables used in V, with possible exceptions of some extension tables of V, are present in the QB. This set of common tables in the QB is hereinafter referred to as CT.
- ♦ No table in CT is updatable in the query Q.
- ♦ All tables in CT belong to the same side of an outer join in QB (that is, they are all in the preserved side of the outer join or all in the null-supplying side of an outer join of QB).
- ♦ It can be decided that the predicates in V subsume the subset of the predicates in QB that reference CT only. In other words, the predicates in V are less restrictive than those in QB. A predicate in QB that exactly matches one in V is called a **matched predicate**.
- ♦ Any expression of QB referencing tables in CT that is not used in a matched predicate must appear in the select list of V.

- If both V and QB are grouped, then QB doesn't contain extra tables besides the ones in CT. Additionally, the set of expressions in the GROUP BY clause of V must be equal to or a superset of the set of expressions in the GROUP BY clause of QB.
- If both V and QB are grouped on an identical set of expressions, all aggregate functions in QB must be also computed in V, or it is possible to compute them from V's aggregate functions. For example, if QB contains AVG(x) then V must contain AVG(x), or it must contain both SUM(x) and COUNT(x).
- If QB's GROUP BY clause is a subset of V's GROUP BY clause, then the simple aggregate functions of QB must be found among V's aggregate functions, while its composite aggregate functions have to be computed from simple aggregate functions of V. The simple aggregate functions are:

  - BIT_AND
  - BIT_OR
  - BIT_XOR
  - COUNT
  - LIST
  - MAX
  - MIN
  - SET_BITS
  - SUM
  - XMLAGG

  The composite aggregate functions that can be computed from the simple aggregate functions:

  - SUM(x)
  - COUNT(x)
  - SUM(CAST(x AS DOUBLE))
  - SUM(CAST(x AS DOUBLE) * CAST(x AS DOUBLE))

  are:

  - VAR_SAMP(x)
  - VAR_POP(x)
  - VARIANCE(x)
  - STDDEV_SAMP(x)
  - STDDEV_POP(x)
  - STDDEV(x)

  The statistical aggregate functions:

  - COVER_SAMP(y,x)
  - COVER_POP(y,x)
  - CORR(y,x)
  - REGR_AVGX(y,x)
  - REGR_AVGY(y,x)
  - REGR_SLOPE(y,x)
  - REGR_INTERCEPT(y,x)
  - REGR_R2(y,x)

- REGR_COUNT(y,x)
- REGR_SXX(y,x)
- REGR_SYY(y,x)
- REGR_SXY(y,x)

can be computed from the simple aggregate functions:

- SUM(y1)
- SUM(x1)
- COUNT(x1)
- COUNT(y1)
- SUM(x1*y1)
- SUM(y1*x1)
- SUM(x1*x1)
- SUM(y1*y1)

where x1 = CAST(IFNULL(x, x,y) AS DOUBLE)) and y1 = CAST(IFNULL(y,y,x) AS DOUBLE).

Example1: Matching select-project-join views

If a certain partition of a base table is frequently accessed by queries, then it may be beneficial to define a materialized view to store that partition. For example, the materialized view V_Canada defined below stores all the customers from the Customer table who live in Canada. As this materialized view is used when the State column is restricted to certain values, it is advisable to create the index V_Canada_State on the State column of the V_Canada materialized view.

```
CREATE MATERIALIZED VIEW V_Canada AS
 SELECT c.ID, c.City, c.State, c.CompanyName
  FROM Customers c
  WHERE c.State IN ( 'AB', 'BC', 'MB', 'NB', 'NL',
   'NT', 'NS', 'NU', 'ON', 'PE', 'QC', 'SK', 'YT' );
REFRESH MATERIALIZED VIEW V_Canada;
CREATE INDEX V_Canada_State on V_Canada( State );
```

Any query block that requires just a subset of customers living in Canada may benefit from this materialized view. For example, Query 1 below, which computes the total price of the products for all customers in Ontario for each company, may use the V_Canada materialized view in its access plans. An access plan for Query 1 using the V_Canada materialized view represents a valid plan as if Query 1 was rewritten as Query 1_v, which is semantically equivalent to it. Note that the optimizer doesn't rewrite the query using the materialized views, instead the generated access plans using materialized views can theoretically be seen corresponding to the rewritten query.

The execution plan of the Query 1 uses the V_Canada materialized view, as shown here: `Work[ GrByH [ V_Canada<V_Canada_State> JNLO SalesOrders<FK_CustomerID_ID> JNLO SalesOrderItems<FK_ID_ID> JNLO Products<ProductsKey> ] ]`

Query 1:

```
SELECT SUM( SalesOrderItems.Quantity
   * Products.UnitPrice ) AS Value
 FROM Customers c
  LEFT OUTER JOIN SalesOrders
   ON( SalesOrders.CustomerID = c.ID )
  LEFT OUTER JOIN SalesOrderItems
```

```
     ON( SalesOrderItems.ID = SalesOrders.ID )
    LEFT OUTER JOIN Products
     ON( Products.ID = SalesOrderItems.ProductID )
   WHERE c.State = 'ON'
   GROUP BY c.CompanyName;
```

Query 1_v:

```
SELECT SUM( SalesOrderItems.Quantity
   * Products.UnitPrice ) AS Value
  FROM V_Canada
   LEFT OUTER JOIN SalesOrders ON( SalesOrders.CustomerID = V_Canada.ID )
   LEFT OUTER JOIN SalesOrderItems ON( SalesOrderItems.ID = SalesOrders.ID )
   LEFT OUTER JOIN Products ON( Products.ID = SalesOrderItems.ProductID )
  WHERE V_Canada.State = 'ON'
  GROUP BY V_Canada.CompanyName;
```

Query 2 may use this view in both the main query block and the HAVING subquery. Some of the access plans enumerated by the optimizer using the V_Canada materialized view represent Query 2_v, which is semantically equivalent to Query 2 where the Customer table was replaced by the V_Canada view .

The execution plan is: `Work[ GrByH[ V_Canada<V_Canada_State> JNLO SalesOrders<FK_CustomerID_ID> JNLO SalesOrderItems<FK_ID_ID> JNLO Products<ProductsKey> ] ] : GrByS[ V_Canada<seq> JNLO SalesOrders<FK_CustomerID_ID> JNLO SalesOrderItems<FK_ID_ID> JNLO Products<ProductsKey>`

Query 2:

```
SELECT  SUM( SalesOrderItems.Quantity
  * Products.UnitPrice ) AS Value
 FROM Customers c
  LEFT OUTER JOIN SalesOrders
   ON( SalesOrders.CustomerID = c.ID )
  LEFT OUTER JOIN SalesOrderItems
   ON( SalesOrderItems.ID = SalesOrders.ID )
  LEFT OUTER JOIN Products
   ON( Products.ID = SalesOrderItems.ProductID )
 WHERE c.State = 'ON'
 GROUP BY CompanyName
 HAVING Value >
  ( SELECT AVG( SalesOrderItems.Quantity
        * Products.UnitPrice ) AS Value
      FROM Customers c1
       LEFT OUTER JOIN SalesOrders
       ON( SalesOrders.CustomerID = c1.ID )
       LEFT OUTER JOIN SalesOrderItems
       ON( SalesOrderItems.ID = SalesOrders.ID )
        LEFT OUTER JOIN Products
       ON( Products.ID = SalesOrderItems.ProductID )
       WHERE c1.State IN ('AB', 'BC', 'MB', 'NB', 'NL', 'NT', 'NS',
      'NU', 'ON', 'PE', 'QC', 'SK', 'YT' ) );
```

Query 2_v:

```
SELECT  SUM( SalesOrderItems.Quantity
         * Products.UnitPrice ) AS Value
 FROM V_Canada
 LEFT OUTER JOIN SalesOrders
  ON( SalesOrders.CustomerID=V_Canada.ID )
 LEFT OUTER JOIN SalesOrderItems
```

```
    ON( SalesOrderItems.ID=SalesOrders.ID )
   LEFT OUTER JOIN Products
    ON( Products.ID=SalesOrderItems.ProductID )
  WHERE V_Canada.State = 'ON'
  GROUP BY V_Canada.CompanyName
  HAVING Value >
     ( SELECT AVG( SalesOrderItems.Quantity
           * Products.UnitPrice ) AS Value
         FROM V_Canada
          LEFT OUTER JOIN SalesOrders
          ON( SalesOrders.CustomerID = V_Canada.ID )
          LEFT OUTER JOIN SalesOrderItems
          ON( SalesOrderItems.ID = SalesOrders.ID )
           LEFT OUTER JOIN Products
          ON( Products.ID = SalesOrderItems.ProductID )
          WHERE V_Canada.State IN ('AB', 'BC', 'MB',
         'NB', 'NL', 'NT', 'NS', 'NU', 'ON', 'PE', 'QC',
         'SK', 'YT' ) );
```

Example 2: Matching grouped-select-project-join views

The grouped materialized views have the potential for the highest performance impact on the grouped queries. If similar aggregations are used in frequently-executed queries, a materialized view should be defined to pre-aggregate data on a superset of the group by clauses used in those queries. Composite aggregate functions of the queries can be computed from the simple aggregates used in the views, hence it is advisable to store only simple aggregate functions in the materialized views.

The materialized view 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 `Work[ GrByH[ V_quantity<seq> ] ]`, corresponding to Query 3_v).

Query 4, which doesn't reference the extension table SalesOrders, can still use V_quantity as the view contains all the data necessary to compute Query 4 (the short plan is `Work[ GrByH [ V_quantity<seq> ] ]`, corresponding to Query 4_v).

```
  CREATE MATERIALIZED VIEW V_Quantity AS
   SELECT s.ProductID,
    Month( o.OrderDate ) AS month,
    Year( o.OrderDate ) AS year,
    SUM( s.Quantity ) AS q_sum,
    COUNT( s.Quantity  ) AS q_count
   FROM SalesOrderItems s KEY JOIN SalesOrders o
   GROUP BY s.ProductID, Month( o.OrderDate ),
    Year( o.OrderDate );
  REFRESH MATERIALIZED VIEW V_Quantity;
```

Query 3:

```
  SELECT s.ProductID,
    Month( o.OrderDate ) AS month,
    AVG( s.Quantity) AS avg,
    SUM( s.Quantity ) AS q_sum,
    COUNT( s.Quantity ) AS q_count
   FROM SalesOrderItems s KEY JOIN SalesOrders o
   WHERE year( o.OrderDate ) = 2000
   GROUP BY s.ProductID, Month( o.OrderDate );
```

Query 3_v:

```
  SELECT V_Quantity.ProductID,
    V_Quantity.month AS month,
    SUM( V_Quantity.q_sum ) / SUM( V_Quantity.q_count )
     AS avg,
    SUM( V_Quantity.q_sum ) AS q_sum,
    SUM( V_Quantity.q_count ) AS q_count
   FROM V_Quantity
   WHERE V_Quantity.year  = 2000
   GROUP BY V_Quantity.ProductID, V_Quantity.month;
```

Query 4:

```
  SELECT s.ProductID,
    AVG( s.Quantity ) AS avg,
    SUM( s.Quantity ) AS sum
   FROM SalesOrderItems s
   WHERE s.ProductID IS NOT NULL
   GROUP BY s.ProductID;
```

Query 4_v

```
  SELECT V_Quantity.ProductID,
    SUM( V_Quantity.q_sum ) / SUM( V_Quantity.q_count )
     AS avg,
    SUM( V_Quantity.q_sum ) AS sum
   FROM V_Quantity
   WHERE V_Quantity.ProductID IS NOT NULL
   GROUP BY V_Quantity.ProductID;
```

Example 3: Matching complex queries

The View Matching algorithm is applied per query block, so it is possible to use more than one materialized view per query block and also more than one materialized view for the whole query. Query 5 below may use the three materialized views: V_Canada for one of the null-supplying sides of the LEFT OUTER JOIN; V_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: `Work[ Window[ Sort [ V_ship_date<V_Ship_date_date> JNLO ( so<SalesOrdersKey> JH V_Canada<V_Canada_state> ) ] ] ] : GrByS[V_quantity<seq> ]`.

```
  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';
  REFRESH MATERIALIZED VIEW V_ship_date;
  CREATE INDEX V_ship_date_date ON V_ship_date( ShipDate );
```

Query 5:

```
  SELECT p.ID, p.Description, s.Quantity,
    s.ShipDate, so.CustomerID, c.CompanyName,
    SUM( s.Quantity ) OVER ( PARTITION BY s.ProductID
                          ORDER BY s.ShipDate
                          ROWS BETWEEN UNBOUNDED PRECEDING
                          AND CURRENT ROW ) AS cumulative_qty
   FROM SalesOrderItems s JOIN Products p
    ON (s.ProductID = p.ID) LEFT OUTER JOIN (
     SalesOrders so JOIN Customers c
      ON ( c.ID = so.CustomerID AND c.State = 'ON' ) )
    ON (s.ID = so.ID )
   WHERE s.ShipDate >= '2000-01-01'
     AND s.ShipDate <= '2000-12-31'
```

```
      AND s.Quantity  > ( SELECT  AVG( s.Quantity ) AS avg
                          FROM SalesOrderItems s KEY JOIN SalesOrders o
                          WHERE year( o.OrderDate) = 2000 )
    FOR READ ONLY;
```

Query 5_v:

```
  SELECT  V_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;
```

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

```
  SELECT MIN( Quantity )
   FROM SalesOrderItems
   WHERE ShipDate = ‘2000-03-25’;
```

is rewritten internally as

```
  SELECT MAX( Quantity )
   FROM ( SELECT FIRST Quantity
          FROM SalesOrderItems
          WHERE ShipDate = ‘2000-03-25’
              AND Quantity IS NOT NULL
   ORDER BY ShipDate ASC, Quantity ASC ) AS s(Quantity);
```

The NULL_VALUE_ELIMINATED warning may not be generated for aggregate queries when this optimization is applied.

The execution plan (short form) for the rewritten query is:

```
GrByS[ RL[ SalesOrderItems<prod_qty> ] ]
```

Execution 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. For queries and INSERT, UPDATE, and DELETE statements performed inside stored procedures, user-defined functions, and triggers, the optimizer caches execution plans between executions of the query.

After a statement in a stored procedure, user-defined function, or trigger has been executed several times by one 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 will choose 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.

You can use the database or connection property QueryCachePages 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.

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

The maximum number of plans to cache is specified with the option setting max_plans_cached. The default is 20. To disable plan caching, set this option to 0.

See also

- “Improving performance with materialized views” on page 494
- “materialized_view_optimization option [database]” [*SQL Anywhere Server - Database Administration*]
- “DB_PROPERTY function [System]” [*SQL Anywhere Server - SQL Reference*]
- “CONNECTION_PROPERTY function [System]” [*SQL Anywhere Server - SQL Reference*]
- “max_plans_cached option [database]” [*SQL Anywhere Server - Database Administration*]

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 529, and "Graphical plans" on page 539.

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, and the switch is 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 on the QueryLowMemoryStrategy property, see "Connection-level properties" [*SQL Anywhere Server - Database Administration*].

  Memory governor limits are dependent upon the server's multiprogramming level 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*].

- 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 on 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" on page 525). 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]" [*SQL Anywhere Server - Database Administration*]).

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 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 antisemijoin
  - ♦ parallel hash filter
  - ♦ parallel hash Group By
- ♦ the number of disk devices used for the storage of the database, and their relative speed to that of the processor and the computer's I/O architecture.

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 within 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” on page 525
- “Reading execution plans” on page 529
- “ALTER DATABASE statement” [*SQL Anywhere Server - SQL Reference*]

Table access algorithms

This section explains the various methods used to access tables, with table and index scans being the most common methods.

Hash Table Scans

The Hash Table Scan is an operator that lets SQL Anywhere scan the build side of a hash join as if it were an in-memory table. This lets you convert a plan with this structure:

```
table1<seq>*JH ( arbitrary dfos... ( table2<seq> ) )
```

to this structure:

```
table1<seq>*JF ( arbitrary dfos... ( HTS JNB table2<idx> ) )
```

where `idx` is an index that you can use to probe the join key values stored in the hash table.

When there are intervening operators between the hash join and the scan, it 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, beyond which 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.

Index scans

An index scan uses an index to determine which rows satisfy a search condition. It reads only those pages that satisfy the condition. Indexes can return rows in sorted order.

Index scans appear in the short and long 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, 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 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.

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 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.

☞ For more information about when SQL Anywhere can make use of indexes, see "Predicate analysis" on page 492.

☞ For more information about optimization goals, see "optimization_goal option [database]" [*SQL Anywhere Server - Database Administration*].

Parallel index scans

Parallel Index Scan operators work together under an exchange operator to do an index scan in parallel. As each Parallel Index Scan 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 operators to achieve parallel processing. Regardless of how the pages are distributed among the Parallel Index Scan operators, all of the rows are visited.

Parallel table scans

Parallel Table Scan operators work together under an exchange operator to do a sequential table scan in parallel. As each Parallel Table Scan 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 operators to achieve parallel processing. Regardless of how the pages are distributed among the parallel table scan operators, all of the rows in the table are visited.

ROWID Scan algorithm

The ROWID Scan algorithm is used to locate efficiently 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 plan as *<correlation_name><rowID>*, where *correlation_name* is the correlation name specified in the FROM clause, or the table name if no correlation name was specified.

It is impossible for the ROWID Scan algorithm 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, the algorithm 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*]

Sequential table scans

A sequential table scan reads all the rows in all the pages of a table in the order in which they are stored in the database.

Sequential table scans appear in the short and long 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.

This type of scan is 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 minimizes overhead due to 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 that evaluating the conditions in a filter after the sequential scan.

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 (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 to be 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-hand 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 antisemijoin. 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 algorithm 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.

Hash Antisemijoin algorithm

The Hash Antisemijoin variant of the Hash Join algorithm performs an antisemijoin 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 Hash Antisemijoin, 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 row is output only if it fails to match any row from the right-hand side. Hash Antisemijoin 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. Hash antisemijoin 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 Hash Join, the Hash Antisemijoin algorithm may revert to a nested loops strategy if there is insufficient cache memory to enable 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.

Memory governor limits are dependent upon the server's multiprogramming level 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 CE.

☞ For more information, see QueryLowMemoryStrategy in "Connection-level properties" [*SQL Anywhere Server - Database Administration*].

Hash Join algorithm

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, which are written to a work table. If the smaller input does not fit into memory, the hash join operator partitions both inputs into smaller work tables. These smaller work tables are processed recursively until the smaller input fits into memory.

The Hash Join algorithm 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 the Hash Join algorithm 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, the Hash Join method discards the interim results and an indexed-based Nested Loops Join 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.

Memory governor limits are dependent upon the server's multiprogramming level 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.

The Hash Join algorithm is disabled on Windows CE in low memory conditions.

Note
The Windows Performance Monitor may not be available on Windows CE.

☞ 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*].

The Hash Join algorithm 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

Hash Semijoin algorithm

The Hash Semijoin variant of the Hash Join algorithm performs a semijoin between the left-hand side and the right-hand side. As with Nested Loops Semijoin described above, the right-hand side is only used to determine which rows from the left-hand side appear in the result. With Hash Semijoin 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. As soon as any match is found, the left-hand row is output to the result and the match process starts again for the next left-hand row. At least one equality join condition must be present for Hash Semijoin to be considered by the query optimizer. As with Nested Loops Semijoin, Hash Semijoin 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. Hash Semijoin tends to outperform Nested Loops Semijoin 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 Hash Join, the Hash Semijoin algorithm may revert to a nested loops semijoin strategy if there is insufficient cache memory to enable 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.

Memory governor limits are dependent upon the server's multiprogramming level 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 CE.

☞ For more information, see QueryLowMemoryStrategy in "Connection-level properties" [*SQL Anywhere Server - Database Administration*].

Merge Join algorithm

The Merge Join algorithm 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 a Merge Join compared to a Hash Join 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 Merge Join over a Hash Join if the sizes of the inputs are likely to be similar, or if it can amortize the cost of the sort over several operations.

Nested Loops Join algorithm

The Nested Loops Join computes the join of its left and right 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 a Nested Loops Join 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 a Nested Loops Join 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. On the other hand, 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, a Nested Loops Join is less efficient than other join methods. However, Nested Loops Join can provide the first matching row quickly compared to join methods that must compute their entire result before returning.

The Nested Loops Join algorithm 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 a Nested Loops Join.

A Semijoin fetches only the first matching row from the right-hand side. It is a more efficient version of the Nested Loops Join, but can only be used when an EXISTS, or sometimes a DISTINCT, keyword is used.

See also

- "optimization_goal option [database]" [*SQL Anywhere Server - Database Administration*]
- "FROM clause" [*SQL Anywhere Server - SQL Reference*]

Nested Loops Semijoin algorithm

Similar to the Nested Loops Join described above, the Nested Loops Semijoin algorithm joins each row of the left-hand side with the right-hand side using a Nested Loops algorithm. As with Nested Loops Join, the right-hand side may be read many times, so for larger inputs an index scan is preferable. However, Nested Loops Semijoin differs from Nested Loops Join in two respects. First, Semijoin 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, the Nested Loops Semijoin algorithm stops each search of the right-hand side as soon as the first match is encountered. Nested Loops Semijoin can be used as the Join algorithm when join's inputs include table expressions from an existentially-quantified (IN, SOME, ANY, EXISTS) nested query that has been rewritten as a join.

Recursive Hash Join algorithm

The Recursive Hash Join is a variant of the Hash Join algorithm that is used in recursive union queries.

☞ For more information, see "Hash Join algorithm" on page 514, and "Recursive common table expressions" on page 372.

Recursive Left Outer Hash Join algorithm

The Recursive Left Outer Hash Join is a variant of the Hash Join algorithm used in certain recursive union queries.

☞ For more information, see "Hash Join algorithm" on page 514, and "Recursive common table expressions" on page 372.

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 "Hash Distinct algorithm" on page 517, and "Ordered Distinct algorithm" on page 518.

Hash Distinct algorithm

The Hash Distinct algorithm reads its input, and builds 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.

The Hash Distinct algorithm:

♦ 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 the Hash Distinct algorithm executes in an environment where there is very little cache memory available, then it is not able to complete. In this case, the Hash Distinct method discards its interim results, and the Indexed Distinct algorithm is used instead.

Memory governor limits are dependent upon the server's multiprogramming level 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*].

The Indexed Distinct algorithm maintains a work table of the unique rows from the input. As rows are read from the input, an index on the work table is searched to find a previously seen duplicate of the input row. If one is found, the input row is ignored. Otherwise, the input row is inserted into the work table. The work table index is created on all the columns of the SELECT-list; in order to improve index performance, a hash expression is included as the first expression. This hash expression is a computed value embodying the values of all the columns in the SELECT-list.

The Indexed Distinct method returns distinct rows as they are encountered. This allows it to return the first few rows quickly compared to other duplicate elimination methods. The Indexed Distinct algorithm only stores two rows in memory at a time, and can work well in extremely low memory situations. However, if the number of distinct rows is large, the execution cost of the Indexed Distinct algorithm is typically worse than Hash Distinct. The work table used to store distinct rows may not fit in cache, leading to rereading work table pages many times in a random access pattern.

Since the Indexed Distinct method uses a work table, it cannot provide fully-sensitive semantics; however, it also does not provide fully-insensitive semantics, and another work table is required for insensitive cursors. The Indexed Distinct method locks the rows of its input.

When the Indexed Distinct algorithm is needed due to low memory, a performance counter is incremented. You can see this using 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. See "Performance Monitor statistics" on page 228.

Ordered Distinct algorithm

If the input is ordered by all the columns, then the Ordered Distinct algorithm 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. The Ordered Distinct algorithm 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 Ordered Distinct 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 "Hash Group By algorithm" on page 519, "Ordered Group By algorithm" on page 520, and "Single Group By algorithm" on page 520.

Clustered Hash Group By algorithm

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 are relatively close within the table. The Clustered Hash Group By algorithm exploits this clustering.

The optimizer may use Clustered Hash Group By 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.

The Clustered Hash Group By algorithm 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. Clustered Hash Group By 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 the Clustered Hash Group By algorithm 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 Clustered Group By Hash 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, see "OLAP Support" on page 383.

Hash Group By algorithm

The Hash Group By algorithm creates an in-memory hash table of group rows. As rows are read from the input, the group rows are updated.

The Hash Group By algorithm 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.

The Hash Group By algorithm can be executed in parallel.

The Hash Group By algorithm 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 the Hash Group By algorithm 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 Hash Group By, and uses the Indexed Group By algorithm instead.

Memory governor limits are dependent upon the server's multiprogramming level 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*].

Indexed Group By algorithm

The Indexed Group By algorithm builds a work table containing one row per group. As input rows are read, the associated group is looked up in the work table using an index. 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.

When the Indexed Group By algorithm is needed due to low memory, a performance counter is incremented. You can see this using 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. See “Performance Monitor statistics” on page 228.

Hash Group By Sets algorithm

A variant of the Hash Group By algorithm used when performing GROUPING SETS queries.

The Hash Group By Sets algorithm can not be executed in parallel.

☞ For more information, see “Hash Group By algorithm” on page 519.

Ordered Group By algorithm

The Ordered Group By algorithm 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.

Ordered Group By Sets algorithm

A variant of the Ordered Group By algorithm used when performing GROUPING SETS queries. This algorithm requires that indexes be available.

☞ For more information, see “Ordered Group By algorithm” on page 520.

Single Group By algorithm

When no GROUP BY is specified, a single row aggregate is produced.

Single XML Group By algorithm

An algorithm used when processing XML-based queries.

Sorted Group By Sets algorithm

An algorithm used when processing OLAP queries that contain GROUPING SETS.

This algorithm may be used when it is not possible to use the Ordered Group By Sets algorithm because of lack of an index.

Query expression algorithms

The query expression algorithms can be broken into the following categories:

- Except algorithms, which include Except Merge and Except Hash
- Intersect algorithms, which include Intersect Merge and Intersect Hash
- Union algorithms, which include Union, Union All, and Recursive Union

Except algorithms

The SQL Anywhere query optimizer chooses between two physical implementations of the set difference SQL operator EXCEPT: a sort-based variant, **Except Merge**, and a hash-based variant, **Except Hash**.

Except Merge uses the Merge Join operator 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, Except Hash uses the Hash Anti-Semijoin algorithm to compute the set difference between the two inputs, and Hash Left-Outer join to compute the difference of the two inputs (EXCEPT ALL).

The Except Hash operator 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.

The Except Hash algorithm is disabled on Windows CE in low memory situations.

In the case of EXCEPT, the Except Merge or Except Hash algorithm is coupled with one of the DISTINCT algorithms to ensure that the result does not contain duplicates. For EXCEPT ALL, the Except algorithm is coupled with a RowReplicate algorithm that computes the correct number of duplicate rows in the result.

See also

- "EXCEPT operation" [*SQL Anywhere Server - SQL Reference*]
- "Performing set operations on query results with UNION, INTERSECT, and EXCEPT" on page 312
- 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 228

Intersect algorithms

The SQL Anywhere query optimizer chooses between two physical implementations of the set intersection SQL operator INTERSECT: a sort-based variant, **Intersect Merge**, and a hash-based variant, **Intersect Hash**.

Intersect Merge uses the Merge Join operator 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, Intersect Hash uses the Hash Inner Join algorithm to compute the set and bag intersection between the two inputs (INTERSECT and INTERSECT ALL).

If necessary, the Intersect Hash operator 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.

The Intersect Hash algorithm is disabled on Windows CE in low memory situations.

In the case of INTERSECT, the Intersect Merge or Intersect Hash algorithm is coupled with one of the DISTINCT algorithms to ensure that the result does not contain duplicates. For the INTERSECT ALL query expression, the Intersect algorithm is coupled with a RowReplicate algorithm that computes the correct number of duplicate rows in the result.

See also

- "INTERSECT operation" [*SQL Anywhere Server - SQL Reference*]
- "Performing set operations on query results with UNION, INTERSECT, and EXCEPT" on page 312
- 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 228

Recursive Table algorithm

A recursive table is a common table expression constructed as a result of a WITH clause in a query. 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.

☞ For more information, see "Common Table Expressions" on page 365.

Recursive Union algorithm

The Recursive Union algorithm is employed during the execution of recursive union queries.

☞ For more information, see "Recursive common table expressions" on page 372.

RowReplicate algorithm

The RowReplicate algorithm 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. The RowReplicate algorithm ensures that the number of rows in the result set is correct.

☞ For more information, see “Performing set operations on query results with UNION, INTERSECT, and EXCEPT” on page 312.

Union All algorithm

The Union All algorithm 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 is needed to remove any duplicates generated by the Union All.

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” on page 523 and “Union All algorithm” on page 523.

Sort algorithm

The Sort operator reads its input into memory, sorts it in memory, and then outputs the sorted 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 the Sort algorithm executes in an environment where there is very little cache memory available, it may not be able to complete. In this case, the Sort algorithm 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 than other join methods. The optimizer avoids generating access plans using a Sort algorithm 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.

Memory governor limits are dependent upon the server's multiprogramming level 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 the Sort does not write output rows to a work table, the results of the Sort must be materialized before rows are returned to the application. If necessary, the optimizer adds a work table to ensure this.

Sort Top N algorithm

The Sort Top N algorithm 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.

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

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 less times than would otherwise be expected.

For a functions 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.

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.

User-defined functions are treated as deterministic unless they are specified as NOT DETERMINISTIC when created.

☞ 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.

Derived Table algorithm

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 on derived tables, see “The FROM clause: specifying tables” on page 275 and “FROM clause” [*SQL Anywhere Server - SQL Reference*].

Example

The following query has derived tables in its graphical plan:

```
SELECT EmployeeID FROM Employees
UNION ALL
SELECT DepartmentID FROM (
    SELECT TOP 5 DepartmentID
    FROM Departments
    ORDER BY DepartmentName DESC ) MyDerivedTable;
```

Exchange algorithm

The Exchange algorithm 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.

The Exchange algorithm is only used when processing SELECT statements, and when intra-query parallelism is enabled.

☞ For more information on parallelism, see "Threading in SQL Anywhere" [*SQL Anywhere Server - Database Administration*].

Filter and Pre-filter algorithms

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 a Filter algorithm in the access plan. For example, various algorithms, such as an Index Scan, have the ability to enforce the application of a predicate without the need for an explicit operator. As a concrete example, consider a BETWEEN predicate involving two literal constants, and the column referenced in predicate is indexed. In this case, the BETWEEN predicate can be enforced by the lower and upper bounds of the index scan (sometimes referred to as fenceposts), and the plan for the query will not contain an explicit Filter algorithm. Predicates that are join conditions also do not normally appear in an access plan as a filter.

A Pre-filter algorithm is the same as a Filter algorithm, except that the expressions used in the predicates of a Pre-filter 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 276
- "EXISTS search condition" [*SQL Anywhere Server - SQL Reference*]

Hash Filter algorithm

A Hash Filter or Hash Map, 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> *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, and 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 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.

IN List

The IN List algorithm is used in cases where an IN predicate can be satisfied using an index. For example, in the following query, the optimizer recognizes that it can access the Employees table using its primary key index.

```
SELECT *
FROM Employees
WHERE EmployeeID IN ( 102, 105, 129 );
```

To accomplish this, a join is built with a special IN list table on the left-hand side. Rows are fetched from the IN list and used to probe the Employees table. Multiple IN lists can be satisfied using the same index. If the optimizer chooses not to use an index to satisfy the IN predicate (perhaps because another index gives better performance), then the IN List appears as a predicate in a filter.

ProcCall algorithm

The ProcCall algorithm, 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 a ProcCall is executed, 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 489

Row Constructor algorithm

The Row Constructor algorithm is a specialized operator that creates a virtual row for use as the input to other algorithms. The Row Constructor algorithm 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 into a Row Constructor algorithm, replacing a scan algorithm over 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 a Row Constructor algorithm has replaced the table scan.

See also

- “DUMMY system table” [*SQL Anywhere Server - SQL Reference*]
- “INSERT statement” [*SQL Anywhere Server - SQL Reference*]
- “Reading execution plans” on page 529

Row Limit algorithm

Row limits are set by the TOP n or FIRST clause of the SELECT statement.

☞ For more information, see “SELECT statement” [*SQL Anywhere Server - SQL Reference*].

Window Functions algorithm

This algorithm is used in evaluating OLAP queries that use window functions.

☞ For more information, see “Window functions” on page 395.

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. It is, however, be semantically equivalent, and may use materialized views instead of the base tables explicitly specified in the query.

☞ For more information about phases a statement goes through until it is executed, see "Phases of query processing" on page 474.

☞ For more information about the rules that the database server follows when rewriting your query, see "Semantic query transformations" on page 476, and "Improving performance with materialized views" on page 494.

The optimizer's job is to understand the semantics of your query and to construct a plan that computes its result. The access plan may not correspond exactly to the syntax you used. The optimizer is free to rewrite your query in any semantically equivalent form.

☞ For more information about the rules SQL Anywhere obeys when rewriting your query, see "Rewriting subqueries as EXISTS predicates" on page 485 and "Semantic query transformations" on page 476.

☞ For information about the methods that the optimizer uses to implement your query, see "Query execution algorithms" on page 507.

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 plan
- Long 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*].

☞ For more information about how to save and see the plan, see "Accessing the execution plan" on page 546. For information about how to read execution plans, see "Text plans" on page 538 and "Graphical plans" on page 539.

Following is an explanation of the statistics and other items that appear in access plans.

Abbreviations used in the plan

Following are the abbreviations that are used in the short plan, and in the short name form of the graphical plan:

Abbreviation	Name
DELETE	Delete
DistH	Hash distinct
DistO	Ordered distinct
DT	Derived table
EAH	Hash except all
EAM	Merge except all
EH	Hash except
EM	Merge except
Exchange	Exchange
Filter	Filter
FS	File scan
GrByH	Hash group by
GrByHClust	Hash group by clustered
GrByHP	Parallel hash group by
GrByHSets	Hash group by sets
GrByO	Ordered group by
GrByOSets	Ordered group by sets
GrByS	Single row group by
GrBySSets	Sorted group by sets
HF	Hash filter
HFP	Parallel hash filter
HTS	Hash table scan
IAH	Hash intersect all
IAM	Merge intersect all

Abbreviation	**Name**
IH	Hash intersect
IM	Merge intersect
IN	In list
INSENSITIVE	Insensitive
INSERT	Insert
IS	Index scan In a short plan, it is *table-name* followed by either <*rowID*>, <*seq*>, or <*rowID*>. In a graphical plan, it is just *table-name*.
ISP	Parallel index scan
JE	Exists join
JH	Hash join
JHE	Exists hash join
JHEP	Exists parallel hash join
JHFO	Full outer hash join
JHNE	Not exists hash join
JHNEP	Not exists parallel hash join
JHO	Left outer hash join
JHP	Parallel hash join
JHPO	Parallel left outer hash join
JHR	Recursive hash join
JHRO	Recursive left outer hash join
JM	Merge join
JMFO	Full outer merge join
JMO	Left outer merge join
JNL	Nested loops join
JNLFO	Full outer nested loops join
JNLO	Left outer nested loops join

Abbreviation	Name
KEYSET	Keyset
LOAD	Load
PC	Procedure call (table function)
PreFilter	Pre filter
RowID Scan	Row identifier scan In a short plan, it is *table-name* <*rowID*>. In a graphical plan, it is just *table-name*.
ROWS	Row constructor
RL	Row limit
RR	Row replicate
RT	Recursive table
RU	Recursive union
SELECT	Select
Sort	Sort (indexed or merge)
SrtN	Sort top n
TS	Table scan In a short plan, it is *table-name* <*seq*>. In a graphical plan, it is just *table-name*.
TSP	Parallel table scan
UA	Union all
UPDATE	Update
Window	Window
Work	Work table

☞ For an explanation of the algorithms, see “Query execution algorithms” on page 507.

Common statistics used in the plan

The following statistics are actual, measured amounts.

Statistic	Explanation
Invocations	Number of times a row was requested from the sub tree.
RowsReturned	Number of rows returned for the current node.

Statistic	Explanation
RunTime	Time required for execution of the sub-tree, including time for children.
CacheHits	Number of successful reads of the cache.
CacheRead	Number of database pages that have been looked up in the cache.
CacheReadTable	Number of table pages that have been read from the cache.
CacheReadIndLeaf	Number of index leaf pages that have been read from the cache.
CacheReadIndInt	Number of index internal node pages that have been read from the cache.
DiskRead	Number of pages that have been read from disk.
DiskReadTable	Number of table pages that have been read from disk.
DiskReadIndLeaf	Number of index leaf pages that have been read from disk.
DiskReadIndInt	Number of index internal node pages that have been read from disk.
DiskWrite	Number of pages that have been written to disk (work table pages or modified table pages).
IndAdd	Number of entries that have been added to indexes.
IndLookup	Number of entries that have been looked up in indexes.
FullCompare	Number of comparisons that have been performed beyond the hash value in an index.

Common estimates used in the plan

Statistic	Explanation
EstRowCount	Estimated number of rows that the node will return each time it is invoked.
AvgRowCount	Average number of rows returned on each invocation. This is not an estimate, but is calculated as RowsReturned / Invocations. If this value is significantly different from EstRowCount, the selectivity estimates may be poor.
EstRunTime	Estimated time required for execution (sum of EstDiskReadTime, EstDiskWriteTime, and EstCpuTime).
AvgRunTime	Average time required for execution (measured).
EstDiskReads	Estimated number of read operations from the disk.
AvgDiskReads	Average number of read operations from the disk (measured).
EstDiskWrites	Estimated number of write operations to the disk.

Statistic	Explanation
AvgDiskWrites	Average number of write operations to the disk (measured).
EstDiskReadTime	Estimated time required for reading rows from the disk.
EstDiskWriteTime	Estimated time required for writing rows to the disk.
EstCpuTime	Estimated processor time required for execution.

Items in the plan related to SELECT, INSERT, UPDATE, and DELETE

Item	Explanation
Optimization Goal	Determines whether query processing is optimized towards returning the first row quickly, or minimizing the cost of returning the complete result set. See "optimization_goal option [database]" [*SQL Anywhere Server - Database Administration*].
Optimization workload	Determines whether query processing is optimized towards a workload that is a mix of updates and reads or a workload that is predominantly read-based. See "optimization_workload option [database]" [*SQL Anywhere Server - Database Administration*].
ANSI update constraints	Controls the range of updates that are permitted (options are Off, Cursors, and Strict). See "ansi_update_constraints option [compatibility]" [*SQL Anywhere Server - Database Administration*]
Optimization level	Reserved.
Select list	List of expressions selected by the query.

Item	Explanation
Materialized views	List of materialized views considered by the optimizer. Each entry in the list is a tuple in the following format: *view-name* [*view-matching-outcome*] [*table-list*] where *view-matching-outcome* reveals the usage of a materialized view; if the value is COSTED, the view was used during enumeration. The *table-list* is a list of query tables that were potentially replaced by this view. Values for *view-matching-outcome* include: ♦ Base table mismatch ♦ Permissions mismatch ♦ Predicate mismatch ♦ Select list mismatch ♦ Costed ♦ Stale mismatch ♦ Snapshot stale mismatch ♦ Cannot be used by optimizer ♦ Cannot be used internally by optimizer ♦ Cannot build definition ♦ Cannot access ♦ Disabled ♦ Options mismatch ♦ Reached view matching threshold ♦ View used ☞ For more information about restrictions and conditions that prevent the optimizer from using a materialized view, see "Improving performance with materialized views" on page 494, and "Restrictions when managing materialized views" on page 69

Items in the plan related to locks

Item	Explanation
Locked tables	List of all locked tables and their isolation levels.

Items in the plan related to scans

Item	Explanation
Table name	Actual name of the table.
Correlation name	Alias for the table.
Estimated rows	Estimated number of rows in the table.
Estimated pages	Estimated number of pages in the table.
Estimated row size	Estimated row size for the table.

Item	Explanation
Page maps	YES when a page map is used to read multiple pages.

Items in the plan related to index scans

Item	Explanation
Index name	Name of the index.
Key type	Can be one of PRIMARY KEY, FOREIGN KEY, CONSTRAINT (unique constraint), or UNIQUE (unique index). The key type does not appear if the index is a non-unique secondary index.
Depth	Height of the index. See "Table and page sizes" on page 549.
Estimated leaf pages	Estimated number of leaf pages.
Cardinality	Cardinality of the index if it is different from the estimated number of rows. This applies only to SQL Anywhere databases version 6.0.0 and earlier.
Selectivity	Estimated number of rows that match the range bounds.
Direction	FORWARD or BACKWARD.
Range bounds	Range bounds are shown as a list (col_name=value) or col_name IN [low, high].

Items in the plan related to joins, filter, and pre-filter

Item	Explanation
Predicate	Search condition that is evaluated in this node, along with selectivity estimates and measurement. See "Selectivity in the plan" on page 544

Items in the plan related to hash filter

Item	Explanation
Build values	Estimated number of distinct values in the input.
Probe values	Estimated number of distinct values in the input when checking the predicate.

Item	Explanation
Bits	Number of bits selected to build the hash map.
Pages	Number of pages required to store the hash map.

Items in the plan related to Union

Item	Explanation
Union List	Columns involved in a UNION operation.

Items in the plan related to GROUP BY

Item	Explanation
Aggregates	All the aggregate functions.
Group-by list	All the columns in the group by clause.

Items in the plan related to DISTINCT

Item	Explanation
Distinct list	All the columns in the distinct clause.

Items in the plan related to IN LIST

Item	Explanation
In List	All the expressions in the specified set.
Expression SQL	Expressions to compare to the list.

Items in the plan related to SORT

Item	Explanation
Order-by	List of all expressions to sort by.

Items in the plan related to row limits

Item	Explanation
Row limit count	Maximum number of rows returned as specified by FIRST or TOP n.

Text plans

There are two types of text representation of a query execution plan: short and long. To choose between the type of text plan in Interactive SQL, open the Options dialog from the Tools menu, and then click the Plan tab. To use SQL functions to access the text plan, see "Accessing the execution plan with SQL functions" on page 546.

There is also a graphical version of the plan. See "Graphical plans" on page 539.

Short text plan

The short 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 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 used in the plan" on page 530.

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 *
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 used in the plan" on page 530.

Long text plan

The long plan provides a little more information than the short 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 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.

```
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 used in the plan" on page 530.

Graphical plans

The graphical plan provides execution plan information using visual cues (shapes, colors, and so on). You can choose to see either a graphical plan, or a graphical plan with statistics. Both allow you to view rapidly which parts of the plan have been estimated as the most expensive. The graphical plan with statistics, though more expensive to view, also 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 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 there to be differences.

By default, SQL Anywhere displays the graphical plan without statistics. However, to change between graphical plans with or without statistics, choose Tools ▶ Options, click the Plan tab, and then choose the type of plan. To access the plan with SQL functions, see “Accessing the execution plan with SQL functions” on page 546.

☞ For more information about text plans, see “Text plans” on page 538.

The graphical plan is designed to provide some key information visually. For example, each operation that appears in the graphical plan is displayed in a container. A container is also referred to as a node. The shape of the container indicates the type of operation performed:

- operations that materialize data are represented by hexagons
- index scans are represented by trapezoids
- table scans are represented by rectangles with square corners
- other operations are represented by rectangles with round corners

You can often determine query performance by looking for thick lines and red borders in the graphical plan. For example:

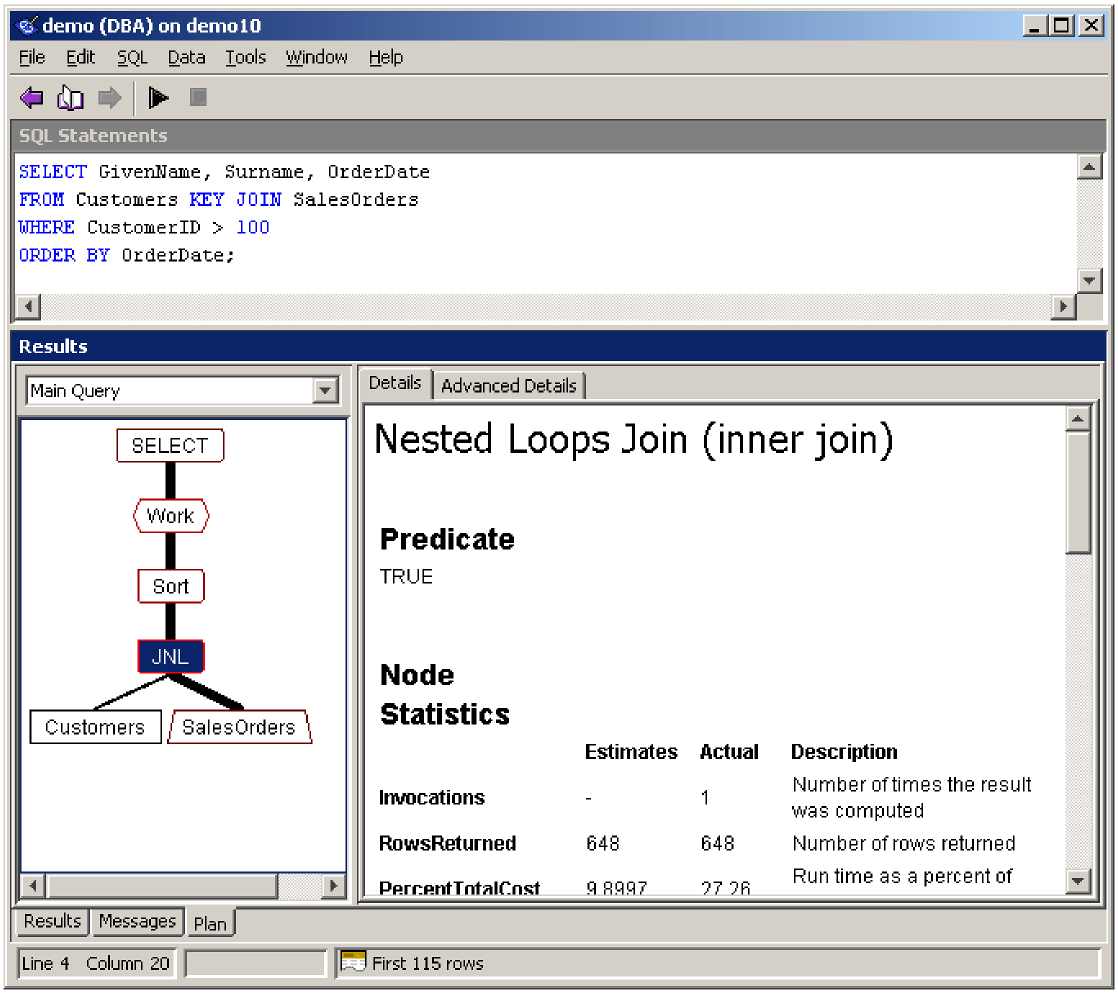
- the number of rows that an operation passes to the next operation in the plan is indicated by the thickness of the line joining the operations. This provides a visual indicator of the operations performed on most data in the query.
- the container for an operation that is particularly slow has a red border.

Customizing the appearance of graphical plans

The appearance of items in the graphical plan is customizable. To change the appearance of the graphical plan, right-click the plan in the lower left pane of Interactive SQL, select Customize, and change the settings accordingly. While you must already have executed a plan in order to customize the appearance of it, your changes are applied to subsequent graphical plans that are displayed.

You can also print a graphical plan by right-clicking on the plan, and choosing 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.



```
SELECT GivenName, Surname, OrderDate
FROM Customers KEY JOIN SalesOrders
WHERE CustomerID > 100
ORDER BY OrderDate;
```

Nested Loops Join (inner join)

Predicate

TRUE

Node Statistics

	Estimates	Actual	Description
Invocations	-	1	Number of times the result was computed
RowsReturned	648	648	Number of rows returned
PercentTotalCost	9.8997	27.26	Run time as a percent of

Context sensitive help

You can obtain detailed information about the nodes in the plan by clicking the node in the graphical diagram and reading the corresponding information in the right pane. In this example, the nested loops join node (JNL) is selected. The information in the right pane pertains only to that node. For example, the Predicate description is `TRUE`, indicating that at this stage in the query execution no predicate is being applied. If you click the Customers table query node, however, the Predicate value changes to be something similar to `Customers.ID > 100 : 100% Index; true 126/126 100%.`

To obtain context-sensitive help for each node in the graphical plan, select the node, right-click it and choose Help.

☞ For more information about the abbreviations used in the plan, see "Abbreviations used in the plan" on page 530.

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 on bypassed queries, see "How the optimizer works" on page 487.

Graphical plan with statistics

The graphical plan with statistics shows you all the estimates that are provided with the graphical plan, but also shows actual runtime costs of executing the statement. To do this, the statement must be executed. This means that there may be a delay in accessing the plan for expensive queries. It also means that any parts of your query, such as deletes or updates, are actually executed; you can perform a rollback to undo these changes.

Use the graphical plan with statistics when you are having performance problems and if the estimated row counts, or run times, differ from your expectations. The graphical plan with statistics provides estimates and actual statistics for you to compare. A large difference between estimated and actual statistics can be a warning sign that the optimizer does not have sufficient information to choose a good access plan.

The database options and other global settings that affect query execution are displayed for the root operator node only.

Following are some of the key statistics you can check in the graphical plan with statistics, and some possible remedies:

- **Selectivity statistics** 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. If the selectivity estimates are poor, the query optimizer may generate a poor access plan. For example, if the optimizer mistakenly estimates a predicate to be highly selective (for example, a selectivity of 5%), but the actual selectivity is much lower (50%), then it may choose an inappropriate plan. Selectivity estimates are not precise, but a significantly large deviation can 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 question. In rare cases, you may want to supply explicit selectivity estimates, although this approach can introduce other problems later when the statistics are updated.

  ☞ For more information about selectivity, see "Selectivity in the plan" on page 544.

  ☞ For more information about creating statistics, see "CREATE STATISTICS statement" [*SQL Anywhere Server - SQL Reference*].

  ☞ For more information about user estimates, see "Explicit selectivity estimates" [*SQL Anywhere Server - SQL Reference*].

  Selectivity statistics are not displayed if the query is determined to be a simple query, so that the optimizer is bypassed. For more information on simple queries, see "How the optimizer works" on page 487.

  Indicators of poor selectivity occur in the following places:

- ♦ **RowsReturned actuals and estimates** 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. A significant difference between the estimated rows returned and the actual number returned is a warning sign that the optimizer is working on poor selectivity information.
- ♦ **Predicate selectivity actuals and estimates** Look for the Predicate subheading to see predicate selectivities. For information about reading the predicate information, see "Selectivity in the plan" on page 544.

  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.

  An estimate source of Guess indicates that the optimizer has no selectivity information to use. If the estimate source is Index and the selectivity estimate is incorrect, your problem may be that the index is skewed; you may benefit from defragmenting the index with the REORGANIZE TABLE statement. See "REORGANIZE TABLE statement" [*SQL Anywhere Server - SQL Reference*].
  For a complete list of the possible sources of selectivity estimates, see "ESTIMATE_SOURCE function [Miscellaneous]" [*SQL Anywhere Server - SQL Reference*].

♦ **Cache reads and hits** If the number of cache reads and cache hits are exactly the same, this indicates that all the information needed to execute the query is in cache. When reads are greater than hits, it means that the database server is attempting to go to cache but failing, and that it must read from disk. In some cases, such as hash joins, this is expected. In other cases, such as nested loops joins, a poor cache-hit ratio may indicate a performance problem, and you may benefit from increasing your cache size.

☞ For more information about cache management, see "Increase the cache size" on page 240.

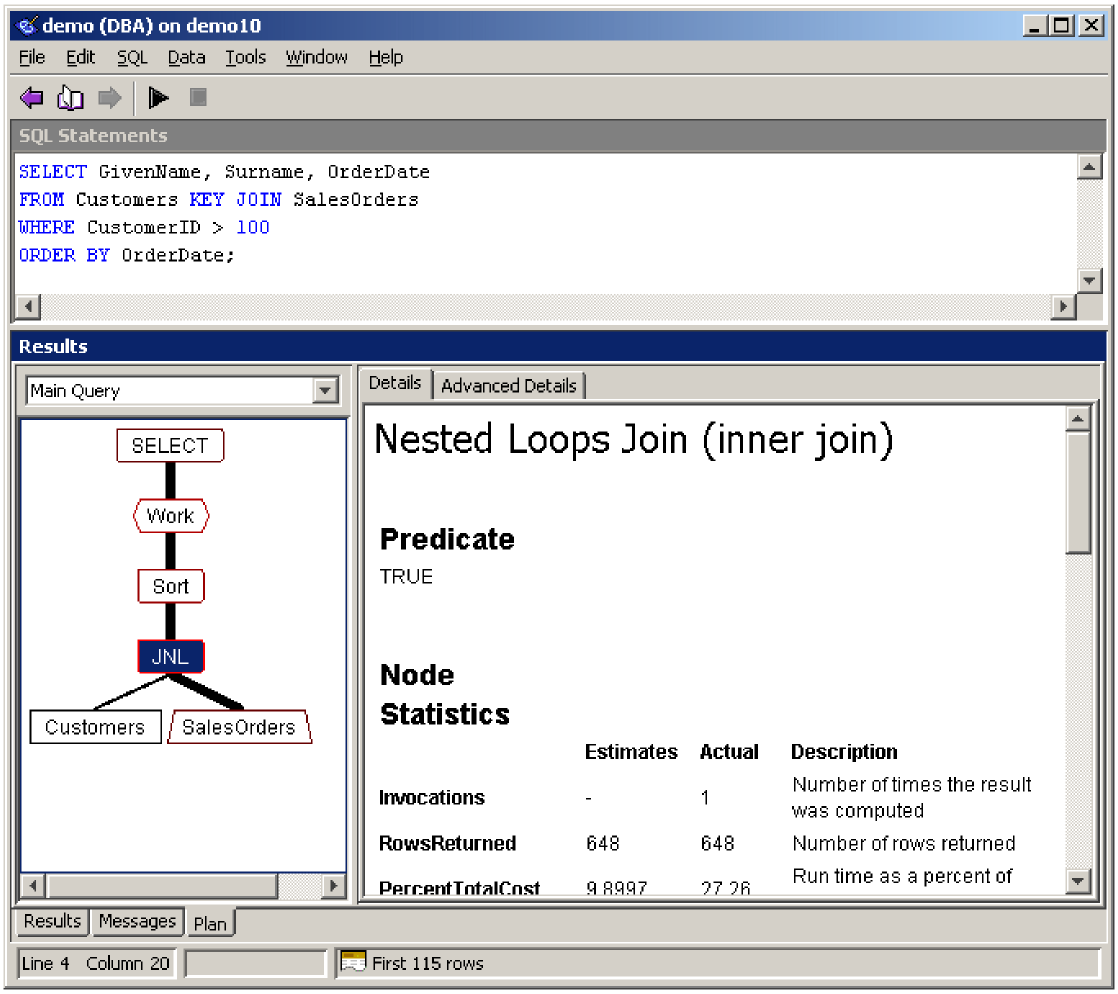
♦ **Lack of effective indexes** It is often not obvious from query execution plans whether an index would help provide better performance or not. Some of the scan-based algorithms used in SQL Anywhere provide excellent performance for many queries without using indexes.

☞ For more information about indexes and performance, see "Use indexes effectively" on page 251 and "Index Consultant" on page 195.

♦ **Data fragmentation problems** The Runtime actual and estimated values are provided in the root node statistics. Runtime measures the time to execute the query. If the runtime is incorrect for a table scan or index scan, you may improve performance by executing the REORGANIZE TABLE statement.

☞ For more information, see "REORGANIZE TABLE statement" [*SQL Anywhere Server - SQL Reference*] and "Reducing table fragmentation" on page 245.

Following is an example of the graphical plan with statistics. Again, the nested loops join node is selected. The statistics in the right pane indicate the resources used by that part of the query.



```
SELECT GivenName, Surname, OrderDate
FROM Customers KEY JOIN SalesOrders
WHERE CustomerID > 100
ORDER BY OrderDate;
```

Nested Loops Join (inner join)

Predicate

TRUE

Node Statistics

	Estimates	Actual	Description
Invocations	-	1	Number of times the result was computed
RowsReturned	648	648	Number of rows returned
PercentTotalCost	9.8997	27.26	Run time as a percent of

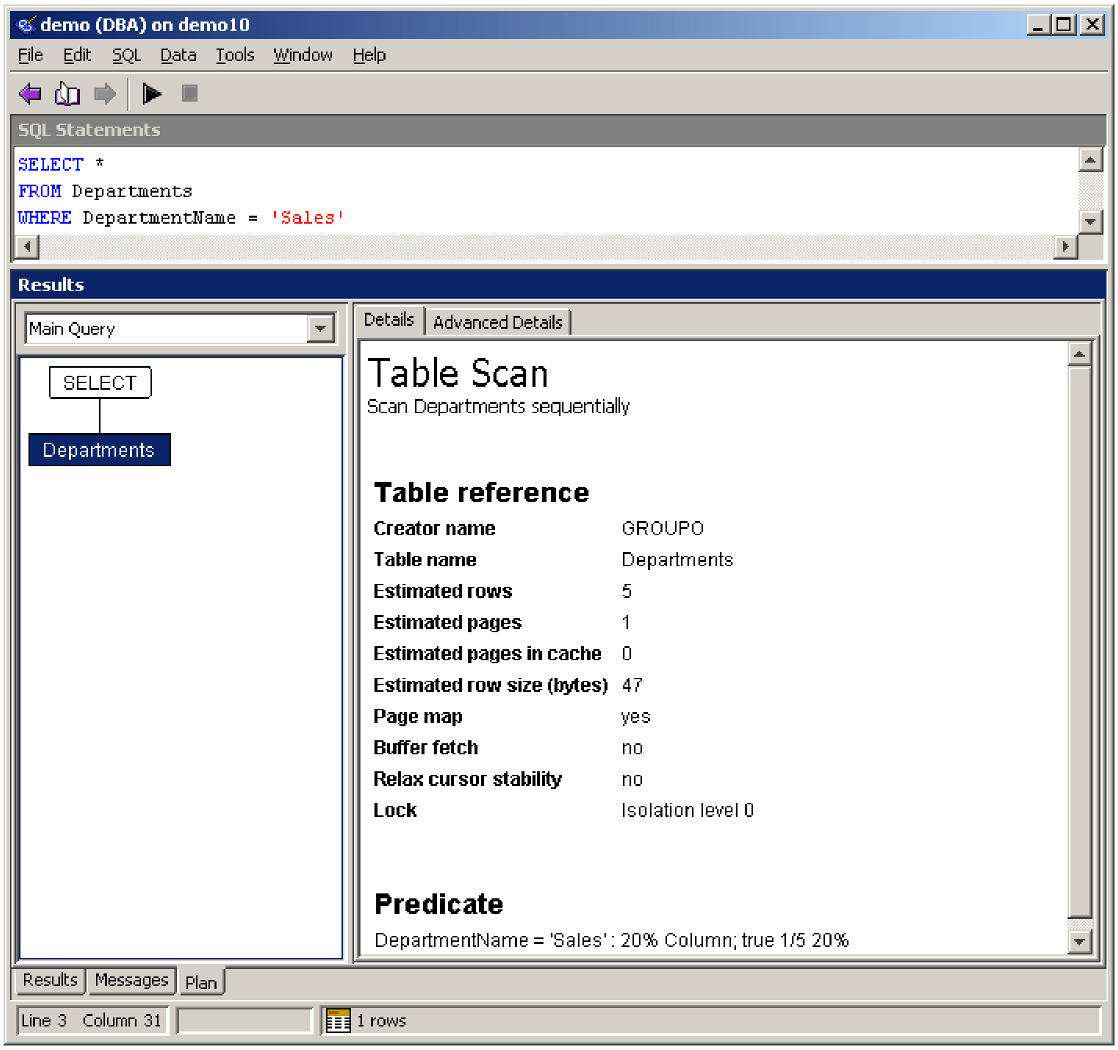
☞ For more information about code words used in the plan, see "Abbreviations used in the plan" on page 530.

Selectivity in the plan

Following is an example of the Predicate showing selectivity of a search condition. In this example, 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.

Selectivity information may not be displayed for simple queries that qualify for optimization bypass. For more information on simple queries, see "How the optimizer works" on page 487.

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.



```
SELECT *
FROM Departments
WHERE DepartmentName = 'Sales'
```

Table Scan

Scan Departments sequentially

Table reference

Creator name	GROUPO
Table name	Departments
Estimated rows	5
Estimated pages	1
Estimated pages in cache	0
Estimated row size (bytes)	47
Page map	yes
Buffer fetch	no
Relax cursor stability	no
Lock	Isolation level 0

Predicate

DepartmentName = 'Sales' : 20% Column; true 1/5 20%

This predicate description is

```
Departments.DepartmentName = 'Sales' : 20% Column; true 1/5 20%
```

This can be read as follows:

- `Departments.DepartmentName = 'Sales'` is the predicate.
- `20%` is the optimizer's estimate of the selectivity. That is, the optimizer is basing its query access selection on the estimate that 20% of the rows satisfy the predicate.

  This is the same output as is provided by the ESTIMATE function. For more information, see "ESTIMATE function [Miscellaneous]" [*SQL Anywhere Server - SQL Reference*].

- `Column` is the source of the estimate. This is the same output as is provided by the ESTIMATE_SOURCE function. For a complete list of the possible sources of selectivity estimates, see "ESTIMATE_SOURCE function [Miscellaneous]" [*SQL Anywhere Server - SQL Reference*].
- `true 1/5 20%` is the actual selectivity of the predicate during execution. The predicate was evaluated five times, and was true once, 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.

Accessing the execution plan

You can see the execution plan in Interactive SQL, or using SQL functions. The plan is also displayed from the Index Consultant.

Accessing the execution plan in Interactive SQL

The following types of plan are available in Interactive SQL:

- Short text plans, see "Short text plan" on page 538
- Long text plans, see "Long text plan" on page 539
- Graphical plans with or without statistics"Graphical plans" on page 539

To choose a different type of plan, click Tools ► Options, click the Plan tab, and then choose a plan type.

To see the plan, execute a query and then click the Plan tab located at the bottom of the Interactive SQL window.

Accessing the execution plan with SQL functions

You can access the execution plan using SQL functions, and retrieve the output in XML format.

- To access the short plan, see the "EXPLANATION function [Miscellaneous]" [*SQL Anywhere Server - SQL Reference*].
- To access the long plan, see the "PLAN function [Miscellaneous]" [*SQL Anywhere Server - SQL Reference*].

- To access the graphical plan, see the "GRAPHICAL_PLAN function [Miscellaneous]" [*SQL Anywhere Server - SQL Reference*].

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 CE 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 will be started on the same server.

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 195.

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 on the ISYSIDX and ISYSPHYSIDX system tables, see their corresponding views, "SYSIDX system view" [*SQL Anywhere Server - SQL Reference*] and "SYSPHYSIDX system view" [*SQL Anywhere Server - SQL Reference*].

Using logical indexes means that the database server does not need to create and maintain duplicate physical indexes, since more than one logical index can point to a single physical index.

When you delete a logical index, its definition is removed from the ISYSIDX system table. If it was the only logical index referencing a particular physical index, the physical index is also deleted, 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 552.

Physical indexes are not created for remote tables. For temporary tables, physical indexes are created, but they are not recorded in ISYPHYSIDX, 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:

```
SELECT tab.table_name, idx.table_id, phys.phys_index_id, COUNT(*)
 FROM SYSIDX idx JOIN SYSTAB tab ON (idx.table_id = tab.table_id)
 JOIN SYSPHYSIDX phys ON ( idx.phys_index_id  = phys.phys_index_id
   AND idx.table_id = phys.table_id )
 GROUP BY tab.table_name, idx.table_id, phys.phys_index_id
   HAVING COUNT(*) > 1
ORDER BY tab.table_name;
```

Following is an example result set for the query:

table_name	table_id	phys_index_id	COUNT(*)
ISYSCHECK	57	0	2
ISYSCOLSTAT	50	0	2
ISYSFKEY	6	0	2
ISYSSOURCE	58	0	2
MAINLIST	94	0	3
MAINLIST	94	1	2

The number of rows for each table indicates the number of shared physical indexes for the tables. In this example, all of the tables have one shared physical index, except for the ficticious 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 84
- "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 554.

- **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 195.

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 80.

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 CE.

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 532.

☞ For more information on 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 549.

You can see the number of levels in an index by using the sa_index_levels system procedure. See "sa_index_levels system procedure" [*SQL Anywhere Server - SQL Reference*].

Composite indexes

An index can contain one, two, or more columns. An index on two or more columns is called a **composite index**. For example, the following statement creates a two-column composite index:

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

A composite index is useful if the first column alone does not provide high selectivity. For example, a composite index on Surname and GivenName is useful when many employees have the same surname. A composite index on EmployeeID and Surname would not be useful because each employee has a unique ID, so the column Surname does not provide any additional selectivity.

Additional columns in an index can allow you to narrow down your search, but having a two-column index is not the same as having two separate indexes. A composite index is structured like a telephone book, which first sorts people by their surnames, and then all the people with the same surname by their given names. A telephone book is useful if you know the surname, even more useful if you know both the given name and the surname, but worthless if you only know the given name and not the surname.

Column order

When you create composite indexes, you should think carefully about the order of the columns. Composite indexes are useful for doing searches on all 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.

```
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 think it likely that you will need to look up people by given name only or surname only, then you should consider creating both of these indexes.

Alternatively, you could make two indexes, each containing only one of the columns. Remember, however, that SQL Anywhere only uses one index to access any one table while processing a single query. Even if you know both names, it is likely that SQL Anywhere needs to read extra rows, looking for those with the correct second name.

When you create an index using the CREATE INDEX command, as in the example above, the columns appear in the order shown in your command.

Composite indexes and ORDER BY

By default, the columns of an index are sorted in ascending order, but they can optionally be sorted in descending order by specifying DESC in the CREATE INDEX statement.

SQL Anywhere can choose to use an index to optimize an ORDER BY query as long as the ORDER BY clause contains only columns included in that index. In addition, the columns in the index must be ordered in exactly the same way, or in exactly the opposite way, as the ORDER BY clause. For single-column indexes, the ordering is always such that it can be optimized, but composite indexes require slightly more thought. The table below shows the possibilities for a two-column index.

Index columns	Optimizable ORDER BY queries	Not optimizable ORDER BY queries
ASC, ASC	ASC, ASC or DESC, DESC	ASC, DESC or DESC, ASC
ASC, DESC	ASC, DESC or DESC, ASC	ASC, ASC or DESC, DESC
DESC, ASC	DESC, ASC or ASC, DESC	ASC, ASC or DESC, DESC
DESC, DESC	DESC, DESC or ASC, ASC	ASC, DESC or DESC, ASC

An index with more than two columns follows the same general rule as above. For example, suppose you have the following index:

```
CREATE INDEX idx_example
ON table1 ( col1 ASC, col2 DESC, col3 ASC );
```

In this case, the following queries can be optimized:

```
SELECT col1, col2, col3 FROM table1
ORDER BY col1 ASC, col2 DESC, col3 ASC;

SELECT col1, col2, col3 FROM example
ORDER BY col1 DESC, col2 ASC, col3 DESC;
```

The index is not used to optimize a query with any other pattern of ASC and DESC in the ORDER BY clause. For example, the following statement is not optimized:

```
SELECT col1, col2, col3 FROM table1
ORDER BY col1 ASC, col2 ASC, col3 ASC;
```

Other uses for indexes

SQL Anywhere uses indexes to achieve other performance benefits. Having an index allows SQL Anywhere to enforce column uniqueness, to reduce the number of rows and pages that must be locked, and to better estimate the selectivity of a predicate.

♦ **Enforce column uniqueness** Without an index, SQL Anywhere has to scan the entire table every time that a value is inserted to ensure that it is unique. For this reason, SQL Anywhere automatically builds an index on every column with a uniqueness constraint.

♦ **Reduce locks** Indexes reduce the number of rows and pages that must be locked during inserts, updates, and deletes. This reduction is a result of the ordering that indexes impose on a table.

  For more information on indexes and locking, see “How locking works” on page 155.

♦ **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 81
♦ “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 IV. 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

About this chapter

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. Complete, detailed information about compliance is provided in the reference documentation for each feature of SQL Anywhere.

This chapter describes those features of SQL Anywhere that are not commonly found in other SQL implementations. It is also a guide for creating applications that are compatible with Sybase Adaptive Server Enterprise, which uses Transact-SQL language elements and statements.

SQL Anywhere SQL features

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 "UltraLite SQL Function Reference 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 will then enforce 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 will then ensure referential integrity for these columns - whenever these columns are modified or a row is inserted into this table, these columns will be 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 will 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. For more information, 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.

```
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 "UltraLite SQL Function Reference 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**, which is 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.

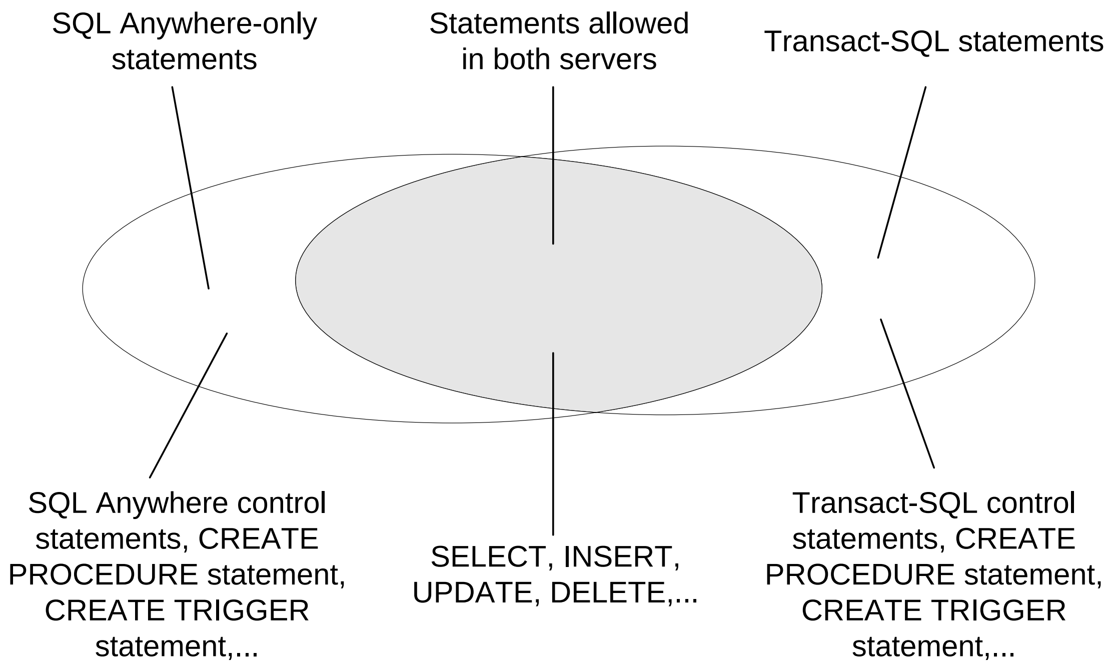
The aim is to write applications to work with both Adaptive Server Enterprise and SQL Anywhere. Existing Adaptive Server Enterprise applications generally require some changes to run on a SQL Anywhere database.

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.


SQL Anywhere-only statements
Statements allowed in both servers
Transact-SQL statements
SQL Anywhere control statements, CREATE PROCEDURE statement, CREATE TRIGGER statement,...
SELECT, INSERT, UPDATE, DELETE,...
Transact-SQL control statements, CREATE PROCEDURE statement, CREATE TRIGGER statement,...

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.

☞ For information about Open Server support, see “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 on 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 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.

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.

☞ For a description of the SQL Anywhere syntax for these statements, see "SQL Statements" [*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. You'll find a list and individual descriptions in "Views for Transact-SQL compatibility" [*SQL Anywhere Server - SQL Reference*], which describes the system catalogs of the two products. This section provides a brief overview of the differences.

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 ID **dbo**) owns the system tables. In SQL Anywhere, the system owner (user ID **SYS**) owns the system tables. A dbo user ID 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 ID **DBA**.
- The RESOURCE permission 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.

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.

☞ For the arguments to each procedure, see "Adaptive Server Enterprise system and catalog procedures" [*SQL Anywhere Server - SQL Reference*].

System procedure	Description
sp_addlogin	In Adaptive Server Enterprise, this adds a user to the server. In SQL Anywhere, this adds a user to a database.
sp_adduser	In Adaptive Server Enterprise and SQL Anywhere, this adds a user to a database. While this is a distinct task from sp_addlogin in Adaptive Server Enterprise, in SQL Anywhere, they are the same.
sp_addgroup	Adds a group to a database.
sp_changegroup	Adds a user to a group, or moves a user from one group to another.
sp_droplogin	In Adaptive Server Enterprise, removes a user from the server. In SQL Anywhere, removes a user from the database.
sp_dropuser	Removes a user from the database.
sp_dropgroup	Removes a group from the database.

In Adaptive Server Enterprise, login IDs are server-wide. In SQL Anywhere, users belong to individual databases.

Database object permissions

The Adaptive Server Enterprise and SQL Anywhere GRANT and REVOKE statements for granting permissions on individual database objects are very similar. Both allow SELECT, INSERT, DELETE, UPDATE, and REFERENCES permissions on database tables and views, and UPDATE permissions on selected columns of database tables. Both allow EXECUTE permissions to be granted on stored procedures.

For example, the following statement is valid in both Adaptive Server Enterprise and SQL Anywhere:

```
GRANT INSERT, DELETE
ON Employees
TO MARY, SALES;
```

This statement grants permission to use the INSERT and DELETE statements on the Employees table to user MARY and to the SALES group.

Both SQL Anywhere and Adaptive Server Enterprise support the WITH GRANT OPTION clause, allowing the recipient of permissions to grant them in turn, although SQL Anywhere does not permit WITH GRANT OPTION to be used on a GRANT EXECUTE statement. In SQL Anywhere, you can only specify WITH GRANT OPTION for users. Members of groups do not inherit the WITH GRANT OPTION if it is granted to a group.

Database-wide permissions

Adaptive Server Enterprise and SQL Anywhere use different models for database-wide user permissions. These are discussed in "Users and groups" on page 569. 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.

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 10 ► Create Database.
3. Follow the instructions in the wizard.
4. When you see a button called 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)

- Enter the following command at a command prompt:

  ```
  dbinit -b -c -k db-name.db
  ```

  ☞ For more information about these options, see "Initialization utility options" [*SQL Anywhere Server - Database Administration*].

♦ To create a Transact-SQL compatible database (SQL)

1. Connect to any SQL Anywhere database.
2. Enter the following statement, for example, in Interactive SQL:

   ```
   CREATE DATABASE 'dbname.db'
   ASE COMPATIBLE
   CASE RESPECT
   BLANK PADDING ON;
   ```

   In this statement, ASE COMPATIBLE means compatible with Adaptive Server Enterprise. It prevents the SYS.SYSCOLUMNS and SYS.SYSINDEXES views from being created.

Make the database case sensitive

By default, string comparisons in Adaptive Server Enterprise databases are case sensitive, while those in SQL Anywhere are case insensitive.

When building an Adaptive Server Enterprise-compatible database using SQL Anywhere, choose the case sensitive option.

- ♦ If you are using Sybase Central, this option is in the Create Database wizard.
- ♦ If you are using the dbinit utility, specify the -c option.

Ignore trailing blanks in comparisons

When building an Adaptive Server Enterprise-compatible database using SQL Anywhere, choose the option to ignore trailing blanks in comparisons.

- ♦ If you are using Sybase Central, this option is in the Create Database wizard.
- ♦ If you are using the dbinit utility, specify the -b option.

When you choose this option, Adaptive Server Enterprise and SQL Anywhere considers the following two strings equal:

```
'ignore the trailing blanks   '
'ignore the trailing blanks'
```

If you do not choose this option, SQL Anywhere considers the two strings above different.

A side effect of choosing this option is that strings are padded with blanks when fetched by a client application.

Remove historical system views

Older versions of SQL Anywhere employed two system views whose names conflict with the Adaptive Server Enterprise system views provided for compatibility. These views include SYSCOLUMNS and SYSINDEXES. If you are using Open Client or JDBC interfaces, create your database excluding these views. You can do this with the dbinit -k option.

If you do not use this option when creating your database, the following statement results in the error, "Table name 'SYSCOLUMNS' is ambiguous.":

```
SELECT * FROM SYSCOLUMNS ;
```

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.

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.

For more information on the quoted_identifier option, see "quoted_identifier option [compatibility]" [*SQL Anywhere Server - Database Administration*].

Set the automatic_ timestamp option to On

Transact-SQL defines a timestamp column with special properties. With the automatic_timestamp option set to On, the SQL Anywhere treatment of timestamp columns is similar to Adaptive Server Enterprise behavior.

With the automatic_timestamp option set to On in SQL Anywhere (the default setting is Off), any new columns with the TIMESTAMP data type that do not have an explicit default value defined receive a default value of timestamp.

☞ For information on timestamp columns, see "The special Transact-SQL timestamp column and data type" on page 576.

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.

☞ For more information on the string_rtruncation option, see "string_rtruncation option [compatibility]" [*SQL Anywhere Server - Database Administration*].

☞ For more information on database options for Transact-SQL compatibility, 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:

```
CREATE TABLE tablename (
   column_1 INTEGER ,
   column_2 TIMESTAMP DEFAULT TIMESTAMP
);
```

The following ALTER TABLE statement adds a Transact-SQL timestamp column to the SalesOrders table:

```
ALTER TABLE SalesOrders
ADD timestamp TIMESTAMP DEFAULT TIMESTAMP;
```

In Adaptive Server Enterprise a column with the name timestamp and no data type specified automatically receives a TIMESTAMP data type. In SQL Anywhere you must explicitly assign the data type yourself.

If you have the automatic_timestamp database option set to On, you do not need to set the default value: any new column created with TIMESTAMP data type and with no explicit default receives a default value of timestamp. The following statement sets automatic_timestamp to On:

```
SET OPTION PUBLIC.automatic_timestamp='On';
```

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.

The value of @@identity changes each time a statement attempts to insert a row into a table.

- If the statement affects a table without an IDENTITY column, @@identity is set to 0.
- If the statement inserts multiple rows, @@identity reflects the last value inserted into the IDENTITY column.

This change is permanent. @@identity does not revert to its previous value if the statement fails or if the transaction that contains it is rolled back.

☞ For more information on 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. 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 on this option, see "Setting options for Transact-SQL compatibility" on page 573.

Temporary tables

You can create a temporary table by placing a pound sign (#) in front of a CREATE TABLE statement. These temporary tables are SQL Anywhere declared temporary tables, and are available only in the current connection. For information about declared temporary tables in SQL Anywhere, see "DECLARE LOCAL TEMPORARY TABLE statement" [*SQL Anywhere Server - SQL Reference*].

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.

☞ For information about the CREATE TABLE statement, see "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*"

☞ For a full description of the SELECT statement, see "SELECT statement" [*SQL Anywhere Server - SQL Reference*].

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
- 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.

  See also, "OLAP Support" on page 383.
- 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.

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 321, and “FROM clause” [*SQL Anywhere Server - SQL Reference*].

☞ For more information on Transact-SQL compatibility of joins, see “Transact-SQL outer joins (*= or =*)” on page 338.

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 executes triggers after the triggering statement has completed: they are **statement level**, **after** triggers. SQL Anywhere supports both **row level** triggers (which execute before or after each row has been modified) and statement level triggers (which execute after the entire statement).

Row-level triggers are not part of the Transact-SQL compatibility features, and are discussed in “Using Procedures, Triggers, and Batches” on page 723.

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. Adaptive Server Enterprise does not support parameters.

☞ For information on parameters, 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:

```
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:

```
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:

```
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 750

Variables in Transact-SQL procedures

SQL Anywhere uses the SET statement to assign values to variables in a procedure. In Transact-SQL, values are assigned using either the SELECT statement with an empty table-list, or the SET statement. The following simple procedure illustrates how the Transact-SQL syntax works:

```
CREATE PROCEDURE multiply
               @mult1 int,
               @mult2 int,
               @result int output
AS
SELECT @result = @mult1 * @mult2;
```

This procedure can be called as follows:

```
CREATE VARIABLE @product int
go
EXECUTE multiply 5, 6, @product OUTPUT
go
```

The variable @product has a value of 30 after the procedure executes.

☞ For more information on using the SELECT statement to assign variables, see "Writing compatible queries" on page 580. For more information on using the SET statement to assign variables, see "SET statement" [*SQL Anywhere Server - SQL Reference*].

Error handling in Transact-SQL procedures

Default procedure error handling is different in the Watcom-SQL and Transact-SQL dialects. By default, Watcom-SQL dialect procedures exit when they encounter an error, returning SQLSTATE and SQLCODE values to the calling environment.

Explicit error handling can be built into Watcom-SQL stored procedures using the EXCEPTION statement, or you can instruct the procedure to continue execution at the next statement when it encounters an error, using the ON EXCEPTION RESUME statement.

When a Transact-SQL dialect procedure encounters an error, execution continues at the following statement. The global variable @@error holds the error status of the most recently executed statement. You can check this variable following a statement to force return from a procedure. For example, the following statement causes an exit if an error occurs.

```
IF @@error != 0 RETURN
```

When the procedure completes execution, a return value indicates the success or failure of the procedure. This return status is an integer, and can be accessed as follows:

```
DECLARE @Status INT
EXECUTE @Status = proc_sample
IF @Status = 0
   PRINT 'procedure succeeded'
ELSE
   PRINT 'procedure failed'
```

The following table describes the built-in procedure return values and their meanings:

Value	Meaning
0	Procedure executed without error
–1	Missing object
–2	Data type error
–3	Process was chosen as deadlock victim
–4	Permission error
–5	Syntax error
–6	Miscellaneous user error
–7	Resource error, such as out of space
–8	Non-fatal internal problem
–9	System limit was reached
–10	Fatal internal inconsistency
–11	Fatal internal inconsistency

Value	Meaning
–12	Table or index is corrupt
–13	Database is corrupt
–14	Hardware error

The RETURN statement can be used to return other integers, with their own user-defined meanings.

Using the RAISERROR statement in procedures

The RAISERROR statement is a Transact-SQL statement for generating user-defined errors. It has a similar function to the SIGNAL statement.

☞ For a description of the RAISERROR statement, see "RAISERROR statement [T-SQL]" [*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.

You lose intermediate RAISERROR statuses and codes after the procedure terminates. If, at return time, an error occurs along with the RAISERROR, then the error information is returned and you lose the RAISERROR information. The application can query intermediate RAISERROR statuses by examining @@error global variable at different execution points.

Transact-SQL-like error handling in the Watcom-SQL dialect

You can make a Watcom-SQL dialect procedure handle errors in a Transact-SQL-like manner by supplying the ON EXCEPTION RESUME clause to the CREATE PROCEDURE statement:

```
CREATE PROCEDURE sample_proc()
ON EXCEPTION RESUME
BEGIN
   ...
END
```

The presence of an ON EXCEPTION RESUME clause prevents explicit exception handling code from being executed, so avoid using these two clauses together.

Part V. XML in the Database

This part describes how to use XML in the database.

CHAPTER 16

Using XML in the Database

Contents

About this chapter

This chapter provides a summary of the XML support in SQL Anywhere, including importing, exporting, storing, and querying XML data.

What is 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:

```
<hat>bowler</hat>
```

If you write a query that specifies that the element content is of type XML, then the greater than and less than signs are not quoted, as follows:

```
SELECT XMLFOREST( CAST( '<hat>bowler</hat>' AS XML )
AS product )
```

You get the following result:

```
<product><hat>bowler</hat></product>
```

However, if the query does not specify that the element content is of type XML, for example:

```
SELECT XMLFOREST( '<hat>bowler</hat>' AS product )
```

In this case, the less than and greater than signs are replaced with entity references as follows:

```
<product>&lt;hat&gt;bowler&lt;/hat&gt;</product>
```

Note that attributes are always quoted, regardless of the data type.

☞ For more information about how element content is escaped, see “Encoding illegal XML names” on page 606.

☞ 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 597.

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
- ♦ **. (single period)** indicates the current node of the XML document
- ♦ ***//*** indicates all descendants of the current node, including the current node
- ♦ **..** indicates the parent node of the current node
- ♦ ***./@attributename*** indicates the attribute of the current node having the name *attributename*
- ♦ ***./childname*** indicates the children of the current node that are elements having the name *childname*

Consider the following XML document:

```
<inventory>
  <product ID="301" size="Medium">Tee Shirt
    <quantity>54</quantity>
  </product>
  <product ID="302" size="One Size fits all">Tee Shirt
    <quantity>75</quantity>
  </product>
  <product ID="400" size="One Size fits all">Baseball Cap
    <quantity>112</quantity>
  </product>
</inventory>
```

The <inventory> element is the root node. You can refer to it using the following XPath expression:

```
/inventory
```

Suppose that the current node is a <quantity> element. You can refer to this node using the following XPath expression:

```
.
```

To find all the <product> elements that are children of the <inventory> element, use the following XPath expression:

```
/inventory/product
```

If the current node is a <product> element and you want to refer to the size attribute, use the following XPath expression:

```
./@size
```

☞ For a complete list of XPath syntax supported by openxml, see "openxml system procedure" [*SQL Anywhere Server - SQL Reference*].

☞ For information about the XPath query language, see http://www.w3.org/TR/xpath.

Generating a result set using openxml

Each match for the first *xpath-query* argument to openxml generates one row in the result set. The WITH clause specifies the schema of the result set and how the value is found for each column in the result set. For example, consider the following query:

```
SELECT * FROM openxml( '<inventory>
                         <product>Tee Shirt
                           <quantity>54</quantity>
                           <color>Orange</color>
                         </product>
                         <product>Baseball Cap
                 <quantity>112</quantity>
                           <color>Black</color>
                         </product>
                        </inventory>',
                       '/inventory/product' )
WITH ( Name CHAR (25) './text()',
       Quantity CHAR(3) 'quantity',
       Color CHAR(20) 'color')
```

The first *xpath-query* argument is **/inventory/product**, and there are two <product> elements in the XML, so two rows are generated by this query.

The WITH clause specifies that there are three columns: Name, Quantity, and Color. The values for these columns are taken from the <product>, <quantity> and <color> elements. The query above generates the following result:

Name	Quantity	Color
Tee Shirt	54	Orange
Baseball Cap	112	Black

☞ For more information, see "openxml system procedure" [*SQL Anywhere Server - SQL Reference*].

Using openxml to generate an edge table

The openxml procedure can be used to generate an edge table, a table that contains a row for every element in the XML document. You may want to generate an edge table so that you can query the data in the result set using SQL.

The following SQL statement creates a variable, x, that contains an XML document. The XML generated by the query has a root element called <root>, which is generated using the XMLELEMENT function, and elements are generated for each column in the Employees, SalesOrders, and Customers tables using FOR XML AUTO with the ELEMENTS modifier specified.

☞ For information about the XMLELEMENT function, see "XMLELEMENT function [String]" [*SQL Anywhere Server - SQL Reference*].

☞ For information about FOR XML AUTO, see "Using FOR XML AUTO" on page 610.

```
CREATE VARIABLE x XML;
SET x=(SELECT XMLELEMENT( NAME root,
         (SELECT * FROM Employees
         KEY JOIN SalesOrders
         KEY JOIN Customers
         FOR XML AUTO, ELEMENTS)));
SELECT x;
```

The generated XML looks as follows (the result has been formatted to make it easier to read—the result returned by the query is one continuous string):

```
<root>
 <Employees>
  <EmployeeID>299</EmployeeID>
  <ManagerID>902</ManagerID>
  <Surname>Overbey</Surname>
  <GivenName>Rollin</GivenName>
  <DepartmentID>200</DepartmentID>
  <Street>191 Companion Ct.</Street>

  <City>Kanata</City>
  <State>CA</State>
  <Country>USA</Country>
  <PostalCode>94608</PostalCode>
  <Phone>5105557255</Phone>
  <Status>A</Status>
  <SocialSecurityNumber>025487133</SocialSecurityNumber>
  <Salary>39300.000</Salary>
  <StartDate>1987-02-19</StartDate>
  <BirthDate>1964-03-15</BirthDate>
  <BenefitHealthInsurance>Y</BenefitHealthInsurance>
  <BenefitLifeInsurance>Y</BenefitLifeInsurance>
  <BenefitDayCare>N</BenefitDayCare>
  <Sex>M</Sex>

  <SalesOrders>
  <ID>2001</ID>
  <CustomerID>101</CustomerID>
  <OrderDate>2000-03-16</OrderDate>
  <FinancialCode>r1</FinancialCode>
  <Region>Eastern</Region>
  <SalesRepresentative>299</SalesRepresentative>

    <Customers>
    <ID>101</ID>
    <Surname>Devlin</Surname>
    <GivenName>Michael</GivenName>
    <Street>114 Pioneer Avenue</Street>
    <City>Kingston</City>
    <State>NJ</State>
    <PostalCode>07070</PostalCode>
```

```
      <Phone>2015558966</Phone>
      <CompanyName>The Power Group</CompanyName>
      </Customers>
    </SalesOrders>
  </Employees>
  ...
```

The following query uses the descendant-or-self (**//***) XPath expression to match every element in the above XML document, and for each element the id metaproperty is used to obtain an ID for the node, and the parent (**../**) XPath expression is used with the ID metaproperty to get the parent node. The localname metaproperty is used to obtain the name of each element. 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.

ID	parent	name	text
5	(NULL)	root	(NULL)
16	5	Employees	(NULL)
28	16	EmployeeID	299
55	16	ManagerID	902
79	16	Surname	Overbey
...	...	...	...

Using openxml with xp_read_file

So far, XML that was generated with a procedure like XMLELEMENT has been used. You can also read XML from a file and parse it using the xp_read_file procedure. Suppose the file *c:\inventory.xml* has the following contents:

```
<inventory>
   <product>Tee Shirt
      <quantity>54</quantity>
      <color>Orange</color>
   </product>
   <product>Baseball Cap
      <quantity>112</quantity>
      <color>Black</color>
   </product>
</inventory>
```

You can use the following statement to read and parse the XML in the file:

```
CREATE VARIABLE x XML;
SELECT xp_read_file( 'c:\\inventory.xml' )
 INTO x;
SELECT * FROM openxml( x, '//*' )
 WITH (ID INT '@mp:id',
       parent INT '../@mp:id',
       name CHAR(128) '@mp:localname',
       text LONG VARCHAR 'text()' )
ORDER BY ID;
```

Querying XML in a column

If you have a table with a column that contains XML, you can use openxml to query all the XML values in the column at once. This can be done using a lateral derived table.

The following statements create a table with two columns, ManagerID and Reports. The Reports column contains XML data generated from the Employees table.

```
CREATE TABLE test (ManagerID INT, Reports XML);
INSERT INTO test
SELECT ManagerID, XMLELEMENT( NAME reports,
              XMLAGG( XMLELEMENT( NAME e, EmployeeID)))
FROM Employees
GROUP BY ManagerID;
```

Execute the following query to view the data in the test table:

```
SELECT * FROM test
ORDER BY ManagerID;
```

This query produces the following result:

ManagerID	Reports

501	`<reports>`
	` <e>102</e>`
	` <e>105</e>`
	` <e>160</e>`
	` <e>243</e>`
	` ...`
	`</reports>`

703	`<reports>`
	` <e>191</e>`
	` <e>750</e>`
	` <e>868</e>`
	` <e>921</e>`
	` ...`
	`</reports>`

902	`<reports>`
	` <e>129</e>`
	` <e>195</e>`
	` <e>299</e>`
	` <e>467</e>`
	` ...`
	`</reports>`

ManagerID	**Reports**
1293	`<reports>` `<e>148</e>` `<e>390</e>` `<e>586</e>` `<e>757</e>` `...` `</reports>`
...	...

The following query uses a lateral derived table to generate a result set with two columns: one that lists the ID for each manager, and one that lists the ID for each employee that reports to that manager:

```
SELECT ManagerID, EmployeeID
FROM test, LATERAL( openxml( test.Reports, '//e' )
WITH (EmployeeID INT '.') ) DerivedTable;
ORDER BY ManagerID, EmployeeID;
```

This query generates the following result:

ManagerID	**EmployeeID**
501	102
501	105
501	160
501	243
...	...

☞ For more information about lateral derived tables, see “FROM clause” [*SQL Anywhere Server - SQL Reference*].

Importing XML using the DataSet object

The ADO.NET DataSet object allows you to read the data and/or schema from an XML document into a DataSet.

- The ReadXml method populates a DataSet from an XML document that contains both a schema and data.
- The ReadXmlSchema method reads only the schema from an XML document. Once the DataSet is filled with data from the XML document, you can update the tables in your database with the changes from the DataSet.

DataSet objects can also be manipulated using the SQL Anywhere ADO.NET data provider.

☞ For information about using a DataSet to read the data and/or schema from an XML document using the SQL Anywhere .NET data provider, see "Getting data using the SADataAdapter object" [*SQL Anywhere Server - Programming*].

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 604 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 622.

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:

```
SELECT ID, Name
FROM Products
WHERE Color='black'
FOR XML AUTO
```

The following query uses the XMLELEMENT function to generate XML:

```
SELECT XMLELEMENT(NAME product,
          XMLATTRIBUTES(ID, Name))
FROM Products
WHERE Color='black'
```

Both queries generate the following XML (the result set has been formatted to make it easier to read):

```
<product ID="302" Name="Tee Shirt"/>
<product ID="400" Name="Baseball Cap"/>
<product ID="501" Name="Visor"/>
<product ID="700" Name="Shorts"/>
```

Tip

If you are generating deeply-nested documents, a FOR XML EXPLICIT query will likely be more efficient than a SQL/XML query because EXPLICIT mode queries normally use a UNION to generate nesting, while SQL/XML uses subqueries to generate the required nesting.

Using the FOR XML clause to retrieve query results as XML

SQL Anywhere allows you to execute a SQL query against your database and return the results as an XML document by using the FOR XML clause in your SELECT statement. The XML document is of type XML.

☞ For information about the XML data type, see "XML data type" [*SQL Anywhere Server - SQL Reference*].

The FOR XML clause can be used in any SELECT statement, including subqueries, queries with a GROUP BY clause or aggregate functions, and view definitions.

☞ For examples of how the FOR XML clause can be used, see "FOR XML examples" on page 607.

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 608.

♦ **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 610.

♦ **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 612.

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.

```
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*] and "Options dialog: Results tab" [*SQL Anywhere 10 - Context-Sensitive Help*].

FOR XML examples

The following examples show how the FOR XML clause can be used in a SELECT statement.

- The following example shows how the FOR XML clause can be used in a subquery:

  ```
  SELECT XMLELEMENT(
     NAME root,
        (SELECT * FROM Employees
         FOR XML RAW));
  ```

- The following example shows how the FOR XML clause can be used in a query with a GROUP BY clause and aggregate function:

```
SELECT Name, AVG(UnitPrice) AS Price
FROM Products
GROUP BY Name
FOR XML RAW;
```

♦ The following example shows how the FOR XML clause can be used in a view definition:

```
CREATE VIEW EmployeesDepartments
AS SELECT Surname, GivenName, DepartmentName
FROM Employees JOIN Departments
ON Employees.DepartmentID = Departments.DepartmentID
FOR XML AUTO;
```

Using FOR XML RAW

When you specify FOR XML RAW in a query, each row is represented as a <row> element, and each column is an attribute of the <row> element.

Syntax

FOR XML RAW[, **ELEMENTS**]

Parameters

ELEMENTS tells FOR XML RAW to generate an XML element, instead of an attribute, for each column in the result. If there are NULL values, the element is omitted from the generated XML document. The following query generates <EmployeeID> and <DepartmentName> elements:

```
SELECT Employees.EmployeeID, Departments.DepartmentName
FROM Employees JOIN Departments
   ON Employees.DepartmentID=Departments.DepartmentID
FOR XML RAW, ELEMENTS
```

This query gives the following result:

```
<row>
   <EmployeeID>102</EmployeeID>
   <DepartmentName>R &amp; 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 606.

FOR XML RAW does not return a well-formed XML document because the document does not have a single root node. If a <root> element is required, one way to insert one is to use the XMLELEMENT function. For example,

```
SELECT XMLELEMENT( NAME root,
                   (SELECT EmployeeID AS id, GivenName AS name
                   FROM Employees FOR XML RAW))
```

☞ For more information about the XMLELEMENT function, see "XMLELEMENT function [String]" [*SQL Anywhere Server - SQL Reference*].

The attribute or element names used in the XML document can be changed by specifying aliases. The following query renames the ID attribute to product_ID:

```
SELECT ID AS product_ID
FROM Products
WHERE Color='black'
FOR XML RAW
```

This query gives the following result:

```
<row product_ID="302"/>
<row product_ID="400"/>
<row product_ID="501"/>
<row product_ID="700"/>
```

The order of the results depend on the plan chosen by the optimizer, unless you request otherwise. If you want the results to appear in a particular order, you must include an ORDER BY clause in the query, for example:

```
SELECT Employees.EmployeeID, Departments.DepartmentName
FROM Employees JOIN Departments
   ON Employees.DepartmentID=Departments.DepartmentID
ORDER BY EmployeeID
FOR XML RAW
```

Example

Suppose you want to retrieve information about which department an employee belongs to, as follows:

```
SELECT Employees.EmployeeID, Departments.DepartmentName
FROM Employees JOIN Departments
   ON Employees.DepartmentID=Departments.DepartmentID
FOR XML RAW
```

The following XML document is returned:

```
<row EmployeeID="102" DepartmentName="R &amp; 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

FOR XML AUTO[, **ELEMENTS**]

Parameters

ELEMENTS tells FOR XML AUTO to generate an XML element, instead of an attribute, for each column in the result. For example,

```
SELECT Employees.EmployeeID, Departments.DepartmentName
FROM Employees JOIN Departments
   ON Employees.DepartmentID=Departments.DepartmentID
ORDER BY EmployeeID
FOR XML AUTO, ELEMENTS;
```

In this case, each column in the result set is returned as a separate element, rather than as an attribute of the <Employees> element. If there are NULL values, the element is omitted from the generated XML document.

```
<Employees>
   <EmployeeID>102</EmployeeID>
   <Departments>
      <DepartmentName>R &amp; D</DepartmentName>
   </Departments>
</Employees>

<Employees>
   <EmployeeID>105</EmployeeID>
   <Departments>
      <DepartmentName>R &amp; D</DepartmentName>
   </Departments>
</Employees>

<Employees>
   <EmployeeID>129</EmployeeID>
   <Departments>
   <DepartmentName>Sales</DepartmentName>
   </Departments>
</Employees>
...
```

Usage

When you execute a query using FOR XML AUTO, data in BINARY, LONG BINARY, IMAGE, and VARBINARY columns is automatically returned in base64-encoded format. By default, NULL values are omitted from the result. You can return NULL values as empty attributes by setting the for_xml_null_treatment option to EMPTY.

☞ For information about setting the for_xml_null_treatment option, see "for_xml_null_treatment option [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 desire.

FOR XML AUTO does not return a well-formed XML document because the document does not have a single root node. If a <root> element is required, one way to insert one is to use the XMLELEMENT function. For example,

```
SELECT XMLELEMENT( NAME root,
                   (SELECT EmployeeID AS id, GivenName AS name
                   FROM Employees FOR XML AUTO ) );
```

☞ For more information about the XMLELEMENT function, see "XMLELEMENT function [String]" [*SQL Anywhere Server - SQL Reference*].

You can change the attribute or element names used in the XML document by specifying aliases. The following query renames the ID attribute to product_ID:

```
SELECT ID AS product_ID
FROM Products
WHERE Color='Black'
FOR XML AUTO;
```

The following XML is generated:

```
<Products product_ID="302"/>
<Products product_ID="400"/>
<Products product_ID="501"/>
<Products product_ID="700"/>
```

You can also rename the table with an alias. The following query renames the table to product_info:

```
SELECT ID AS product_ID
FROM Products AS product_info
WHERE Color='Black'
FOR XML AUTO;
```

The following XML is generated:

```
<product_info product_ID="302"/>
<product_info product_ID="400"/>
<product_info product_ID="501"/>
<product_info product_ID="700"/>
```

Example

The following query generates XML that contains both <employee> and <department> elements, and the <employee> element (the table listed first in the select list) is the parent of the <department> element.

```
SELECT EmployeeID, DepartmentName
FROM Employees AS employee JOIN Departments AS department
   ON employee.DepartmentID=department.DepartmentID
ORDER BY EmployeeID
FOR XML AUTO;
```

The following XML is generated by the above query:

```
<employee EmployeeID="102">
   <department DepartmentName="R &amp; D"/>
```

```
</employee>
<employee EmployeeID="105">
   <department DepartmentName="R &amp; D"/>
</employee>


<employee EmployeeID="129">
   <department DepartmentName="Sales;"/>
</employee>
<employee EmployeeID="148">
   <department DepartmentName="Finance;"/>
</employee>
...
```

If you change the order of the columns in the select list as follows:

```
SELECT DepartmentName, EmployeeID
FROM Employees AS employee JOIN Departments AS department
   ON employee.DepartmentID=department.DepartmentID
ORDER BY 1, 2
FOR XML AUTO;
```

The result is nested as follows:

```
<department DepartmentName="Finance">
   <employee EmployeeID="148"/>
   <employee EmployeeID="390"/>
   <employee EmployeeID="586"/>
   ...
</department>

<Department name="Marketing">
   <employee EmployeeID="184"/>
   <employee EmployeeID="207"/>
   <employee EmployeeID="318"/>
   ...
</department>
...
```

Again, the XML generated for the query contains both <employee> and <department> elements, but in this case the <department> element is the parent of the <employee> element.

Using FOR XML EXPLICIT

FOR XML EXPLICIT allows you to control the structure of the XML document returned by the query. The query must be written in a particular way so that information about the nesting you desire 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 614.

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 613.)

Tag	Parent	GivenName!1	ID!2
1	NULL	'Beth'	NULL
2	NULL	NULL	'102'

In this example, the values in the Tag column are the tag numbers for each element in the result set. The Parent column for both rows contains the value NULL. This means that both elements are generated at the top level of the hierarchy, giving the following result when the query includes the FOR XML EXPLICIT clause:

```
<GivenName>Beth</GivenName>
<ID>102</ID>
```

However, if the second row had the value 1 in the Parent column, the result would look as follows:

```
<GivenName>Beth
   <ID>102</ID>
</GivenName>
```

☞ For an example of how to write a query using FOR XML EXPLICIT, see "Writing an EXPLICIT mode query" on page 614.

Adding data columns to the query

In addition to the Tag and Parent columns, the query must also contain one or more data columns. The names of these data columns control how the columns are interpreted during tagging. Each column name is split into fields separated by an exclamation mark (!). The following fields can be specified for data columns:

ElementName!*TagNumber*!*AttributeName*!*Directive*

ElementName the name of the element. For a given row, the name of the element generated for the row is taken from the *ElementName* field of the first column with a matching tag number. If there are multiple columns with the same *TagNumber*, the *ElementName* is ignored for subsequent columns with the same *TagNumber*. In the example above, the first row generates an element called <GivenName>.

TagNumber the tag number of the element. For a row with a given tag value, all columns with the same value in their *TagNumber* field will contribute content to the element that corresponds to that row.

AttributeName specifies that the column value is an attribute of the *ElementName* element. For example, if a data column had the name productID!1!Color, then Color would appear as an attribute of the <productID> element.

Directive this optional field allows you to control the format of the XML document further. You can specify any one of the following values for *Directive*:

♦ **hide** indicates that this column is ignored 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 619.

♦ **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 618.

♦ **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 620.

♦ **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 621.

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 606.

Writing an EXPLICIT mode query

Suppose you want to write a query using FOR XML EXPLICIT that generates the following XML document:

```
<employee EmployeeID='129'>
   <customer CustomerID='107' Region='Eastern'/>
   <customer CustomerID='119' Region='Western'/>
   <customer CustomerID='131' Region='Eastern'/>
</employee>
```

```
<employee EmployeeID='195'>
   <customer CustomerID='109' Region='Eastern'/>
   <customer CustomerID='121' Region='Central'/>
</employee>
```

You do this by writing a SELECT statement that returns the following result set in the exact order specified, and then appending FOR XML EXPLICIT to the query.

Tag	Parent	employee!1!EmployeeID	customer!2!CustomerID	customer!2!Region
1	NULL	129	NULL	NULL
2	1	129	107	Eastern
2	1	129	119	Western
2	1	129	131	Central
1	NULL	195	NULL	NULL
2	1	195	109	Eastern
2	1	195	121	Central

When you write your query, only some of the columns for a given row become part of the generated XML document. A column is included in the XML document only if the value in the *TagNumber* field (the second field in the column name) matches the value in the Tag column.

In the example, the third column is used for the two rows that have the value 1 in their Tag column. In the fourth and fifth columns, the values are used for the rows that have the value 2 in their Tag column. The element names are taken from the first field in the column name. In this case, <employee> and <customer> elements are created.

The attribute names come from the third field in the column name, so an EmployeeID attribute is created for <employee> elements, while CustomerID and Region attributes are generated for <customer> elements.

The following steps explain how to construct the FOR XML EXPLICIT query that generates an XML document similar to the one found at the beginning of this section using the SQL Anywhere sample database.

♦ To write a FOR XML EXPLICIT query

1. Write a SELECT statement to generate the top-level elements.

   In this example, the first SELECT statement in the query generates the <employee> elements. The first two values in the query must be the Tag and Parent column values. The <employee> element is at the top of the hierarchy, so it is assigned a Tag value of 1, and a Parent value of NULL.

Note
If you are writing an EXPLICIT mode query that uses a UNION, then only the column names specified in the first SELECT statement are used. Column names that are to be used as element or attribute names must be specified in the first SELECT statement because column names specified in subsequent SELECT statements are ignored.

To generate the <employee> elements for the table above, your first SELECT statement is as follows:

```
SELECT
      1          AS tag,
      NULL       AS parent,
      EmployeeID AS [employee!1!EmployeeID],
      NULL       AS [customer!2!CustomerID],
      NULL       AS [customer!2!Region]
FROM Employees;
```

2. Write a SELECT statement to generate the child elements.

   The second query generates the <customer> elements. Because this is an EXPLICIT mode query, the first two values specified in all the SELECT statements must be the Tag and Parent values. The <customer> element is given the tag number 2, and because it is a child of the <employee> element, it has a Parent value of 1. The first SELECT statement has already specified that EmployeeID, CustomerID, and Region are attributes.

```
SELECT
      2,
      1,
      EmployeeID,
      CustomerID,
      Region
FROM Employees KEY JOIN SalesOrders
```

3. Add a UNION ALL to the query to combine the two SELECT statements together:

```
SELECT
      1          AS tag,
      NULL       AS parent,
      EmployeeID AS [employee!1!EmployeeID],
      NULL       AS [customer!2!CustomerID],
      NULL       AS [customer!2!Region]
FROM Employees
UNION ALL;

SELECT
      2,
      1,
      EmployeeID,
      CustomerID,
      Region
FROM Employees KEY JOIN SalesOrders
```

4. Add an ORDER BY clause to specify the order of the rows in the result. The order of the rows is the order that is used in the resulting document.

```
SELECT
      1          AS tag,
      NULL       AS parent,
      EmployeeID AS [employee!1!EmployeeID],
      NULL       AS [customer!2!CustomerID],
            NULL        AS [customer!2!Region]
      FROM Employees
      UNION ALL

      SELECT
            2,
            1,
            EmployeeID,
            CustomerID,
            Region
      FROM Employees KEY JOIN SalesOrders
      ORDER BY 3, 1
      FOR XML EXPLICIT;
```

☞ For information about the syntax of EXPLICIT mode, see "Parameters" on page 613.

FOR XML EXPLICIT examples

The following example query retrieves information about the orders placed by employees. In this example, there are three types of elements: <employee>, <order>, and <department>. The <employee> element has ID and name attributes, the <order> element has a date attribute, and the <department> element has a name attribute.

```
SELECT
        1           tag,
        NULL        parent,
        EmployeeID  [employee!1!ID],
        GivenName   [employee!1!name],
        NULL        [order!2!date],
        NULL        [department!3!name]
FROM Employees
UNION ALL;

SELECT
        2,
        1,
        EmployeeID,
        NULL,
        OrderDate,
        NULL
FROM Employees KEY JOIN SalesOrders
UNION ALL;

SELECT
        3,
        1,
        EmployeeID,
        NULL,
        NULL,
        DepartmentName
FROM Employees e JOIN Departments d
   ON e.DepartmentID=d.DepartmentID
ORDER BY 3, 1
FOR XML EXPLICIT;
```

You get the following result from this query:

```
<employee ID="102" name="Fran">
   <department name="R &amp; D"/>
</employee>
<employee ID="105" name="Matthew">
   <department name="R &amp; D"/>
```

```
</employee>
<employee ID="129" name="Philip">
   <order date="2000-07-24"/>
   <order date="2000-07-13"/>
   <order date="2000-06-24"/>
   <order date="2000-06-08"/>
   ...
   <department name="Sales"/>
</employee>
<employee ID="148" name="Julie">
   <department name="Finance"/>
</employee>
...
```

Using the element directive

If you want to generate sub-elements rather than attributes, you can add the **element** directive to the query, as follows:

```
SELECT
        1          tag,
        NULL       parent,
        EmployeeID [employee!1!id!element],
        GivenName  [employee!1!name!element],
        NULL       [order!2!date!element],
        NULL       [department!3!name!element]
FROM Employees;

UNION ALL
SELECT
        2,
        1,
        EmployeeID,
        NULL,
        OrderDate,
        NULL
FROM Employees KEY JOIN SalesOrders
UNION ALL;

SELECT
        3,
        1,
        EmployeeID,
        NULL,
        NULL,
        DepartmentName
FROM Employees e JOIN Departments d
   ON e.DepartmentID=d.DepartmentID
ORDER BY 3, 1
FOR XML EXPLICIT;
```

You get the following result from this query:

```
<employee>
   <id>102</id>
   <name>Fran</name>
   <department>
      <name>R &amp; 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],
        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 606) 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 & Co.</CompanyName>
</customer>
```

Note that this XML is not well-formed because it contains an ampersand, which is a special character in XML. When XML is generated by a query, it is your responsibility to ensure that the XML is well-formed and valid: SQL Anywhere does not check whether the XML being generated is well-formed or valid.

When you specify the **xml** directive, the *AttributeName* field is ignored, and elements are generated rather than attributes.

Using the cdata directive

The following query uses the **cdata** directive to return the customer name in a CDATA section:

```
SELECT
        1               AS tag,
        NULL            AS parent,
        ID              AS [product!1!ID],
        Description     AS [product!1!!cdata]
FROM Products
FOR XML EXPLICIT;
```

The result produced by this query lists the description for each product in a CDATA section. Data contained in the CDATA section is not quoted:

```
<product ID="300">
   <![CDATA[Tank Top]]>
</product>
<product ID="301">
   <![CDATA[V-neck]]>
</product>

<product ID="302">
   <![CDATA[Crew Neck]]>
</product>
<product ID="400">
   <![CDATA[Cotton Cap]]>
</product>
...
```

Using SQL/XML to obtain query results as XML

SQL/XML is a draft standard that describes a functional integration of XML into the SQL language: it describes the ways that SQL can be used in conjunction with XML. The supported functions allow you to write queries that construct XML documents from relational data.

Invalid names and SQL/XML

In SQL/XML, expressions that are not legal XML names, for example expressions that include spaces, are escaped in the same manner as the FOR XML clause. Element content of type XML is not quoted.

☞ For more information about quoting invalid expressions, see "Encoding illegal XML names" on page 606.

☞ For information about using the XML data type, see "Storing XML documents in relational databases" on page 595.

Using the XMLAGG function

The XMLAGG function is used to produce a forest of XML elements from a collection of XML elements. XMLAGG is an aggregate function, and produces a single aggregated XML result for all the rows in the query.

In the following query, XMLAGG is used to generate a <name> element for each row, and the <name> elements are ordered by employee name. The ORDER BY clause is specified to order the XML elements:

```
SELECT XMLELEMENT( NAME Departments,
                   XMLATTRIBUTES (DepartmentID ),
                   XMLAGG( XMLELEMENT( NAME name,
                             Surname )
                             ORDER BY Surname )
                  ) AS department_list
FROM Employees
GROUP BY DepartmentID
ORDER BY DepartmentID;
```

This query produces the following result:

department_list
`<Departments DepartmentID="100"> <name>Breault</name> <name>Cobb</name> <name>Diaz</name> <name>Driscoll</name> ... </Departments>`

department_list
`<Departments DepartmentID="200"> <name>Chao</name> <name>Chin</name> <name>Clark</name> <name>Dill</name> ... </Departments>`
`<Departments DepartmentID="300"> <name>Bigelow</name> <name>Coe</name> <name>Coleman</name> <name>Davidson</name> ... </Departments>`
...

☞ For more information about the XMLAGG function, see "XMLAGG function [Aggregate]" [*SQL Anywhere Server - SQL Reference*].

Using the XMLCONCAT function

The XMLCONCAT function creates a forest of XML elements by concatenating all the XML values passed in. For example, the following query concatenates the <given_name> and <surname> elements for each employee in the Employees table:

```
SELECT XMLCONCAT( XMLELEMENT( NAME given_name, GivenName ),
                  XMLELEMENT( NAME surname, Surname )
                ) AS "Employee_Name"
FROM Employees;
```

This query returns the following result:

Employee_Name
`<given_name>Fran</given_name> <surname>Whitney</surname>`
`<given_name>Matthew</given_name> <surname>Cobb</surname>`
`<given_name>Philip</given_name> <surname>Chin</surname>`
`<given_name>Julie</given_name> <surname>Jordan</surname>`
...

☞ For more information, see "XMLCONCAT function [String]" [*SQL Anywhere Server - SQL Reference*].

Using the XMLELEMENT function

The XMLELEMENT function constructs an XML element from relational data. You can specify the content of the generated element and if you want, you can also specify attributes and attribute content for the element.

Generating nested elements

The following query generates nested XML, producing a <product_info> element for each product, with elements that provide the name, quantity, and description of each product:

```
SELECT ID,
XMLELEMENT( NAME product_info,
            XMLELEMENT( NAME item_name, Products.name ),
            XMLELEMENT( NAME quantity_left, Products.Quantity ),
            XMLELEMENT( NAME description, Products.Size || ' ' ||
                        Products.Color || ' ' || Products.name )
          ) AS results
FROM Products
WHERE Quantity > 30;
```

This query produces the following result:

ID	results
301	`<product_info>` `<item_name>Tee Shirt` `</item_name>` `<quantity_left>54` `</quantity_left>` `<description>Medium Orange` `Tee Shirt</description>` `</product_info>`
302	`<product_info>` `<item_name>Tee Shirt` `</item_name>` `<quantity_left>75` `</quantity_left>` `<description>One Size fits` `all Black Tee Shirt` `</description>` `</product_info>`
400	`<product_info>` `<item_name>Baseball Cap` `</item_name>` `<quantity_left>112` `</quantity_left>` `<description>One Size fits` `all Black Baseball Cap` `</description>` `</product_info>`
...	...

Specifying element content

The XMLELEMENT function allows you to specify the content of an element. The following statement produces an XML element with the content **hat**.

```
SELECT ID, XMLELEMENT( NAME product_type, 'hat' )
FROM Products
WHERE Name IN ( 'Baseball Cap', 'Visor' );
```

Generating elements with attributes

You can add attributes to the elements by including the XMLATTRIBUTES argument in your query. This argument specifies the attribute name and content. The following statement produces an attribute for the name, Color, and UnitPrice of each item.

```
SELECT ID, XMLELEMENT( NAME item_description,
                       XMLATTRIBUTES( Name,
                                      Color,
                                      UnitPrice )
                      ) AS item_description_element
FROM Products
WHERE ID > 400;
```

Attributes can be named by specifying the AS clause:

```
SELECT ID, XMLELEMENT( NAME item_description,
                       XMLATTRIBUTES ( UnitPrice AS

                                     price ),
                       Products.name
                     ) AS products
FROM Products
WHERE ID > 400;
```

☞ For more information, see “XMLELEMENT function [String]” [*SQL Anywhere Server - SQL Reference*].

Example

The following example uses XMLELEMENT with an HTTP web service.

```
ALTER PROCEDURE "DBA"."http_header_example_with_table_proc"()
RESULT ( res LONG VARCHAR )
BEGIN
   DECLARE var LONG VARCHAR;
   DECLARE varval LONG VARCHAR;
   DECLARE i INT;
 DECLARE res LONG VARCHAR;
 DECLARE tabl XML;
   SET var  = NULL;
loop_h:
   LOOP
      SET var = NEXT_HTTP_HEADER( var );
      IF var IS NULL THEN LEAVE loop_h END IF;
      SET varval = http_header( var );
      -- ... do some action for <var,varval> pair...
        SET tabl = tabl ||
                        XMLELEMENT( name "tr",
                        XMLATTRIBUTES( 'left' AS "align", 'top' AS
"valign" ),
                        XMLELEMENT( name "td", var ),
                        XMLELEMENT( name "td", varval ) )  ;

   END LOOP;

    SET res = XMLELEMENT( NAME "table",
                XMLATTRIBUTES( '' AS "BORDER", '10' as "CELLPADDING", '0' AS
"CELLSPACING" ),

                XMLELEMENT( NAME "th",
                    XMLATTRIBUTES( 'left' AS "align", 'top' AS "valign" ),
                    'Header Name' ),

                XMLELEMENT( NAME "th",
                    XMLATTRIBUTES( 'left' AS "align", 'top' AS "valign" ),
                    'Header Value' ),

                 tabl ) ;
    SELECT res;
END
```

Using the XMLFOREST function

XMLFOREST constructs a forest of XML elements. An element is produced for each XMLFOREST argument.

The following query produces an <item_description> element, with <name>, <color>, and <price> elements:

```
SELECT ID, XMLELEMENT( NAME item_description,
                       XMLFOREST( Name as name,
                                  Color as color,
                                  UnitPrice AS price )
                      ) AS product_info
FROM Products
WHERE ID > 400;
```

The following result is generated by this query:

ID	product_info

401	`<item_description>`
	` <name>Baseball Cap</name>`
	` <color>White</color>`
	` <price>10.00</price>`
	`</item_description>`

500	`<item_description>`
	` <name>Visor</name>`
	` <color>White</color>`
	` <price>7.00</price>`
	`</item_description>`

501	`<item_description>`
	` <name>Visor</name>`
	` <color>Black</color>`
	` <price>7.00</price>`
	`</item_description>`

...	...

☞ For more information, see "XMLFOREST function [String]" [*SQL Anywhere Server - SQL Reference*].

Using the XMLGEN function

The XMLGEN function is used to generate an XML value based on an XQuery constructor.

The XML generated by the following query provides information about customer orders in the SQL Anywhere sample database. It uses the following variable references:

- **{$ID}** Generates content for the <ID> element using values from the ID column in the SalesOrders table.
- **{$OrderDate}** Generates content for the <date> element using values from the OrderDate column in the SalesOrders table.
- **{$Customers}** Generates content for the <customer> element from the CompanyName column in the Customers table.

```
SELECT XMLGEN ( '<order>
              <ID>{$ID}</ID>
              <date>{$OrderDate}</date>
              <customer>{$Customers}</customer>
              </order>',
              SalesOrders.ID,
              SalesOrders.OrderDate,
              Customers.CompanyName AS Customers
              ) AS order_info
FROM SalesOrders JOIN Customers
ON Customers.ID = SalesOrders.CustomerID
ORDER BY SalesOrders.CustomerID;
```

This query generates the following result:

order_info
`<order> <ID>2001</ID> <date>2000-03-16</date> <customer>The Power Group</customer> </order>`
`<order> <ID>2005</ID> <date>2001-03-26</date> <customer>The Power Group</customer> </order>`
`<order> <ID>2125</ID> <date>2001-06-24</date> <customer>The Power Group</customer> </order>`
`<order> <ID>2206</ID> <date>2000-04-16</date> <customer>The Power Group</customer> </order>`

order_info
...

Generating attributes

If you want the order ID number to appear as an attribute of the <order> element, you would write query as follows (note that the variable reference is contained in double quotes because it specifies an attribute value):

```
SELECT XMLGEN ( '<order ID="{$ID}">
                  <date>{$OrderDate}</date>
                  <customer>{$Customers}</customer>
                  </order>',
                 SalesOrders.ID,
                 SalesOrders.OrderDate,
                 Customers.CompanyName AS Customers
               ) AS order_info
FROM SalesOrders JOIN Customers
ON Customers.ID = SalesOrders.CustomerID
ORDER BY SalesOrders.OrderDate;
```

This query generates the following result:

order_info

`<order ID="2131">`
` <date>2000-01-02</date>`
` <customer>BoSox Club</customer>`
`</order>`

`<order ID="2065">`
` <date>2000-01-03</date>`
` <customer>Bloomfield&apos;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&apos;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 622.

Specifying header information for XML documents

The FOR XML clause and the SQL/XML functions supported by SQL Anywhere do not include version declaration information in the XML documents they generate. You can use the XMLGEN function to generate header information.

```
SELECT XMLGEN( '<?xml version="1.0"
                encoding="ISO-8859-1" ?>
                <r>{$x}</r>',
                (SELECT GivenName, Surname
      FROM Customers FOR XML RAW) AS x );
```

This produces the following result:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<r>
 <row GivenName="Michaels" Surname="Devlin"/>
 <row GivenName="Beth" Surname="Reiser"/>
 <row GivenName="Erin" Surname="Niedringhaus"/>
 <row GivenName="Meghan" Surname="Mason"/>
 ...
</r>
```

☞ For more information about the XMLGEN function, see “XMLGEN function [String]” [*SQL Anywhere Server - SQL Reference*].

Part VI. 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

About this chapter

This chapter describes the process of transferring bulk data into and out of your SQL Anywhere database.

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 often grouped together as 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 to the database server: server-side

The database server executes LOAD TABLE, UNLOAD TABLE, and UNLOAD statements. Paths to the files being written or read are relative to the database server. A LOAD TABLE statement is recorded in the transaction log as a single command. Internal importing and exporting only provides access to text and BCP formats, but it is a faster method.

External to the database server: client-side

Interactive SQL executes INPUT and OUTPUT commands. Paths to files being read or written are relative to Interactive SQL and the computer on which it is running, not to the server. An INPUT is recorded in the transaction log as a separate INSERT statement for each row read. As a result, INPUT is considerably slower than LOAD TABLE. This also means that INSERT triggers will fire during an INPUT. The OUTPUT statement is useful when compatibility is an issue since it can write out the result set of a SELECT statement to any one of a number of file formats.

☞ For more information, see “Performance tips for importing data” on page 636.

Data recovery issues for bulk operations

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.

If you are running the database server in bulk operations mode (the -b option), the database server does not perform certain important functions. Specifically:

Function	Implication
Maintain a transaction log	There is no record of the changes. Each COMMIT causes a checkpoint.
Lock any records	There are no serious implications.

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 ASCII files
- import entire tables or portions of tables from other database formats, such as DBASE or EXCEL
- 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
- 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 655.

Import considerations

Before you begin importing data into your database, consider in detail what resources you have and exactly what you want to accomplish.

Consideration...	**For information, see...**
The import tools that are available.	“Import tools” on page 637
Impact on database performance that importing might have.	“Performance aspects of bulk operations” on page 634 “Performance tips for importing data” on page 636
Compatibility issues between your data and the destination table.	“Table structures for import” on page 643

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, as described in "ALTER DBSPACE statement" [*SQL Anywhere Server - SQL Reference*]. 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.
- 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 using the LOAD TABLE statement, do not start the database server with the -b option (bulk operations mode); it is not necessary.
- 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.

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 in text, DBASEII, DBASEIII, Excel 2.1, FOXPRO, and Lotus formats. 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.

Use the Interactive SQL Import wizard when you:

- want to create a table at the same time you import the data
- prefer using a graphical interface to import data in a format other than text

Examples

This example shows you how to import data using the Interactive SQL Data menu.

♦ To import data (Interactive SQL Data menu)

1. From the Data menu, choose Import.

   The Open dialog appears.

2. Locate the file you want to import, and then click Open.

   The Import wizard appears.

3. Specify how the database values are stored in the file that you are importing.

4. Select the Use An Existing Table option and then enter the name and location of the existing table. Click Next.

   You can also click Browse and locate the table you want to import the data into.

5. For ASCII files, you can specify the following:

   - the field delimiter for the file
   - the escape character
   - whether trailing blanks should be included
   - the encoding for the file

   The encoding option allows you to specify the code page that is used to read the file to ensure that characters are treated correctly. If you select (Default), the default encoding for the system that Interactive SQL is running on is used.

   ☞ For more information, see "Recommended character sets and collations" [*SQL Anywhere Server - Database Administration*].

6. Follow the remaining instructions in the wizard.

   Importing adds the new data to the existing table. If the import is successful, the Messages tab displays the amount of time it to took to import the data. If the import is unsuccessful, a message appears indicating that the import was unsuccessful.

The following example shows you how to add data to the Employees table of the SQL Anywhere sample database. This example adds information about an employee named Julia Chan.

♦ To import data into the SQL Anywhere sample database

1. Create and save a text file named *import.txt* with the following values (on a single line):

```
100,500,'Chan','Julia',100,'300 Royal Drive',
'Springfield','OR','USA','97015','6175553985',
'A','017239033',55700,'1984-09-29',,'1968-05-05',
1,1,0,'F'
```

2. In Interactive SQL, choose Import from the Data menu.

3. Locate the *import.txt* file and then click Open.

   The Import wizard appears.

4. Select the Separated by commas or some other delimited option.

5. Select the Use An Existing Table option and then enter **Employees** as the name of the destination table. Click Next.

6. Follow the remaining steps, accepting the defaults, in the Import wizard.

Using the INPUT statement to import data

The INPUT statement lets you import data in a variety of file formats into one or more tables. You can choose a default input format, or you can specify the file format on each INPUT statement. Interactive SQL can execute a command file containing multiple INPUT statements. 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 into one or more tables
- ♦ automate the import process using a command file
- ♦ import data in a format other than text

☞ For more information, see "INPUT statement [Interactive SQL]" [*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.

The INPUT statement is slower than the LOAD TABLE statement.

If a data file is in DBASE, DBASEII, DBASEIII, FOXPRO, or LOTUS format and the table does not exist, it is created.

Example

This example shows you how to import data using the Interactive SQL INPUT statement.

♦ To import data (Interactive SQL 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. Ensure that the destination table exists.

3. Enter an INPUT statement in the SQL Statements pane of Interactive SQL.

   The following is an example of an INPUT statement from an ASCII text file:

```
INPUT INTO Employees
FROM c:\new_employees.txt
FORMAT ASCII;
SELECT * FROM Employees;
```

In this statement, Employees is the name of the destination table, and *new_employees.txt* is the name of the source file.

4. Execute the statement.

   If the import is successful, the Messages tab displays the amount of time it to took to import the data. If the import is unsuccessful, a message appears indicating the import was unsuccessful.

☞ For information about using the INPUT statement to import data, see "INPUT statement [Interactive SQL]" [*SQL Anywhere Server - SQL Reference*].

Using the LOAD TABLE statement to import data

The LOAD TABLE statement lets you efficiently import data into an existing table in text/ASCII formats. The LOAD TABLE statement imports one row per line, with values separated by a delimiter.

Use the LOAD TABLE statement when you want to:

- import data in text format
- improve performance and can use the LOAD TABLE statement instead of an INPUT statement

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

Impact on the database

Changes are not recorded in the transaction log when the LOAD TABLE statement is used, which makes this statement unusable with SQL Remote or MobiLink remote databases. If you need to recover data using the current transaction log and an older copy of the database, recovery fails if the original data file is no longer present.

The LOAD TABLE statement is considerably faster than the INPUT statement.

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.

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

☞ For more information, see "INSERT 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.

☞ For more information, see "The transaction log" [*SQL Anywhere Server - Database Administration*].

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.

   ```
   INSERT
   INTO Departments ( DepartmentID, DepartmentName, DepartmentHeadID )
   VALUES ( 700, 'Training', 501)
   SELECT * FROM Departments;
   ```

   Inserting values adds the new data to the existing table.

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 673.

☞ 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. The table data can be in text, DBASEII, Excel 2.1, FOXPRO, and Lotus formats.

♦ To import a table (Interactive SQL Data menu)

1. Ensure that the table you want to place the data in exists.
2. From the Data menu, choose Import.

   The Open dialog appears.
3. Locate the file you want to import and click Open. The file extension must be one of those recognized by the Import wizard

   The Import wizard appears.
4. Select the Use an Existing Table option and enter the name of the table in the field.
5. Follow the remaining instructions in the wizard.

   If the import is successful, the Messages tab displays the amount of time it to took to import the data and the number of rows inserted. If the import is unsuccessful, a message appears indicating that the import was unsuccessful. The Results tab in the Results pane displays the contents of the table.

♦ To import a table (Interactive SQL Statements pane)

1. Use the CREATE TABLE statement to create the destination table. For example:

```
CREATE TABLE GROUPO.Departments
(
DepartmentID           integer NOT NULL,
DepartmentName         char(40) NOT NULL,
DepartmentHeadID       integer NULL,
CONSTRAINT DepartmentsKey PRIMARY KEY (DepartmentID)
);
```

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

2. Execute a LOAD TABLE statement. For example,

```
LOAD TABLE Departments
FROM 'departments.csv'
```

3. To keep trailing blanks in your values, use the STRIP OFF clause in your LOAD TABLE statement. The default setting (STRIP ON) strips trailing blanks from values before inserting them.

   The LOAD TABLE statement adds the contents of the file to the existing rows of the table; it does not replace the existing rows in the table. You can use the TRUNCATE TABLE statement to remove all the rows from a table.

   Neither the TRUNCATE TABLE statement nor the LOAD TABLE statement fires triggers, including referential integrity actions, such as cascaded deletes.

☞ For more information about the LOAD TABLE statement syntax, see "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 Employees table and then load new rows into the Employees table using a LOAD TABLE statement like this:

```
ALTER TABLE Employees
ALTER City DEFAULT 'Waterloo';
LOAD TABLE Employees (Surname, GivenName, EmployeeID, DepartmentID,
StartDate)
FROM 'new_employees.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.

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

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.

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 467.

Exporting data from databases

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 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.

☞ For more information about unloading and reloading complete databases, see "Rebuilding databases" on page 655.

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.

Consideration...	For information, see...
The export tools that are available.	"Export tools" on page 645
Impact of exporting on the database performance.	"Performance aspects of bulk operations" on page 634
How to handle NULL values.	"Outputting NULLs using the OUTPUT statement" on page 652

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. In Interactive SQL, from the Data menu, choose Export. This allows you to choose the file format of the exported query results.

Example

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 the following query while connected to the SQL Anywhere sample database.

   ```
   SELECT * FROM Employees
   WHERE State = 'GA';
   ```

   The result set includes a list of all employees who live in Georgia.

2. In Interactive SQL, choose Export from the Data menu.

   The Export dialog appears.

3. Specify a name and location for the exported data.

4. Specify the file format and then click OK.

   The result set is exported to a file in the location that you specified.

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 dialog (accessed by choosing Tools ► Options in Interactive SQL).

Use the Interactive SQL OUTPUT statement when you want to:

- export all or part of a table or view in a format other than text
- automate the export process using a command file

Impact on the database

If you have a choice between using the OUTPUT statement, UNLOAD statement, or UNLOAD TABLE statement, choose the UNLOAD TABLE statement for performance reasons.

There are performance impacts associated with exporting large amounts of data with the OUTPUT statement. Use the OUTPUT statement on the same computer as the server if possible to avoid sending large amounts of data across the network.

☞ For more information, see “OUTPUT statement [Interactive SQL]” [*SQL Anywhere Server - SQL Reference*].

Example

The following example exports the data from the Employees table in the SQL Anywhere sample database to a *.txt* file named *Employees.txt*.

```
SELECT *
FROM Employees;
OUTPUT TO Employees.txt
FORMAT ASCII;
```

Using the UNLOAD TABLE statement

The UNLOAD TABLE statement lets you export data efficiently in text/ASCII 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

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.

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

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.

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/ASCII 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

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.

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

Example

Using the SQL Anywhere sample database, you can unload a subset of the Employees table to a text file named *employee_data.csv* by executing the following command:

```
UNLOAD
SELECT * FROM Employees
WHERE State = 'GA'
TO 'employee_data.csv';
```

Because it is the database server that unloads the result set, *employee_data.csv* specifies a file on the database server computer.

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*].

Unload tools

You can use the following tools to help you unload data from your database:

- "Using the dbunload utility to export data" on page 648
- "Using the Unload Database wizard" on page 649
- "Using the Unload Data dialog" on page 649

Using the Unload Database wizard

You can use the Sybase Central Unload Database wizard to unload an existing database into a new database.

You can also use this wizard to unload an entire database in ASCII 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.

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.

☞ For information about special considerations when unloading a database, see "The Unload utility" [*SQL Anywhere Server - Database Administration*].

Using the Unload Data dialog

You can use the Unload Data dialog 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 dialog allows you to unload tables in one step, instead of completing the entire Unload Database wizard.

♦ To unload tables using the Unload Data dialog

1. Connect to your database in Sybase Central.
2. From the Context dropdown list, choose your database.
3. Open the Tables folder.

4. Select the table you want to export data from.

   You can select multiple tables by pressing Ctrl while you click the tables.

5. From the File menu, choose Unload Data.

   The Unload Data dialog appears.

6. Select the desired options and then click OK to unload the data.

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 671.

♦ To export query results (Interactive SQL Data menu)

1. Enter your query in the SQL Statements pane of Interactive SQL.
2. Click Execute SQL statement(s) to display the result set.
3. From the Data menu, choose Export.

   The Save As dialog appears.

4. Specify a name and location for the exported data.
5. Specify the file format, and then click OK.

   If the export is successful, the Messages tab displays the amount of time it to took to export the query result set, the file name and path of the exported data, and the number of rows written.

   If the export is unsuccessful, a message appears indicating that the export was unsuccessful.

♦ 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 '***file-name***'**.

   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:

To do this...	Use this clause...	Example...
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;`
Specify a file format other than ASCII (the default)	FORMAT	`SELECT * FROM Employees; OUTPUT TO 'employees.dbf' FORMAT DBASEIII;` Thus, *employees.dbf* is the name and extension of the new file and DBASEIII is the file format. You can enclose the string in single or double quotation marks, but they are only required if the file path or name contain embedded spaces. If you leave the FORMAT option out, the file type defaults to ASCII.

4. Execute the statement by choosing Execute from the SQL menu.

   If the export is successful, the Messages tab displays the amount of time it to took to export the query result set, the file name and path of the exported data, and the number of rows written. If the export is unsuccessful, a message appears indicating that the export was unsuccessful.

   ☞ For more information about exporting query results using the OUTPUT statement, see "OUTPUT statement [Interactive SQL]" [*SQL Anywhere Server - SQL Reference*].

Tips

You can combine the APPEND and VERBOSE clauses to append both results and messages to an existing file.

For example, type **OUTPUT TO '***file-name***' 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 the "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. Execute the UNLOAD statement by choosing Execute from the SQL menu.

If the export is successful, the Messages tab displays the amount of time it to took to export the query result set, the file name and path of the exported data, and the number of rows written. If the export is unsuccessful, a message appears indicating that the export was unsuccessful.

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 to the desired value.

  The following example changes the value that appears for NULL values to (unknown):

  ```
  SET OPTION output_nulls = '(unknown)';
  ```

☞ For more information about setting Interactive SQL options, see "SET OPTION statement" [*SQL Anywhere Server - SQL Reference*].

Tip
You can change the value that appears in place of a NULL value on the Interactive SQL Results pane. From the Tools menu, choose Options. On the Results tab of the Interactive SQL Options dialog, configure the Display Null Values As value.

Exporting databases

♦ To unload all or part of a database (Sybase Central)

1. From the Tools menu, choose SQL Anywhere 10 ▶ Unload Database.

   The Unload Database wizard appears.

2. Follow the instructions in the wizard.

♦ To unload all or part of a database (Command line)

1. At a command prompt, enter the dbunload command and specify connection parameters using the -c option.

   For example, the following command unloads 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.

2. Do either of the following:

   ♦ 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

Tip
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.
For more information, see "Exporting query results" on page 650.

Tip
You can export views just as you would export tables.

♦ To export a table (Command line)

- At a command prompt, enter the following dbunload command, and then press Enter:

  ```
  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.

☞ For more information about the UNLOAD TABLE statement syntax, see “UNLOAD 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 10 - 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.

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 ASCII 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.

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 663.

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** If you want to use SQL Anywhere 10, you must unload and reload your database. You should back up your database after rebuilding the database.

  ☞ For more information about upgrading your database, see "Upgrading SQL Anywhere" [*SQL Anywhere 10 - Changes and Upgrading*].

- **Reclaim disk space** Databases do not shrink if you delete data. Instead, any empty pages are simply marked as free so they can be used again. They are not removed from the database unless you rebuild it. Rebuilding a database can reclaim disk space if you have deleted a large amount of data from your database and do not anticipate adding more.

- **Improve database performance** Rebuilding databases can improve performance. Since the database can be unloaded and reloaded in order by primary keys, access to related information can be faster as related rows may appear on the same or adjacent pages.

Note
If you detect that performance is poor because a table is highly fragmented, you can reorganize the table. See "REORGANIZE TABLE statement" [*SQL Anywhere Server - SQL Reference*].

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.

Consideration...	For information, see...
What is involved in rebuilding a database	"Rebuilding databases" on page 655 "Reasons to rebuild databases" on page 656
The available tools for rebuilding	"Rebuild and reload tools" on page 659
The impact that rebuilding has on the database users	"Minimizing downtime during rebuilding" on page 661

Consideration...	For information, see...
Whether or not the database is involved in synchronization or replication	"Rebuilding databases involved in synchronization or replication" on page 658 "Rebuilding databases not involved in synchronization or replication" on page 657
Changing the collation sequence of the database	"Changing a database from one collation to another" [*SQL Anywhere Server - Database Administration*]

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. At a command prompt, execute the dbunload utility using one of the following options:

To do this...	Use this option...	Example
Rebuild to a new database	-an	`dbunload -c "DBF=demo.db;UID=DBA;PWD=sql" -an DemoBackup.db`
Reload to an existing database	-ac	`dbunload -c "DBF=demo.db;UID=DBA;PWD=sql" -ac "UID=DBA;PWD=sql;DBF=NewDemo.db"`
Replace an existing database	-ar	`dbunload -c "DBF=demo.db;UID=DBA;PWD=sql" -ar`

   If you use one of these options, no interim copy of the data is created on disk, so you do not need to specify an unload directory on the command line. This provides greater security for your data. The -ar and -an options should also execute more quickly than the Unload Database wizard in Sybase Central, but -ac is slower than the Unload Database wizard.

2. Shut down the database and archive the transaction log before using the reloaded database.

Notes

The -an and -ar options only apply to connections to a personal server, or connections to a network server over shared memory.

There are additional options available for the dbunload utility that allow you to tune the unload, 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. At a command prompt, run dbunload 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 "The Unload utility" [*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 "The Unload utility" [*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 on this step, see "Rebuilding databases" on page 655.
6. Shut down the new database.
7. Erase the current transaction log file for the new database.
8. Use dblog on the new database with the ending offset noted in Step 3 as the -z parameter, and also set the relative offset to zero.

   ```
   dblog -x 0 -z 137829 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 ASCII 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 Sybase Central Unload Database wizard are functionally equivalent. You can use them interchangeably to produce the same results.

Use the dbunload utility when you:

- ♦ want to rebuild your database or extract data from your database
- ♦ want to export in text format
- ♦ need to process large amounts of data quickly
- ♦ have flexible file format requirements

☞ For more information, see:

- ♦ "Rebuilding databases not involved in synchronization or replication" on page 657
- ♦ "Rebuilding databases involved in synchronization or replication" on page 658

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 TABLE 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)

- At a command prompt, enter the dbunload command and specify connection parameters using the -c option, specify the table(s) you want to export data for using the -t option, and specify that you want to unload only data by specifying the -d option.

  For example, to export the data from the Employees table, execute 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)

- At a command prompt, enter the dbunload command and specify connection parameters using the -c option, specify the table(s) you want to export data for using the -t option, and specify that 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. Open a command prompt.
2. Create a new, empty database file.

   For example, the following command creates a file named *newdemo.db*.

   ```
   dbinit newdemo.db
   ```
3. 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 the following:

- "Database Extraction utility" [*SQL Remote*]
- "Using the extraction utility" [*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 stored 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 stored 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" on page 31.

Migrating Access databases
You can migrate Microsoft Access databases to SQL Anywhere using the SQL Anywhere for MS Access Migration utility (upsize tool). You can download this utility at http://www.ianywhere.com/developer/code_samples/sqlany_migration_utility.html.

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. From Sybase Central, connect to the target database.
2. Choose Tools ▶ SQL Anywhere 10 ▶ Migrate Database.

   The Migrate Database wizard appears.
3. Select the target database from the list, and then click Next.
4. Select the remote server you want to use to connect to the remote database, and then click Next.

   If you have not already created a remote server, click Create Remote Server Now to access the Create Remote Server wizard.

   ☞ For more information about creating a remote server, see "Creating remote servers" on page 677.

   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.
5. 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.

6. Select the user that will own the tables on the target database, and then click Next.

   If you have not already created a user, click Create User Now to open the User Creation wizard.

   ☞ For more information, see "Creating new users" [*SQL Anywhere Server - Database Administration*].

7. 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.

8. Click Finish.

Using the sa_migrate stored procedures

The sa_migrate stored 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 stored 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" on page 31.

2. Create a remote server to connect to the remote database.

   ☞ For more information, see "Creating remote servers" on page 677.

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 685.

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 Interactive SQL Statements pane, run the sa_migrate stored procedure. For example,

   ```
   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 stored 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 stored 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 stored 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 stored 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 stored 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 stored 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 stored 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**.

Opening SQL command files in Interactive SQL

On 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 "Options dialog: General tab" [*SQL Anywhere 10 - Context-Sensitive Help*].

Executing SQL command files

You can execute command files in any of the following ways:

♦ You can use the Interactive SQL READ statement to execute command files. For example, the following statement executes the file *temp.sql*:

```
READ temp.sql;
```

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

♦ You can load a command file into the SQL Statements pane and execute it directly from there.

  You load command files into the SQL Statements pane by choosing File ► Open. Enter *temp.sql* when prompted for the file name.

♦ You can supply a command file as a command line argument for Interactive SQL.

Running SQL command files

You can run command files interactively or in batch mode.

♦ To run a command file immediately

1. In Interactive SQL, from the File menu, choose Run Script.

   The Open dialog appears.

2. In the Open dialog, locate the file, and then click Open.

The Run Script menu item is the equivalent of a READ statement. See below for an example of the READ statement.

♦ To run a command file using the SQL Statements pane

- Type the following command:

  ```
  READ 'c:\\filename.sql';
  ```

  In this statement, *c:\filename.sql* is the path, name, and extension of the file. Single quotation marks (as shown) are required only if the path contains spaces.

♦ To run a command file in batch mode

1. Open a command prompt.
2. Supply a command file as a command line argument for Interactive SQL.

   For example, the following command runs the command file *myscript.sql* against the SQL Anywhere sample database.

   ```
   dbisql -c "DSN=SQL Anywhere 10 Demo" myscript.sql
   ```

Loading SQL command files

When you begin a new Interactive SQL session, you can load the contents of a command file into the SQL Statements pane, or you can run the contents immediately.

♦ To load commands from a file into the SQL Statements pane

1. From the File menu, choose Open.

   The Open dialog appears.
2. In the Open dialog, find and select the file.
3. Click Open to load the command file.

Writing database output to a file

In Interactive SQL, the result set data for each command remains on the Results tab in the Results pane only until the next command is executed. To keep a record of your data, you can save the output of each statement to a separate file. If *statement1* and *statement2* are two SELECT statements, then you can output them to *file1* and *file2*, respectively, as follows:

statement1; OUTPUT TO *file1*;
statement2; OUTPUT TO *file2*;

For example, the following command saves the result of a query to a file named *Employees.txt*:

```
SELECT * FROM EMPLOYEE;
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 ASCII 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 TABLE statement" [*SQL Anywhere Server - SQL Reference*].

CHAPTER 18

Accessing Remote Data

Contents

About this chapter

SQL Anywhere can access data located on different servers, both Sybase and non-Sybase, as if the data were stored on the local server.

This chapter describes how to configure SQL Anywhere to access remote data.

Introduction

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 the SQL Anywhere table in SQL Anywhere 10.0.0 Components by Platform.

Basic concepts to access remote data

This section describes the basic concepts required to access remote data.

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 677.
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 685.
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 687.

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 directly.

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.
- **aseodbc** for Adaptive Server Enterprise and SQL Server (version 10 and later).
- **db2odbc** for IBM DB2
- **mssodbc** for Microsoft SQL Server
- **oraodbc** for Oracle servers (version 8.0 and later)
- **odbc** for all other ODBC data sources

Note
The JDBC classes have a significant performance impact and should only be used in situations where the ODBC classes cannot be used, for example, on NetWare.

The JDBC-based server classes are:

- **sajdbc** for SQL Anywhere.
- **asejdbc** for Adaptive Server Enterprise and SQL Server (version 10 and later).

☞ For a full description of remote server classes, see "Server Classes for Remote Data Access" on page 705.

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 dialog 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. This section describes how to create, alter, and delete a remote server definition.

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 machine 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 10;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 from Sybase Central.
2. Open the Remote Servers folder for that database.
3. From the File menu, choose New ► Remote Server.

   The Create Remote Server wizard appears.
4. On the first page of the wizard, enter a name to use for the remote server. This name refers to the remote server from within the local database; it does not need to correspond with the name the server supplies. Click Next.
5. Select the appropriate type of server, and then click Next.
6. Select a data access method (ODBC or JDBC), and supply connection information.

   - For ODBC, supply a data source name or specify the ODBC Driver= parameter.

- For JDBC, supply a URL in the form *machine-name*:*port-number*.
- Click Next.

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. Specify whether you want the remote server to be read-only. Click Next.
8. 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. See "CREATE EXTERNLOGIN statement" [*SQL Anywhere Server - SQL Reference*].
9. Optionally, you can click Test Connection to test whether you can connect to the remote server before the remote server definition is created.

   Click Finish to create the remote server definition.

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.

You must have DBA authority to delete a remote server.

♦ To delete a remote server (Sybase Central)

1. Connect to the host database from Sybase Central.
2. Open the Remote Servers folder.
3. Select the remote server you want to delete, and then choose File ► Delete.

♦ To delete a remote server (SQL)

1. Connect to the host database from Interactive SQL.
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;
```

Altering remote servers

You can use Sybase Central or an ALTER SERVER statement to modify the attributes of a server. These changes do not take effect until the next connection to the remote server.

You must have RESOURCE authority to alter a server.

♦ To alter the properties of a remote server (Sybase Central)

1. Connect to the host database from Sybase Central.
2. Open the Remote Servers folder for that database.
3. Select the remote server, and then choose File ► Properties.
4. Configure the various remote server properties, and then click OK.

♦ To alter the properties of a remote server (SQL)

1. Connect to the host database from Interactive SQL.
2. Execute an ALTER SERVER statement.

Example

The following statement changes the server class of the server named RemoteASE to aseodbc. In this example, the Data Source Name for the server is RemoteASE.

```
ALTER SERVER RemoteASE
CLASS 'aseodbc';
```

The ALTER SERVER statement can also be used to enable or disable a server's known capabilities. See “ALTER SERVER statement” [*SQL Anywhere Server - SQL Reference*].

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.

```
sp_remote_tables(
@server-name
[, @table-name
[, @table-owner
[, @table-qualifier
[, @with-table-type ] ] ] ]
)
```

If *table-name* or *table-owner*,is specified, 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:

```
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:

```
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 off to a remote server.

The system tables which contain server capabilities are not populated until after SQL Anywhere first connects to the remote server. This information comes from the ISYSCAPABILITY and ISYSCAPABILITYNAME system tables. 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.
2. Open the Directory Access Servers folder.
3. From the File menu, choose New ► Directory Access Server.

   The Create Directory Access Server wizard appears.
4. Follow the instructions in the wizard.

♦ To create and configure a directory access server (SQL)

1. Create a remote server using the CREATE SERVER statement.

   For example:

   ```
   CREATE SERVER my_dir_tree
   CLASS 'directory'
   USING 'root=c:\Program Files';
   ```
2. Create an external login using the CREATE EXTERNLOGIN statement.

   For example:

```
CREATE EXTERNLOGIN DBA TO my_dir_tree;
```

3. 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 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:\Program Files 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:\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. Open the Directory Access Servers folder for the database.
2. Select the directory access server, and then choose Edit ▶ Delete.

♦ To drop a directory access server (SQL)

- 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. Open the Directory Access Servers folder for the database.
2. In the right pane, click the Proxy Tables tab.
3. Select the proxy table, and then choose Edit ▶ Delete.

♦ To drop a proxy table (SQL)

- 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

You can create an external login using either Sybase Central or the CREATE EXTERNLOGIN statement.

Only the DBA or user who will be using the external login can add or modify an external login.

♦ To create an external login (Sybase Central)

1. Connect to the host database from Sybase Central.
2. Open the Remote Servers folder for that database and then select the remote server.
3. Click the External Logins tab.
4. From the File menu, choose New ► External Login.

   The Create External Login wizard appears.
5. Follow the instructions in the wizard.

♦ To create an external login (SQL)

1. Connect to the host database from Interactive SQL.
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

You can use either Sybase Central or a DROP EXTERNLOGIN statement to delete an external login from the SQL Anywhere system tables.

Only the DBA or user who will be using the external login can delete an external login.

♦ To delete an external login (Sybase Central)

1. Connect to the host database from Sybase Central.
2. Open the Remote Servers folder.
3. In the left pane, select the remote server and then click the External Logins tab in the right pane.
4. Select the external login, and then choose File ► Delete.

♦ To delete an external login (SQL)

1. Connect to the host database from Interactive SQL.
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 115.

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)

You can create a proxy table using either Sybase Central or a CREATE EXISTING TABLE statement. 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 from Sybase Central.
2. In the left pane, open the Remote Servers folder.
3. In the right pane, click the Proxy Tables tab.
4. From the File menu choose New ► Proxy Table.
5. Follow the instructions in the wizard.

Tip
Proxy tables are displayed in the right pane on the Proxy Tables tab when their remote server is selected in the left pane. They also appear in the Tables folder.

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)

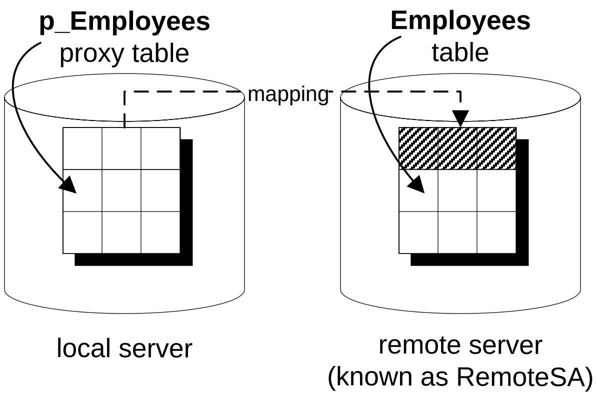
1. Connect to the host database.
2. Execute a CREATE EXISTING TABLE statement.

☞ For more information, see the "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 the "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.

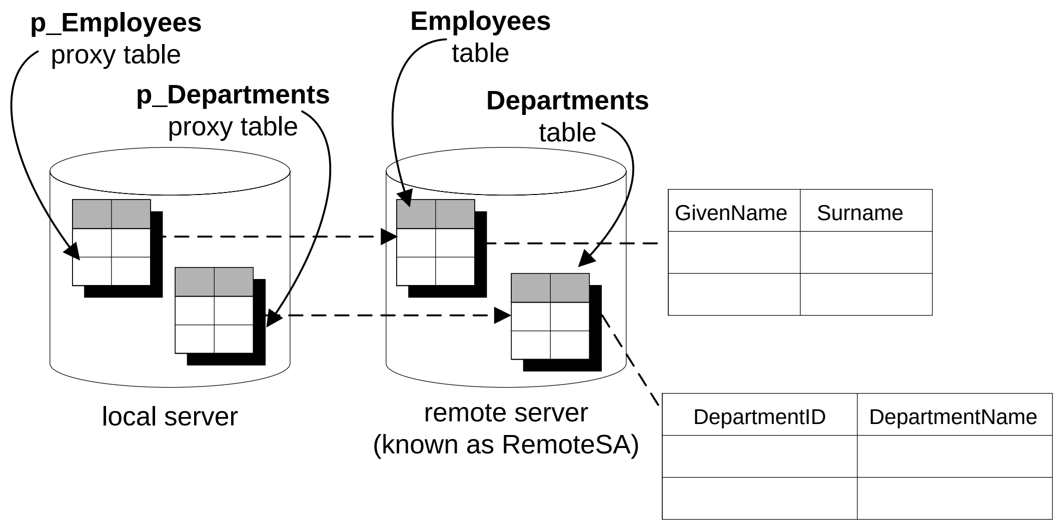
For example, the following returns a list of the columns in the sysobjects table in the production database on an Adaptive Server Enterprise server named asetest:

```
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:

   ```
   dbeng10 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 10 Demo:

   ```
   CREATE SERVER RemoteSA
   CLASS 'saodbc'
   USING 'SQL Anywhere 10 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 value in the CREATE SERVER statement" on page 707.

Example

Suppose you are using database db1, and you want to access data in tables in database db2. You need to set up proxy table definitions that point to the tables in database db2. For example, on a SQL Anywhere server named RemoteSA, you might have three databases available, db1, db2, and db3.

1. If you are using ODBC, create an ODBC data source name for each database you will be accessing.
2. Connect to one of the databases from which you will be performing. For example, connect to db1.
3. Perform a CREATE SERVER statement for each other local database you will be accessing. This sets up a **loopback** connection to your SQL Anywhere server.

   ```
   CREATE SERVER remote_db2
   CLASS 'saodbc'
   USING 'RemoteSA_db2';
   CREATE SERVER remote_db3
   CLASS 'saodbc'
   USING 'RemoteSA_db3';
   ```

   Alternatively, using JDBC:

   ```
   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.

   ```
   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

You can issue a remote procedure call using either Sybase Central or the CREATE PROCEDURE statement.

You must have DBA authority to create a remote procedure.

♦ To issue a remote procedure call (Sybase Central)

1. Connect to the host database from Sybase Central.
2. Open the Remote Servers folder.
3. In the left pane, select the remote server for which you want to create a remote procedure.
4. In the right pane, click the Remote Procedures tab.
5. From the File menu, choose New ► Remote Procedure.

   The Create Remote Procedure wizard appears.
6. Follow the instructions in the wizard.

♦ To issue a remote procedure call (SQL)

1. 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.

   ```
   CREATE PROCEDURE remotewho()
   AT 'bostonase.master.dbo.sp_who';
   ```
2. Execute the procedure as follows:

   ```
   CALL remotewho();
   ```

☞ For more information, see “CREATE PROCEDURE statement” [*SQL Anywhere Server - SQL Reference*].

Example

Here is an example that specifies a parameter when calling a remote procedure:

```
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

You can delete a remote procedure using either Sybase Central or the DROP PROCEDURE statement.

You must have DBA authority to delete a remote procedure.

♦ To delete a remote procedure (Sybase Central)

1. Open the Remote Servers folder.
2. In the left pane, select the remote server.
3. In the right pane, click the Remote Procedures tab.
4. On the Remote Procedures tab, select the remote procedure, and then choose File ► Delete.

♦ To delete a remote procedure (SQL)

1. Connect to a database.
2. Execute a DROP PROCEDURE statement.

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

Example

Delete a remote procedure called remoteproc.

```
DROP PROCEDURE remoteproc;
```

Transaction management and remote data

Transactions provide a way to group SQL statements so that they are treated as a unit—either all work performed by the statements is committed to the database, or none of it is.

For the most part, transaction management with remote tables is the same as transaction management for local tables in SQL Anywhere, but there are some differences. They are discussed in the following section.

☞ For a general discussion of transactions, see "Using Transactions and Isolation Levels" on page 111.

Remote transaction management overview

The method for managing transactions involving remote servers uses a two-phase commit protocol. SQL Anywhere implements a strategy that ensures transaction integrity for most scenarios. However, when more than one remote server is invoked in a transaction, there is still a chance that a distributed unit of work will be left in an undetermined state. Even though two-phase commit protocol is used, no recovery process is included.

The general logic for managing a user transaction is as follows:

1. SQL Anywhere prefaces work to a remote server with a BEGIN TRANSACTION notification.
2. When the transaction is ready to be committed, SQL Anywhere sends a PREPARE TRANSACTION notification to each remote server that has been part of the transaction. This ensures that the remote server is ready to commit the transaction.
3. If a PREPARE TRANSACTION request fails, all remote servers are instructed to roll back the current transaction.

   If all PREPARE TRANSACTION requests are successful, the server sends a COMMIT TRANSACTION request to each remote server involved with the transaction.

Any statement preceded by BEGIN TRANSACTION can begin a transaction. Other statements are sent to a remote server to be executed as a single, remote unit of work.

Restrictions on transaction management

Restrictions on transaction management are as follows:

♦ Savepoints are not propagated to remote servers.

♦ If nested BEGIN TRANSACTION and COMMIT TRANSACTION statements are included in a transaction that involves remote servers, only the outermost set of statements is processed. The innermost set, containing the BEGIN TRANSACTION and COMMIT TRANSACTION statements, is not transmitted to remote servers.

Internal operations

This section describes the underlying steps that SQL Anywhere performs on remote servers on behalf of client applications.

Query parsing

When a statement is received from a client, the database server parses it. The database server raises an error if the statement is not a valid SQL Anywhere SQL statement.

Query normalization

Referenced objects in the query are verified and some data type compatibility is checked.

For example, consider the following query:

```
SELECT *
FROM t1
WHERE c1 = 10;
```

The query normalization stage verifies that table t1 with a column c1 exists in the system tables. It also verifies that the data type of column c1 is compatible with the value 10. If the column's data type is 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, let's say a remote server can not process the ATAN2 function in the following statement:

```
SELECT a,b,c
WHERE ATAN2( b, 10 ) > 3
AND c = 10;
```

The statement sent to the remote server would be converted to:

```
SELECT a,b,c WHERE c = 10;
```

Then, SQL Anywhere locally applies `WHERE ATAN2( b, 10 ) > 3` to the intermediate result set.

Joins

When two tables are joined, one table is selected to be the outer table. The outer table is scanned based on the WHERE conditions that apply to it. For every qualifying row found, the other table, known as the inner table is scanned to find a row that matches the join condition.

This same algorithm is used when remote tables are referenced. Since the cost of searching a remote table is usually much higher than a local table (due to network I/O), every effort is made to make the remote table the outermost table in the join.

UPDATE and DELETE

When a qualifying row is found, if SQL Anywhere cannot pass off an UPDATE or DELETE statement entirely to a remote server, it must change the statement into a table scan containing as much of the original WHERE clause as possible, followed by a positioned UPDATE or DELETE statement that specifies WHERE CURRENT OF *cursor-name*.

For example, when the function ATAN2 is not supported by a remote server:

```
UPDATE t1
SET a = atan2( b, 10 )
WHERE b > 5;
```

Would be converted to the following:

```
SELECT a,b
FROM t1
WHERE  b > 5;
```

Each time a row is found, SQL Anywhere would calculate the new value of a and issue:

```
UPDATE t1
SET a = 'new value'
WHERE CURRENT OF CURSOR;
```

If a already has a value that equals the new value, a positioned UPDATE would not be necessary, and would not be sent remotely.

To process an UPDATE or DELETE statement that requires a table scan, the remote data source must support the ability to perform a positioned UPDATE or DELETE (WHERE CURRENT OF *cursor-name*). Some data sources do not support this capability.

Temporary tables cannot be updated
An UPDATE or DELETE cannot be performed if an intermediate temporary table is required. This occurs in queries with ORDER BY and some queries with subqueries.

Troubleshooting remote data access

This section provides some hints for troubleshooting access to remote servers.

Features not supported for remote data

The following SQL Anywhere features are not supported on remote data:

- ALTER TABLE statement against remote tables.
- Triggers defined on proxy tables.
- SQL Remote.
- Foreign keys that refer to remote tables.
- READTEXT, WRITETEXT, and TEXTPTR functions.
- Positioned UPDATE and DELETE statements.
- UPDATE and DELETE statements requiring an intermediate temporary table.
- Backward scrolling on cursors opened against remote data. Fetch statements must be NEXT or RELATIVE 1.
- If a column on a remote table has a name that is a keyword on the remote server, you cannot access data in that column. You can execute a CREATE EXISTING TABLE statement, and import the definition but you cannot select that column.

Case sensitivity

The case sensitivity setting of your SQL Anywhere database should match the settings used by any remote servers accessed.

SQL Anywhere databases are created case insensitive by default. With this configuration, unpredictable results may occur when selecting from a case-sensitive database. Different results will occur depending on whether ORDER BY or string comparisons are pushed off to a remote server, or evaluated by the local SQL Anywhere server.

Connectivity tests

Take the following steps to be sure you can connect to a remote server:

- Determine that you can connect to a remote server using a client tool such as Interactive SQL before configuring SQL Anywhere.
- Perform a simple passthrough statement to a remote server to check your connectivity and remote login configuration. For example:

```
FORWARD TO RemoteSA {SELECT @@version};
```

- Turn on remote tracing for a trace of the interactions with remote servers. For example:

  ```
  SET OPTION cis_option = 7;
  ```

  Once you have turned on remote tracing, the tracing information appears in the 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 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

If you access multiple databases on a single SQL Anywhere server, you may need to increase the number of threads used by the database server on Windows using the -gx option.

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 130.

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

About this chapter

This chapter describes how SQL Anywhere interfaces with different classes of servers. It describes:

- the types of servers that each server class supports
- the USING clause value for the CREATE SERVER statement for each server class
- special configuration requirements

Overview

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:

- ♦ JDBC-based server classes
- ♦ ODBC-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.

When using this chapter, refer both to the section generic to the server class category (JDBC-based or ODBC-based), and to the section specific to the individual server class.

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** Adaptive Server Enterprise and SQL Server (version 10 and later).

If you are using NetWare, only the sajdbc class is supported.

Configuration notes for JDBC classes

When you access remote servers defined with JDBC-based classes, consider that:

- For optimum performance, Sybase recommends an ODBC-based class (saodbc or aseodbc).
- Any remote server that you access using the asejdbc or sajdbc server class must be set up to handle a jConnect 5.5-based client. The jConnect setup scripts are *jcatalog.sql* for SQL Anywhere or *sql_server.sql* for Adaptive Server Enterprise. Run these against any remote server you will be using.

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 value in the CREATE SERVER statement

You must perform a separate CREATE SERVER 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 similar to this:

```
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 about the CREATE SERVER statement, see “CREATE SERVER statement” [*SQL Anywhere Server - SQL Reference*].

Server class asejdbc

A server with server class asejdbc is an Adaptive Server Enterprise or SQL Server (version 10 and later) server. No special requirements exist for the configuration of an Adaptive Server Enterprise data source.

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. The following table describes the SQL Anywhere to Adaptive Server Enterprise data type conversions.

SQL Anywhere data type	Adaptive Server Enterprise default data type
bit	bit
tinyint	tinyint
smallint	smallint
int	int
integer	integer
decimal [defaults p=30, s=6]	numeric(30,6)
decimal(128,128)	not supported
numeric [defaults p=30 s=6]	numeric(30,6)
numeric(128,128)	not supported
float	real
real	real
double	float
smallmoney	numeric(10,4)
money	numeric(19,4)
date	datetime
time	datetime
timestamp	datetime
smalldatetime	datetime
datetime	datetime
char(n)	varchar(n)

SQL Anywhere data type	Adaptive Server Enterprise default data type
character(n)	varchar(n)
varchar(n)	varchar(n)
character varying(n)	varchar(n)
long varchar	text
text	text
binary(n)	binary(n)
long binary	image
image	image
bigint	numeric(19,0)

ODBC-based server classes

The ODBC-based server classes include:

♦ saodbc
♦ aseodbc
♦ db2odbc
♦ mssodbc
♦ oraodbc
♦ odbc

Defining ODBC external servers

The most common way of defining an ODBC-based server bases it on an ODBC data source. To do this, you must create a data source in the ODBC Administrator.

Once you have the data source defined, 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 on creating data sources, see "Creating an ODBC data source" [*SQL Anywhere Server - Database Administration*].

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 10;eng=TestSA;dbn=sample;links=tcpip{}'
```

This defines a connection to a SQL Anywhere database server named TestSA and a database sample using the TCP-IP protocol.

See also

For information specific to particular ODBC server classes, see:

♦ "Server class saodbc" on page 711
♦ "Server class aseodbc" on page 711
♦ "Server class db2odbc" on page 713
♦ "Server class oraodbc" on page 715
♦ "Server class mssodbc" on page 716
♦ "Server class odbc" on page 718

Server class saodbc

A server with server class saodbc is a SQL Anywhere database server. No special requirements exist for the configuration of a SQL Anywhere data source.

To access SQL Anywhere database servers that support multiple databases, create an ODBC data source name defining a connection to each database. Issue a CREATE SERVER statement for each of these ODBC data source names.

Server class aseodbc

A server with server class aseodbc is an Adaptive Server Enterprise or SQL Server (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. However, the performance is better than with the asejdbc class.

Notes

- Open Client should be version 11.1.1, EBF 7886 or above. 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 above.
- Configure a User Data Source in the Configuration Manager with the following attributes:
  - **General tab** Enter 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.
  - **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 Prepare Method to **2-Full**.

    Set Fetch Array Size as large as possible for best performance. This increases memory requirements since this is the number of rows that must be cached in memory. Sybase recommends using a value of 100.

    Set Select Method to **0-Cursor**.

    Set Packet Size to as large as possible. Sybase recommends using a value of -1.

    Set Connection Cache to 1.

Data type conversions: ODBC and Adaptive Server Enterprise

When you issue a CREATE TABLE statement, SQL Anywhere automatically converts the data types to the corresponding Adaptive Server Enterprise data types. The following table describes the SQL Anywhere to Adaptive Server Enterprise data type conversions.

SQL Anywhere data type	Adaptive Server Enterprise default data type
Bit	bit
Tinyint	tinyint
Smallint	smallint
Int	int
Integer	integer
decimal [defaults p=30, s=6]	numeric(30,6)
decimal(128,128)	not supported
numeric [defaults p=30 s=6]	numeric(30,6)
numeric(128,128)	not supported
Float	real
Real	real
Double	float
Smallmoney	numeric(10,4)
Money	numeric(19,4)
Date	datetime
Time	datetime
Timestamp	datetime
Smalldatetime	datetime
Datetime	datetime
char(n)	varchar(n)
Character(n)	varchar(n)
varchar(n)	varchar(n)
Character varying(n)	varchar(n)

SQL Anywhere data type	Adaptive Server Enterprise default data type
long varchar	text
Text	text
binary(n)	binary(n)
long binary	image
Image	image
Bigint	numeric(20,0)

Server class db2odbc

A server with server class db2odbc is IBM DB2.

Notes

- Sybase certifies the use of IBM's DB2 Connect version 5, with fix pack WR09044. Configure and test your ODBC configuration using the instructions for that product. SQL Anywhere has no specific requirements on 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.

SQL Anywhere data type	DB2 default data type
Bit	smallint
Tinyint	smallint
Smallint	smallint
Int	int
Integer	int
Bigint	decimal(20,0)

SQL Anywhere data type	DB2 default data type
char(1–254)	varchar(n)
char(255–4000)	varchar(n)
char(4001–32767)	long varchar
Character(1–254)	varchar(n)
Character(255–4000)	varchar(n)
Character(4001–32767)	long varchar
varchar(1–4000)	varchar(n)
varchar(4001–32767)	long varchar
Character varying(1–4000)	varchar(n)
Character varying(4001–32767)	long varchar
long varchar	long varchar
text	long varchar
binary(1–4000)	varchar for bit data
binary(4001–32767)	long varchar for bit data
long binary	long varchar for bit data
image	long varchar for bit data
decimal [defaults p=30, s=6]	decimal(30,6)
numeric [defaults p=30 s=6]	decimal(30,6)
decimal(128, 128)	NOT SUPPORTED
numeric(128, 128)	NOT SUPPORTED
real	real
float	float
double	float
smallmoney	decimal(10,4)
money	decimal(19,4)
date	date
time	time

SQL Anywhere data type	DB2 default data type
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'
  ```

- Due to Oracle ODBC driver restrictions, you cannot issue a CREATE EXISTING TABLE 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. The following table describes the SQL Anywhere to Oracle data type conversions.

SQL Anywhere data type	Oracle data type
bit	number(1,0)
tinyint	number(3,0)
smallint	number(5,0)
int	number(11,0)
bigint	number(20,0)
decimal(prec, scale)	number(prec, scale)
numeric(prec, scale)	number(prec, scale)
float	float

SQL Anywhere data type	Oracle data type
real	real
smallmoney	numeric(13,4)
money	number(19,4)
date	date
time	date
timestamp	date
smalldatetime	date
datetime	date
char(n)	if (n > 255) long else varchar(n)
varchar(n)	if (n > 2000) long else varchar(n)
long varchar	long or clob
binary(n)	if (n > 255) long raw else raw(n)
varbinary(n)	if (n > 255) long raw else raw(n)
long binary	long raw

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 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'
```

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. The following table describes the SQL Anywhere to Microsoft SQL Server data type conversions.

SQL Anywhere data type	Microsoft SQL Server default data type
bit	bit
tinyint	tinyint
smallint	smallint
int	int
bigint	numeric(20,0)
decimal [defaults p=30, s=6]	decimal(prec, scale)
numeric [defaults p=30 s=6]	numeric(prec, scale)
float	if (prec) float(prec) else float
real	real
smallmoney	smallmoney
money	money
date	datetime
time	datetime
timestamp	datetime
smalldatetime	datetime
datetime	datetime
char(n)	if (length > 255) text else varchar(length)
character(n)	char(n)
varchar(n)	if (length > 255) text else varchar(length)
long varchar	text
binary(n)	if (length > 255) image else binary(length)
long binary	image
double	float
uniqueidentifierstr	uniqueidentifier

The SQL Server uniqueidentifier columns is mapped to a SQL Anywhere uniqueidentifierstr column. The resulting string can be converted to a binary UUID value using the STRTOUUID function. See "STRTOUUID function [String]" [*SQL Anywhere Server - SQL Reference*].

Server class odbc

ODBC data sources that do not have their own server class use server class **odbc**. You can use any ODBC driver that complies with ODBC version 2.0 compliance level 1 or higher. Sybase certifies the following ODBC data sources:

- "Microsoft Excel (Microsoft 3.51.171300)" on page 718
- "Microsoft Access (Microsoft 3.51.171300)" on page 719
- "Microsoft FoxPro (Microsoft 3.51.171300)" on page 719
- "Lotus Notes SQL 2.0" on page 719

The latest versions of Microsoft ODBC drivers can be obtained through the Microsoft Data Access Components (MDAC) distribution found at http://msdn.microsoft.com/data/downloads/default.aspx. The Microsoft driver versions listed below are part of MDAC 2.0.

The following sections provide notes on accessing these data sources.

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 Access (Microsoft 3.51.171300)

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.

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. Read the documentation that is included with it for an explanation of how Notes data maps to relational tables. You can easily map SQL Anywhere tables to Notes forms.

Here is how to set up SQL Anywhere to access the Address sample file.

- Create an ODBC data source using the NotesSQL driver. The database will be the sample names file: *c:\notes\data\names.nsf*. The Map Special Characters option should be turned on. For this example, the Data Source Name is *my_notes_dsn*.
- Create a server in SQL Anywhere:

  ```
  CREATE SERVER names
  CLASS 'odbc'
  USING 'my_notes_dsn'
  ```

- Map the Person form into a SQL Anywhere table:

  ```
  CREATE EXISTING TABLE Person
  AT 'names...Person'
  ```

- Query the table

  ```
  SELECT * FROM Person
  ```

Avoiding password prompts

Lotus Notes does not support sending a user name and password through the ODBC API. If you try to access Lotus notes using a password protected ID, a window appears on the computer where SQL Anywhere is running, and prompts you for a password. Avoid this behavior in multi-user server environments.

To access Lotus Notes unattended, without ever receiving a password prompt, you must use a non-password-protected ID. You can remove password protection from your ID by clearing it (choose File ▶ Tools ▶ User ID ▶ Clear Password), unless your Domino administrator required a password when your ID was created. In this case, you will not be able to clear it.

Part VII. 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

About this chapter

Procedures and triggers store procedural SQL statements in the database for use by all applications. They enhance the security, efficiency, and standardization of databases. User-defined functions are one kind of procedures that return a value to the calling environment for use in queries and other 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.

☞ For many purposes, server-side JDBC provides a more flexible way to build logic into the database than SQL stored procedures. For information about JDBC, see "JDBC API" [*SQL Anywhere Server - Programming*].

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.

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. For more information, see “Debugging Procedures, Functions, Triggers, and Events” on page 779.

☞ You can profile stored procedures to analyze performance characteristics in Sybase Central. For more information, see “Procedure profiling using system procedures” on page 220.

Benefits of procedures and triggers

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, you can use a wizard to provide necessary information. The Create Procedure wizard also provides the option of using procedure templates.

Alternatively, you can use Interactive SQL to execute a CREATE PROCEDURE statement to create a procedure. However, you must have RESOURCE authority.

♦ To create a new procedure (Sybase Central)

1. Connect to a database as a user with DBA or Resource authority.
2. Open the Procedures & Functions folder of the database.
3. From the File menu, choose New ▶ Procedure.

   The Create Procedure wizard appears.
4. Follow the instructions in the wizard.
5. When the wizard finishes, you can complete the code of the procedure on the SQL tab in the right pane.

   The new procedure appears in the Procedures & Functions folder.

☞ For more information about connecting, see "Connecting to a Database" [*SQL Anywhere Server - Database Administration*].

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.

☞ For more information, see "CREATE PROCEDURE statement" [*SQL Anywhere Server - SQL Reference*], "ALTER PROCEDURE statement" [*SQL Anywhere Server - SQL Reference*], and "Using compound statements" on page 745.

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).

☞ For more information on altering database object properties, see "Setting properties for database objects" on page 35.

☞ For more information on granting or revoking permissions for procedures, see "Granting permissions on procedures" [*SQL Anywhere Server - Database Administration*] and "Revoking user permissions" [*SQL Anywhere Server - Database Administration*].

♦ To alter the code of a procedure (Sybase Central)

1. Open the Procedures & Functions folder.
2. Select the desired procedure. You can then do one of the following:
   - ♦ Edit the code directly on the SQL tab in the right pane.
   - ♦ Edit the code in a separate window by right-clicking the procedure and choosing Edit In New Window from the popup menu.

Tip
If you want to copy code between procedures, you can open a separate window for each procedure.

For information about translating a procedure, see "Using Sybase Central to translate stored procedures" on page 585.

☞ For more information, see "ALTER PROCEDURE statement" [*SQL Anywhere Server - SQL Reference*], "CREATE PROCEDURE statement" [*SQL Anywhere Server - SQL Reference*], and "Creating procedures" on page 726.

Calling procedures

CALL statements invoke procedures. Procedures can be called by an application program, or by other procedures and triggers.

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

The following statement calls the NewDepartment procedure to insert an Eastern Sales department:

```
CALL NewDepartment( 210, 'Eastern Sales', 902 );
```

After this call, you may want to check the Departments table to see that the new department has been added.

All users who have been granted EXECUTE permissions for the procedure can call the NewDepartment procedure, even if they have no permissions on the Departments table.

☞ For more information about EXECUTE permissions, see "GRANT statement" [*SQL Anywhere Server - SQL Reference*].

Another way of calling a procedure that returns a result set is to call it in a query. You can execute queries on result sets of procedures and apply WHERE clauses and other SELECT features to limit the result set.

```
SELECT t.ID, t.QuantityOrdered AS q
FROM ShowCustomerProducts( 149 ) t;
```

☞ For more information, see "FROM clause" [*SQL Anywhere Server - SQL Reference*].

Copying procedures in Sybase Central

In Sybase Central, you can copy procedures between databases. To do so, select the procedure in the left pane of Sybase Central and drag it to the Procedures & Functions folder of another connected database. A new procedure is then created, and the original procedure's code is copied to it.

Note that only the procedure code is copied to the new procedure. The other procedure properties (permissions, and so on) are not copied. A procedure can be copied to the same database, provided it is given 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 a database as a user with DBA authority or as the owner of the procedure.
2. Open the Procedures & Functions folder.
3. Right-click the desired procedure and choose Delete from the popup menu.

♦ 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;
```

☞ For more information, see "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.

This section describes how to return results from procedures as parameters.

The following procedure on the SQL Anywhere sample database returns the average salary of employees as an OUT parameter.

```
CREATE PROCEDURE AverageSalary( OUT avgsal NUMERIC(20,3) )
BEGIN
   SELECT AVG( Salary )
   INTO avgsal
   FROM Employees;
END;
```

♦ **To run this procedure and display its output (SQL)**

1. Using Interactive SQL, connect to the SQL Anywhere sample database as the DBA. For more information about connecting, see "Connecting to a Database" [*SQL Anywhere Server - Database Administration*].
2. In the SQL Statements pane, type the above procedure code.
3. Create a variable to hold the procedure output. In this case, the output variable is numeric, with three decimal places, so create a variable as follows:

   ```
   CREATE VARIABLE Average NUMERIC(20,3);
   ```

4. Call the procedure using the created variable to hold the result:

   ```
   CALL AverageSalary(Average);
   ```

   If the procedure was created and run properly, the Interactive SQL Messages tab does not display any errors.

5. To inspect the value of the variable, execute the following statement:

   ```
   SELECT Average;
   ```

   Look at the value of the output variable Average. The Results tab in the Results pane displays the value 49988.623 for this variable, the average employee salary.

Returning procedure results in result sets

In addition to returning results to the calling environment in individual parameters, procedures can return information in result sets. A result set is typically the result of a query. The following procedure returns a result set containing the salary for each employee in a given department:

```
CREATE PROCEDURE SalaryList( IN department_id INT )
RESULT ( "Employee ID" INT, Salary NUMERIC(20,3) )
BEGIN
     SELECT EmployeeID, Salary
     FROM Employees
     WHERE Employees.DepartmentID = department_id;
END;
```

If Interactive SQL calls this procedure, the names in the RESULT clause are matched to the results of the query and used as column headings in the displayed results.

To test this procedure from Interactive SQL, you can CALL it, specifying one of the departments of the company. In Interactive SQL, the results appear on the Results tab in the Results pane.

Example

To list the salaries of employees in the R & D department (department ID 100), type the following:

```
CALL SalaryList( 100 );
```

Employee ID	Salary
102	45700.000
105	62000.000
160	57490.000
243	72995.000
…	…

Interactive SQL can only return multiple result sets if you have this option enabled on the Results tab of the Options dialog. Each result set appears on a separate tab in the Results pane.

☞ For more information, see "Returning multiple result sets from procedures" on page 753.

Advanced information

Creating a temporary procedure

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.

☞ For more information on creating temporary procedures, see "CREATE PROCEDURE statement" [*SQL Anywhere Server - SQL Reference*].

Creating a remote procedure

In order to create a remote procedure, you must have at least one remote server. See "Creating remote servers using Sybase Central" on page 678.

♦ To create a new remote procedure (Sybase Central)

1. Connect to a database as a user with DBA authority.
2. Open the Procedures & Functions folder of the database.
3. From the File menu, choose New ► Remote Procedure.

   The Create Remote Procedure wizard appears.
4. Follow the instructions in the wizard.
5. When the wizard finishes, you can complete the code on the SQL tab in the right pane.

   The new remote procedure appears in the Procedures and Functions folder.

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.

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;
```

☞ For more information, see "CREATE FUNCTION statement" [*SQL Anywhere Server - SQL Reference*].

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.

Calling user-defined functions

A user-defined function can be used, subject to permissions, in any place you would use a built-in non-aggregate function.

The following statement in Interactive SQL returns a full name from two columns containing a first and last name:

```
SELECT FullName(GivenName, Surname)
 AS "Full Name"
 FROM Employees;
```

Full Name
Fran Whitney

Full Name
Matthew Cobb
Philip Chin
…

The following statement in Interactive SQL returns a full name from a supplied first and last name:

```
SELECT FullName('Jane', 'Smith')
 AS "Full Name";
```

Full Name
Jane Smith

Any user who has been granted EXECUTE permissions for the function can use the FullName function.

Example

The following user-defined function illustrates local declarations of variables.

The Customers table includes some Canadian customers sprinkled among those from the USA. The user-defined function Nationality forms a 3-letter country code based on the Country column.

```
CREATE FUNCTION Nationality( CustomerID INT )
RETURNS CHAR( 3 )
BEGIN
    DECLARE nation_string CHAR(3);
    DECLARE nation country_t;
    SELECT DISTINCT Country INTO nation
    FROM Customers
    WHERE ID = CustomerID;
    IF nation = 'Canada' THEN
            SET nation_string = 'CDN';
    ELSE IF nation = 'USA' OR nation = ' ' THEN
            SET nation_string = 'USA';
        ELSE
            SET nation_string = 'OTH';
        END IF;
    END IF;
RETURN ( nation_string );
END;
```

This example declares a variable nation_string to hold the nationality string, uses a SET statement to set a value for the variable, and returns the value of the nation_string string to the calling environment.

The following query lists all Canadian customers in the Customers table:

```
SELECT *
FROM Customers
WHERE Nationality(ID) = 'CDN';
```

Declarations of cursors and exceptions are discussed in later sections.

Notes

While this function is useful for illustration, it may perform very poorly if used in a SELECT involving many rows. For example, if you used the function in the SELECT list of a query on a table containing 100,000 rows, of which 10,000 are returned, the function will be called 10,000 times. If you use it in the WHERE clause of the same query, it would be called 100,000 times.

Dropping user-defined functions

Once you create a user-defined function, it remains in the database until someone explicitly removes it. Only the owner of the function or a user with DBA authority can drop a function from the database.

The following statement removes the function FullName from the database:

```
DROP FUNCTION FullName;
```

Permissions to execute user-defined functions

Ownership of a user-defined function belongs to the user who created it, and that user can execute it without permission. The owner of a user-defined function can grant permissions to other users with the GRANT EXECUTE command.

For example, the creator of the function FullName could allow another user to use FullName with the statement:

```
GRANT EXECUTE ON Nationality TO BobS;
```

The following statement revokes permissions to use the function:

```
REVOKE EXECUTE ON Nationality FROM BobS;
```

☞ For more information on managing user permissions on functions, see “Granting permissions on procedures” [*SQL Anywhere Server - Database Administration*].

Advanced information

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 524.

Introduction to triggers

A trigger is a special form of stored procedure that is executed automatically when a query that modifies data is executed. You use triggers whenever referential integrity and other declarative constraints are insufficient.

☞ For more information on referential integrity, see "Ensuring Data Integrity" on page 89 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.

Trigger execution permissions
Triggers execute with the permissions of the owner of the associated table, 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.

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* has been modified

You may write separate triggers for each event that you need to handle or, if you have some shared actions and some actions that depend on the event, you can create a trigger for all events and use an IF statement to distinguish the action taking place.

☞ For more information, 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.

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 (see "Atomic compound statements" on page 745). 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.

☞ For a full description of trigger syntax, see "CREATE TRIGGER statement" [*SQL Anywhere Server - SQL Reference*].

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.

☞ For more information, see the list of cross-references at the end of this section.

♦ To create a trigger for a given table (Sybase Central)

1. Open the Triggers folder of the desired table.
2. From the File menu, choose New ▶ Trigger.

   The Create Trigger wizard appears.
3. Follow the instructions in the wizard.
4. When the wizard finishes, you can complete the code of the trigger on the SQL tab in the right pane.

♦ To create a trigger for a given table (SQL)

1. Connect to a database.
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, you will receive the error message "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.

☞ For more information, see “CREATE TRIGGER statement” [*SQL Anywhere Server - SQL Reference*], and “Using compound statements” on page 745.

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).

☞ For more information on altering database object properties, see "Setting properties for database objects" on page 35.

♦ To alter the code of a trigger (Sybase Central)

1. Open the Triggers folder.
2. Select the desired trigger. You can then do one of the following:
   - ♦ Edit the code directly on the SQL tab in the right pane.
   - ♦ Edit the code in a separate window by right-clicking the trigger in the right pane and choose Edit In New Window from the popup menu.

   Tip
   If you want to copy code between triggers, you can open a separate window for each trigger.

   For information about translating a trigger, see "Using Sybase Central to translate stored procedures" on page 585.

♦ To alter the code of a trigger (SQL)

1. Connect to the database.
2. Execute an ALTER TRIGGER statement. Include the entire new trigger in this statement.

☞ For more information, see "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. Open the Triggers folder.
2. Right-click the desired trigger and choose Delete from the popup menu.

♦ To delete a trigger (SQL)

1. Connect to a database.
2. Execute a DROP TRIGGER statement.

Example

The following statement removes the trigger mytrigger from the database:

```
DROP TRIGGER mytrigger;
```

☞ For more information, see “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, a **table not found** message results when the trigger fires.

Also, user_1 must have permissions to perform the operations specified in the trigger.

Introduction to batches

A simple Transact-SQL 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.

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, you will get a "variable not found" error. This happens because Interactive SQL sends three separate statements to the server. They are not executed as a batch. As you have already seen, the remedy is to use a compound statement to force Interactive SQL to send these statements as a batch to the server. The following example accomplishes this.

```
BEGIN
  DECLARE @CurrentID INTEGER;
  SET @CurrentID = 207;
  SELECT Surname FROM Employees
    WHERE EmployeeID=@CurrentID;
END
```

Putting a BEGIN and END around a set of statements forces Interactive SQL to treat them as a batch.

The IF statement is another example of a compound statement. Interactive SQL sends the following statements as a single batch to the server.

```
IF EXISTS(   SELECT *
            FROM SYSTAB
            WHERE table_name='Employees' )
THEN
   SELECT   Surname AS LastName,
            GivenName AS FirstName
   FROM Employees;
   SELECT Surname, GivenName
   FROM Customers;
   SELECT Surname, GivenName
   FROM Contacts;
ELSE
   MESSAGE 'The Employees table does not exist'
   TO CLIENT;
END IF
```

This situation does not arise when using other techniques to prepare and execute SQL statements. For example, an application that uses ODBC can prepare and execute a series of semicolon-separated statements as a batch.

Care must be exercised when mixing Interactive SQL statements with SQL statements intended for the server. The following is an example of how mixing Interactve 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';
```

Many statements used in procedures and triggers can also be used in batches. You can use control statements (CASE, IF, LOOP, and so on), including compound statements (BEGIN and END), in batches. Compound statements can include declarations of variables, exceptions, temporary tables, or cursors inside the compound statement.

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:

Control statement	**Syntax**

Compound statements
 	
See "BEGIN statement" [*SQL Anywhere Server - SQL Reference*].
	`BEGIN [ ATOMIC ]`
	    Statement-list
	`END`

Conditional execution: IF
 	
See "IF statement" [*SQL Anywhere Server - SQL Reference*].
	`IF` *condition* `THEN`
	    Statement-list
	`ELSEIF` *condition* `THEN`
	    Statement-list
	`ELSE`
	    Statement-list
	`END IF`

Conditional execution: CASE
 	
See "CASE statement" [*SQL Anywhere Server - SQL Reference*].
	`CASE` *expression*
	`WHEN` *value* `THEN`
	    Statement-list
	`WHEN` *value* `THEN`
	    Statement-list
	`ELSE`
	    Statement-list
	`END CASE`

Repetition: WHILE, LOOP
 	
See "LOOP statement" [*SQL Anywhere Server - SQL Reference*].
	`WHILE` *condition* `LOOP`
	    Statement-list
	`END LOOP`

Repetition: FOR cursor loop
 	
See "FOR statement" [*SQL Anywhere Server - SQL Reference*].
	`FOR` *loop-name*
	    `AS` *cursor-name* `CURSOR FOR`
	    select-statement
	`DO`
	    Statement-list
	`END FOR`

Break: LEAVE See "LEAVE statement" [*SQL Anywhere Server - SQL Reference*].	`LEAVE` *label*
CALL See "CALL statement" [*SQL Anywhere Server - SQL Reference*].	`CALL` *procname*`(` *arg*`, ... )`

Using compound statements

A compound statement starts with the keyword BEGIN and concludes with the keyword END. The body of a procedure or trigger is a **compound statement**. Compound statements can also be used in batches. Compound statements can be nested, and combined with other control statements to define execution flow in procedures and triggers or in batches.

A compound statement allows a set of SQL statements to be grouped together and treated as a unit. Delimit SQL statements within a compound statement with semicolons.

☞ For more information about compound statements, see "BEGIN statement" [*SQL Anywhere Server - SQL Reference*].

Declarations in compound statements

Local declarations in a compound statement immediately follow the BEGIN keyword. These local declarations exist only within the compound statement. Within a compound statement you can declare:

- ♦ Variables
- ♦ Cursors
- ♦ Temporary tables
- ♦ Exceptions (error identifiers)

Local declarations can be referenced by any statement in that compound statement, or in any compound statement nested within it. Local declarations are not visible to other procedures called from the compound statement.

Atomic compound statements

An **atomic** statement is a statement 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 768).

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 763.

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 745. 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:

```
CREATE PROCEDURE CustomerProducts(
        IN customer_ID
                        INTEGER DEFAULT NULL )
RESULT ( product_ID INTEGER,
            quantity_ordered INTEGER )
BEGIN
     IF customer_ID IS NULL THEN
          RETURN;
     ELSE
          SELECT     Products.ID,
                      sum( SalesOrderItems.Quantity )
          FROM    Products,
                     SalesOrderItems,
                     SalesOrders
          WHERE SalesOrders.CustomerID = customer_ID
          AND SalesOrders.ID = SalesOrderItems.ID
          AND SalesOrderItems.ProductID = Products.ID
          GROUP BY Products.ID;
     END IF;
END;
```

The following statement assigns the DEFAULT NULL, and the procedure RETURNs instead of executing the query.

```
CALL CustomerProducts();
```

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 732 to retrieve the names of employees:

♦ To list the names of all employees

- Type the following:

  ```
  SELECT FullName(GivenName, Surname) AS Name
  FROM Employees;
  ```

Name
Fran Whitney

Name
Matthew Cobb
Philip Chin
Julie Jordan
…

Notes

- ♦ Default parameters can be used in calling functions. However, parameters cannot be passed to functions by name.
- ♦ Parameters are passed by value, not by reference. Even if the function changes the value of the parameter, this change is not returned to the calling environment.
- ♦ Output parameters cannot be used in user-defined functions.
- ♦ User-defined functions cannot return result sets.

Returning results from procedures

Procedures can return results in the form of a single row of data, or multiple rows. Results consisting of a single row of data can be passed back as arguments to the procedure. Results consisting of multiple rows of data are passed back as result sets. Procedures can also return a single value given in the RETURN statement.

For simple examples of how to return results from procedures, see “Introduction to procedures” on page 726. For more detailed information, see the following sections.

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 751.

If the query results in no rows being selected, a **row not found** warning appears.

The following procedure returns the results of a single-row SELECT statement in the procedure parameters.

♦ **To return the number of orders placed by a given customer**

- Type the following:

  ```
  CREATE PROCEDURE OrderCount( IN customer_ID INT,
                                        OUT Orders INT )
  BEGIN
      SELECT COUNT(SalesOrders.ID)
          INTO Orders
      FROM Customers
          KEY LEFT OUTER JOIN SalesOrders
      WHERE Customers.ID = customer_ID;
  END;
  ```

You can test this procedure in Interactive SQL using the following statements, which show the number of orders placed by the customer with ID 102:

```
CREATE VARIABLE orders INT;
CALL OrderCount ( 102, orders );
SELECT orders;
```

Notes

♦ The *customer_ID* parameter is declared as an IN parameter. This parameter holds the customer ID passed in to the procedure.

♦ The *Orders* parameter is declared as an OUT parameter. It holds the value of the orders variable returned to the calling environment.

♦ No DECLARE statement is necessary for the *Orders* variable, as it is declared in the procedure argument list.

♦ The SELECT statement returns a single row and places it into the variable *Orders*.

Returning result sets from procedures

Result sets allow a procedure to return more than one row of results to the calling environment.

The following procedure returns a list of customers who have placed orders, together with the total value of the orders placed. The procedure does not list customers who have not placed orders.

```
CREATE PROCEDURE ListCustomerValue()
RESULT ("Company" CHAR(36), "Value" INT)
```

```
BEGIN
    SELECT CompanyName,
        CAST( sum(    SalesOrderItems.Quantity *
                        Products.UnitPrice)
                        AS INTEGER ) AS value
    FROM Customers
        INNER JOIN SalesOrders
        INNER JOIN SalesOrderItems
        INNER JOIN Products
    GROUP BY CompanyName
    ORDER BY value DESC;
END;
```

- Type the following:

  ```
  CALL ListCustomerValue ()
  ```

Company	Value
The Hat Company	5016
The Igloo	3564
The Ultimate	3348
North Land Trading	3144
Molly's	2808
…	…

Notes

- The number of variables in the RESULT list must match the number of the SELECT list items. Automatic data type conversion is performed where possible if data types do not match.

- The RESULT clause is part of the CREATE PROCEDURE statement, and does not have a command delimiter.

- The names of the SELECT list items do not need to match those of the RESULT list.

- When testing this procedure, Interactive SQL displays only the first result set by default. You can configure Interactive SQL to display more than one result set by setting the Show Multiple Result Sets option on the Results tab of the Options dialog.

- 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.

  For information about modifying result sets in Interactive SQL, 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 dialog. 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

1. In Interactive SQL, choose Tools ► Options.
2. In the resulting Options dialog, click the Results area.
3. Select the Show Multiple Result Sets checkbox.

After you enable this option, a procedure can return more than one result set to the calling environment. If a RESULT clause is employed, the result sets must be compatible: they must have the same number of items in the SELECT lists, and the data types must all be of types that can be automatically converted to the data types listed in the RESULT list.

Example

The following procedure lists the names of all employees, customers, and contacts listed in the database:

```
CREATE PROCEDURE ListPeople()
RESULT ( Surname CHAR(36), GivenName CHAR(36) )
BEGIN
    SELECT Surname, GivenName
    FROM Employees;
    SELECT Surname, GivenName
    FROM Customers;
    SELECT Surname, GivenName
    FROM Contacts;
END;
```

To test this procedure and view multiple result sets in Interactive SQL, enter the following statement in the SQL Statements pane:

```
CALL ListPeople ();
```

Returning variable result sets from procedures

The RESULT clause is optional in procedures. Omitting the result clause allows you to write procedures that return different result sets, with different numbers or types of columns, depending on how they are executed.

If you do not use the variable result sets feature, you should use a RESULT clause for performance reasons.

For example, the following procedure returns two columns if the input variable is Y, but only one column otherwise:

```
CREATE PROCEDURE Names( IN formal char(1) )
BEGIN
    IF formal = 'y' THEN
        SELECT Surname, GivenName
        FROM Employees
        ELSE
            SELECT GivenName
            FROM Employees
        END IF
    END;
```

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 on 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 751, it illustrates several features of the stored procedure language.

```
CREATE PROCEDURE TopCustomerValue(
        OUT TopCompany CHAR(36),
        OUT TopValue INT )
BEGIN
    -- 1. Declare the "row not found" exception
    DECLARE err_notfound
        EXCEPTION FOR SQLSTATE '02000';

 -- 2.    Declare variables to hold
    --        each company name and its value
    DECLARE ThisName CHAR(36);
    DECLARE ThisValue INT;

 -- 3.    Declare the cursor ThisCompany
    --        for the query
    DECLARE ThisCompany CURSOR FOR
    SELECT CompanyName,
            CAST( sum( SalesOrderItems.Quantity *
                    Products.UnitPrice ) AS INTEGER )
```

```
                AS value
        FROM Customers
            INNER JOIN SalesOrders
            INNER JOIN SalesOrderItems
            INNER JOIN Products
        GROUP BY CompanyName;

     -- 4. Initialize the values of TopValue
        SET TopValue = 0;
        -- 5. Open the cursor
        OPEN ThisCompany;

     -- 6. Loop over the rows of the query
        CompanyLoop:
        LOOP
            FETCH NEXT ThisCompany
                INTO ThisName, ThisValue;
            IF SQLSTATE = err_notfound THEN
                LEAVE CompanyLoop;
            END IF;
            IF ThisValue > TopValue THEN
                SET TopCompany = ThisName;
                SET TopValue = ThisValue;
            END IF;
        END LOOP CompanyLoop;

     -- 7. Close the cursor
        CLOSE ThisCompany;
    END
```

Notes

The TopCustomerValue procedure has the following notable features:

- The "row not found" exception is declared. This exception signals, later in the procedure, when a loop over the results of a query completes.

  For more information about exceptions, see "Errors and warnings in procedures and triggers" on page 759.

- Two local variables ThisName and ThisValue are declared to hold the results from each row of the query.
- The cursor ThisCompany is declared. The SELECT statement produces a list of company names and the total value of the orders placed by that company.
- The value of TopValue is set to an initial value of 0, for later use in the loop.
- The ThisCompany cursor opens.
- The LOOP statement loops over each row of the query, placing each company name in turn into the variables ThisName and ThisValue. If ThisValue is greater than the current top value, TopCompany and TopValue are reset to ThisName and ThisValue.
- The cursor closes at the end of the procedure.
- You can also write this procedure without a loop by adding an ORDER BY value DESC clause to the SELECT statement. Then, only the first row of the cursor needs to be fetched.

The LOOP construct in the TopCompanyValue procedure is a standard form, exiting after the last row is processed. You can rewrite this procedure in a more compact form using a FOR loop. The FOR statement combines several aspects of the above procedure into a single statement.

```
CREATE PROCEDURE TopCustomerValue2(
        OUT TopCompany CHAR(36),
        OUT TopValue INT )
BEGIN
    -- 1. Initialize the TopValue variable
    SET TopValue = 0;
    -- 2. Do the For Loop
    FOR CompanyFor AS ThisCompany
        CURSOR FOR
        SELECT CompanyName AS ThisName ,
            CAST( sum( SalesOrderItems.Quantity *
                    Products.UnitPrice ) AS INTEGER )
            AS ThisValue
        FROM Customers
            INNER JOIN SalesOrders
            INNER JOIN SalesOrderItems
            INNER JOIN Products
        GROUP BY ThisName
    DO
        IF ThisValue > TopValue THEN
            SET TopCompany = ThisName;
            SET TopValue = ThisValue;
        END IF;
    END FOR;
END;
```

Updating a cursor inside a stored procedure

The following procedure uses an updatable cursor on a SELECT statement. It illustrates how to perform an UPDATE on a row using the stored procedure language.

```
CREATE PROCEDURE UpdateSalary(
  IN employeeIdent INT,
  IN salaryIncrease NUMERIC(10,3) )
BEGIN


-- Procedure to increase (or decrease) an employee's salary
   DECLARE err_notfound
      EXCEPTION FOR SQLSTATE '02000';
  DECLARE oldSalary NUMERIC(20,3);
  DECLARE employeeCursor
    CURSOR FOR SELECT Salary from Employees
               WHERE EmployeeID = employeeIdent
    FOR UPDATE;


  OPEN employeeCursor;
  FETCH employeeCursor INTO oldSalary FOR UPDATE;
  IF SQLSTATE = err_notfound THEN
    MESSAGE 'No such employee' TO CLIENT;
  ELSE
    UPDATE Employees SET Salary = oldSalary + salaryIncrease
      WHERE CURRENT OF employeeCursor;
  END IF;
  CLOSE employeeCursor;
END
```

The following statement calls the above stored procedure:

```
CALL UpdateSalary( 105, 220.00 );
```

Errors and warnings in procedures and triggers

After an application program executes a SQL statement, it can examine a **status code**. This status code (or return code) indicates whether the statement executed successfully or failed and gives the reason for the failure. You can use the same mechanism to indicate the success or failure of a CALL statement to a procedure.

Error reporting uses either the SQLCODE or SQLSTATE status descriptions. For full descriptions of SQLCODE and SQLSTATE error and warning values and their meanings, see SQL Anywhere 10 - Error Messages [*SQL Anywhere 10 - 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 755 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 763. Warnings are handled in a slightly different manner from errors: for a description, see "Default handling of warnings in procedures and triggers" on page 762.

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. For more information, 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 terminates execution 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.

```
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;
```

Notes

- 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.

The following statement executes the OuterProc procedure:

```
CALL OuterProc();
```

The Interactive SQL Messages tab displays the following:

```
Hello from OuterProc.

Hello from InnerProc.
```

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. For more information, 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 example illustrates how this works.

Remember to drop both the InnerProc and OuterProc procedures by entering the following commands in the SQL Statements pane before continuing with the tutorial:

```
DROP PROCEDURE OuterProc;
DROP PROCEDURE InnerProc;
```

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:

```
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;
```

The following statement executes the OuterProc procedure:

```
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 is as follows:

1. OuterProc executes and calls InnerProc.
2. In InnerProc, the SIGNAL statement signals an error.
3. The MESSAGE statement is not an error-handling statement, so control is passed back to OuterProc and the message is not displayed.
4. In OuterProc, the statement following the error assigns the SQLSTATE value to the variable named **res**. This is an error-handling statement, and so execution continues and the OuterProc message appears.

Default handling of warnings in procedures and triggers

Errors and warnings are handled differently. While the default action for errors is to set a value for the SQLSTATE and SQLCODE variables, and return control to the calling environment in the event of an error, the default action for warnings is to set the SQLSTATE and SQLCODE values and continue execution of the procedure.

Remember to drop both the InnerProc and OuterProc procedures by entering the following commands in the SQL Statements pane before continuing with the tutorial:

```
DROP PROCEDURE OuterProc;
DROP PROCEDURE InnerProc;
```

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 759. In this case, the SIGNAL statement generates a `row not found` condition, which is a warning rather than an error.

```
CREATE PROCEDURE OuterProc()
BEGIN
    MESSAGE 'Hello from OuterProc.' TO CLIENT;
    CALL InnerProc();
    MESSAGE 'SQLSTATE set to ',
        SQLSTATE,' in OuterProc.' TO CLIENT;
END;
CREATE PROCEDURE InnerProc()
BEGIN
    DECLARE 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;
```

The following statement executes the OuterProc procedure:

```
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 745). 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.

Remember to drop both the InnerProc and OuterProc procedures by entering the following commands in the SQL Statements pane before continuing with the tutorial:

```
DROP PROCEDURE OuterProc;
DROP PROCEDURE InnerProc;
```

The demonstration procedures used to illustrate exception handling are based on those used in "Default error handling in procedures and triggers" on page 759. In this case, additional code handles `the column not found` error in the InnerProc procedure.

```
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;
```

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.

The following statement executes the OuterProc procedure:

```
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.
```

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.

Remember to drop both the InnerProc and OuterProc procedures by entering the following commands in the SQL Statements pane before continuing with the tutorial:

```
DROP PROCEDURE OuterProc;
DROP PROCEDURE InnerProc;
```

The following demonstration procedure 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 759.

```
CREATE PROCEDURE InnerProc()
BEGIN
    BEGIN
        DECLARE column_not_found
        EXCEPTION FOR SQLSTATE VALUE '52003';
        MESSAGE 'Hello from InnerProc' TO CLIENT;
        SIGNAL column_not_found;
        MESSAGE 'Line following SIGNAL' TO CLIENT
        EXCEPTION
            WHEN column_not_found THEN
                MESSAGE 'Column not found handling' TO
                CLIENT;
            WHEN OTHERS THEN
                RESIGNAL;
    END;
    MESSAGE 'Outer compound statement' TO CLIENT;
END;
```

The following statement executes the InnerProc procedure:

```
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.

Drop the InnerProc procedure by executing the following statement:

```
DROP PROCEDURE InnerProc;
```

Using the EXECUTE IMMEDIATE statement in procedures

The EXECUTE IMMEDIATE statement allows statements to be constructed inside procedures using a combination of literal strings (in quotes) and variables.

For example, the following procedure includes an EXECUTE IMMEDIATE statement that creates a table.

```
CREATE PROCEDURE CreateTableProcedure(
        IN tablename char(128) )
BEGIN
    EXECUTE IMMEDIATE 'CREATE TABLE '
   || tablename
   || '(column1 INT PRIMARY KEY)'
END;
```

The EXECUTE IMMEDIATE statement can be used with queries that return result sets. For example:

```
CREATE PROCEDURE DynamicResult(
   IN Columns LONG VARCHAR,
   IN TableName CHAR(128),
   IN Restriction LONG VARCHAR DEFAULT NULL )
BEGIN
    DECLARE Command LONG VARCHAR;
    SET Command = 'SELECT ' || Columns || ' FROM ' || TableName;
    IF ISNULL( Restriction,'') <> '' THEN
          SET Command = Command || ' WHERE ' || Restriction;
    END IF;
    EXECUTE IMMEDIATE WITH RESULT SET ON Command;
END;
```

The following statement calls this procedure:

```
CALL DynamicResult(
   'table_id,table_name',
   'SYSTAB',
   'table_id <= 10');
```

table_id	table_name
1	ISYSTAB
2	ISYSTABCOL
3	ISYSIDX
...	...

In ATOMIC compound statements, you cannot use an EXECUTE IMMEDIATE statement that causes a COMMIT, as COMMITs are not allowed in that context.

☞ For more information about the EXECUTE IMMEDIATE statement, 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 (see "Using Transactions and Isolation Levels" on page 111). 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 (see "Atomic compound statements" on page 745). 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 (see "Savepoints within transactions" on page 115) 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.

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 terminate 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.

☞ For more information on dates and times, see “Date and time data types” [*SQL Anywhere Server - SQL Reference*].

Verifying that procedure input arguments are passed correctly

One way to verify input arguments is to display the value of the parameter on the Interactive SQL Messages tab using the MESSAGE statement. For example, the following procedure simply displays the value of the input parameter *var*:

```
CREATE PROCEDURE message_test( IN var char(40) )
BEGIN
    MESSAGE var TO CLIENT;
END;
```

You can also use the debugger to verify that procedure input arguments were passed correctly. See “Lesson 2: Debug a stored procedure” on page 782.

Statements allowed in procedures, triggers, events, and batches

All SQL statements are acceptable in batches (including data definition statements such as CREATE TABLE, ALTER TABLE, and so on), with the exception of the following:

- ALTER DATABASE (syntax 3)
- CONNECT
- CREATE DATABASE
- CREATE DECRYPTED FILE
- CREATE ENCRYPTED FILE
- DISCONNECT
- DROP CONNECTION
- DROP DATABASE
- FORWARD TO
- Interactive SQL commands such as INPUT or OUTPUT
- PREPARE TO COMMIT
- STOP 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 768.

For more information, see the Usage for each SQL statement in the chapter "SQL Statements" [*SQL Anywhere Server - SQL Reference*].

Using SELECT statements in batches

You can include one or more SELECT statements in a batch.

The following is a valid batch:

```
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.

Calling external libraries from procedures

You can call a function in an external library from a stored procedure or user-defined function. You can call functions in a DLL under Windows operating systems, in an NLM under NetWare, and in a shared object on Unix. You cannot call external functions on Windows CE.

This section describes how to use the external library calls in procedures. Sample external stored procedures, plus the files required to build a DLL containing them, are located in the following folder: *samples-dir \SQLAnywhere\ExternalProcedures*. For information on the location of *samples-dir*, see "The samples directory" [*SQL Anywhere Server - Database Administration*].

Caution
External libraries called from procedures share the memory of the server. If you call an external library from a procedure and the external library contains memory-handling errors, you can crash the server or corrupt your database. Ensure that you thoroughly test your libraries before deploying them on production databases.

The API described in this section replaces an older API. The older API is deprecated. Libraries written to the older API, used in versions before version 7.0.x, are still supported, but in new development you should use the new API.

SQL Anywhere includes a set of system procedures that make use of this capability, for example to send MAPI email messages.

☞ For more information on system procedures, see "System Procedures" [*SQL Anywhere Server - SQL Reference*].

Creating procedures and functions with external calls

This section presents some examples of procedures and functions with external calls.

DBA authority required
You must have DBA authority to create procedures or functions that reference external libraries. This requirement is more strict than the RESOURCE authority required for creating other procedures or functions.

Syntax

You can create a procedure that calls a function *function_name* in DLL *library.dll* as follows:

```
CREATE PROCEDURE dll_proc( parameter-list )
EXTERNAL NAME 'function_name@library.dll'
```

If you call an external DLL from a procedure, the procedure cannot perform any other tasks; it just forms a wrapper around the DLL.

An analogous CREATE FUNCTION statement is as follows:

```
CREATE FUNCTION dll_func ( parameter-list )
RETURNS data-type
EXTERNAL NAME 'function_name@library.dll'
```

In these statements, *function_name* is the exported name of a function in the dynamic link library, and *library.dll* is the name of the library. The arguments in *parameter-list* must correspond in type and order to the arguments expected by the library function. The library function accesses the procedure arguments using an API described in "External function prototypes" on page 773.

Any value returned by the external function is in turn returned by the procedure to the calling environment.

No other statements permitted

A procedure that references an external function can include no other statements: its sole purpose is to take arguments for a function, call the function, and return any value and returned arguments from the function to the calling environment. You can use IN, INOUT, or OUT parameters in the procedure call in the same way as for other procedures: the input values get passed to the external function, and any parameters modified by the function are returned to the calling environment in OUT or INOUT parameters.

System-dependent calls

You can specify operating-system dependent calls, so that a procedure calls one function when run on one operating system, and another function (presumably analogous) on another operating system. The syntax for such calls involves prefixing the function name with the operating system name. The operating system identifier must be one of **Unix** or **NetWare**.

If the list of functions does not contain an entry for the operating system on which the server is running, but the list does contain an entry without an operating system specified, the database server calls the function in that entry.

NetWare calls have a slightly different format than the other operating systems. All symbols are globally known under NetWare, so any symbol (such as a function name) exported must be unique to all NLMs on the system. Consequently, the NLM name is not necessary in the call as long as the NLM is already loaded. It is recommended that you always use the library name, regardless of whether the NLM is already loaded. If the NLM is *not* already loaded, you *must* provide a library name. The file extension *.nlm* is optional.

☞ For more information about the CREATE PROCEDURE statement syntax, see "CREATE PROCEDURE statement" [*SQL Anywhere Server - SQL Reference*].

☞ For more information about the CREATE FUNCTION statement syntax, see "CREATE FUNCTION statement" [*SQL Anywhere Server - SQL Reference*].

External function prototypes

This section describes the API for functions in external libraries.

The API is defined by a header file named *extfnapi.h*, in the *h* subdirectory of your SQL Anywhere installation directory. This header file handles the platform-dependent features of external function prototypes. The API supercedes a previous API for functions in external libraries.

Declaring the API version

To notify the database server that the external library is not written using the old API, provide a function as follows:

uint32 extfn_use_new_api( )

The function returns an unsigned 32-bit integer. The returned value must be the API version number, EXTFN_API_VERSION, defined in *extfnapi.h*. A return value of 0 means that the old API is being used.

If the function is not exported by the DLL, the database server assumes that the old API is in use. The new API must be used for all Unix platforms and for all 64-bit platforms, including 64-bit Windows.

On NetWare, to notify the database server that the external procedure is written using the new API, your NLM must export either a function called extfn_use_new_api, or a function called *name*_use_new_api, where *name* is the name of the NLM. For example, an NLM named *external.nlm* would export a function external_use_new_api.

Exporting *name*_use_new_api avoids export name conflicts when more than one external NLM is in use at one time. If the NLM exports a function called *name*_use_new_api then the CREATE PROCEDURE or CREATE FUNCTION statement must contain the NLM name.

Function prototypes

The name of the function must match that referenced in the CREATE PROCEDURE or CREATE FUNCTION statement. The function declaration must be as follows:

void *function-name***(an_extfn_api ***api*, **void ***argument-handle* **)**

The function must return void, and must take as arguments a structure used to pass the arguments, and a handle to the arguments provided by the SQL procedure.

The **an_extfn_api** structure has the following form:

```
typedef struct an_extfn_api {
    short (SQL_CALLBACK *get_value)(
                    void *          arg_handle,
                    a_sql_uint32    arg_num,
                    an_extfn_value *value
                    );
    short (SQL_CALLBACK *get_piece)(
                    void *          arg_handle,
                    a_sql_uint32    arg_num,
                    an_extfn_value *value,
                    a_sql_uint32    offset
                    );
    short (SQL_CALLBACK *set_value)(
                    void *          arg_handle,
                    a_sql_uint32    arg_num,
                    an_extfn_value *value
                    short          append
                    );
void (SQL_CALLBACK *set_cancel)(
                    void *    arg_handle,
                    void *    cancel_handle
                    );
} an_extfn_api;
```

The **an_extfn_value** structure has the following form:

```
typedef struct an_extfn_value {
    void *                  data;
    a_sql_uint32           piece_len;
    union {
        a_sql_uint32    total_len;
        a_sql_uint32    remain_len;
    } len;
    a_sql_data_type         type;
} an_extfn_value;
```

Notes

Calling **get_value** on an OUT parameter returns the data type of the argument, and returns data as NULL.

The **get_piece** function for any given argument can only be called immediately after the **get_value** function for the same argument,

To return NULL, set **data** to NULL in **an_extfn_value**.

The **append** field of **set_value** determines whether the supplied data replaces (false) or appends to (true) the existing data. You must call **set_value** with `append=FALSE` before calling it with `append=TRUE` for the same argument. The **append** field is ignored for fixed length data types.

The header file itself contains some additional notes.

The following table shows the conditions under which the functions defined in **an_extfn_api** return false:

Function	**Returns 0 when the following is true; else returns 1**
get_value()	- **arg_num** is invalid; for example, **arg_num** is greater than the number of arguments in **ext_fn** - It is called before the external function call has been properly initialized
get_piece()	- **arg_num** is invalid; for example, **arg_num** does not correspond to the last **get_value**. |- The offset is greater than the total length of the value for the arg_num argument. - It is called before the external function call has been properly initialized.
set_value()	- **arg_num** is invalid; for example, **arg_num** is greater than the number of arguments in **ext_fn**. - **arg_num** argument is input only. - The type of value supplied does not match that of the **arg_num** argument. - It is called before the external function call has been properly initialized.

☞ For more information about the values you can enter in the a_sql_data_type field, see “Embedded SQL data types” [*SQL Anywhere Server - Programming*].

For more information about passing parameters to external functions, see “Passing parameters to external functions” on page 776.

Implementing cancel processing

An external function that expects to be canceled must inform the database server by calling the **set_cancel** API function. You must export a special function to enable external operations to be canceled. This function must have the following form:

void an_extfn_cancel(void * *cancel_handle* **)**

If the DLL does not export this function, the database server ignores any user interrupts for functions in the DLL. In this function, *cancel_handle* is a pointer provided by the function being canceled to the database server upon each call to the external function by the *set_cancel* API function listed in the **an_extfn_api** structure, above.

Passing parameters to external functions

Data types

The following SQL data types can be passed to an external library:

SQL data type	C type
CHAR	Character data, with a specified length
VARCHAR	Character data, with a specified length
LONG VARCHAR	Character data, with a specified length
BINARY	Binary data, with a specified length
LONG BINARY	Character data, with a specified length
TINYINT	1-byte integer
[UNSIGNED] SMALLINT	[Unsigned] 2-byte integer
[UNSIGNED] INT	[Unsigned] 4-byte integer
[UNSIGNED] BIGINT	[Unsigned] 8-byte integer
VARBINARY	Binary data, with a specified length
REAL	Single precision floating point number
DOUBLE	Double precision floating point number

You cannot use date or time data types, and you cannot use exact numeric data types.

To provide values for INOUT or OUT parameters, use the **set_value** API function. To read IN and INOUT parameters, use the **get_value** API function.

Passing NULL

You can pass NULL as a valid value for all arguments. Functions in external libraries can supply NULL as a return value for any data type.

Return values

To set a return value for an external fucntion, call the set_value function with an arg_num parameter of 0. If set_value is not called with arg_num=0, the function result is NULL.

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.SYSUSERPERM 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;
```

☞ For more information, see the “ALTER FUNCTION statement” [*SQL Anywhere Server - SQL Reference*], the “ALTER PROCEDURE statement” [*SQL Anywhere Server - SQL Reference*], the “ALTER TRIGGER statement” [*SQL Anywhere Server - SQL Reference*], and the “ALTER VIEW statement” [*SQL Anywhere Server - SQL Reference*].

CHAPTER 21

Debugging Procedures, Functions, Triggers, and Events

Contents

About this chapter

This chapter describes how to use the SQL Anywhere debugger to assist in developing SQL stored procedures, triggers, and event handlers.

Introduction to debugging in the database

You can use the debugger during the development of SQL stored procedures, triggers, event handlers, and user-defined functions.

This chapter describes how to set up and use the debugger.

Debugger features

You can perform many tasks with the debugger, including the following:

- **Debug procedures and triggers** You can debug SQL stored procedures and 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.

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 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 "The samples directory" [*SQL Anywhere Server - Database Administration*].

Lesson 1: Connect to a database and start the debugger

This lesson describes how to start the SQL Anywhere debugger.

Start the debugger

♦ To start the debugger

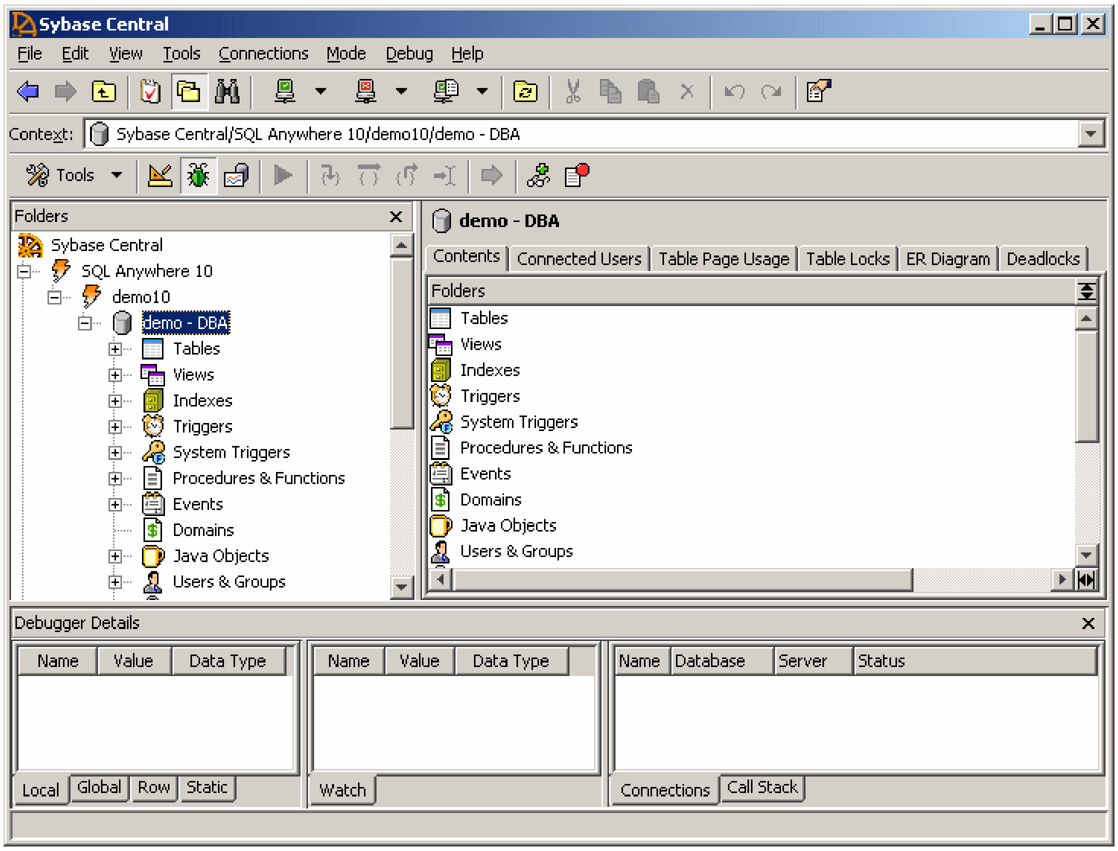
1. Start Sybase Central by choosing Start ► Programs ► SQL Anywhere 10 ► Sybase Central.
2. Connect to the database.

   a. From the Connections menu, choose Connect with SQL Anywhere 10.

      The Connect dialog appears.

   b. In the ODBC Data Source Name field, type **SQL Anywhere 10 Demo** and then click OK.
3. Choose Mode ► Debug.

   Sybase Central can be used in Design, Debug, or Application Profiling mode. When running in Debug mode, debugger breakpoints are active. The Debugger Details pane appears at the bottom of Sybase Central and the Sybase Central toolbar displays a set of debugger tools.



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, open the Procedures and Functions folder.
2. Execute the procedure.

   Right-click the debugger_tutorial procedure and choose Execute from Interactive SQL from the popup menu.

   Interactive SQL opens and the following result set appears:

top_company	top_value
(NULL)	(NULL)

   This is clearly 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. Ensure you are in Debug mode in Sybase Central.
2. Set a breakpoint at the first executable statement in the procedure.

   The statement contains the following text:

   ```
   OPEN cursor_this_customer;
   ```

   Click to the left of this line in the vertical gray bar to set a breakpoint. The breakpoint appears as a red circle. An alternative way to set a breakpoint is to press F9.
3. Execute the procedure again.

   a. In the left pane, right-click the debug_tutorial procedure and choose Execute from Interactive SQL from the popup menu.

   b. A message box appears, asking if you want to debug the connection from Sybase Central. Click Yes.

      Execution of the procedure stops at the breakpoint. A yellow arrow in the source code window indicates the current position, which is at the breakpoint.
4. Inspect variables.

The Local tab in the Debugger Details pane displays 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.

5. Step through the stored procedure by pressing F11. As you step through the lines of the stored procedure, the value of the variables changes.

6. Stop stepping through the code when you reach the following line:

   ```
   IF this_value > top_value THEN
   ```

   When you are at the `IF` statement, this_value is set to 1452 and top_value is still NULL.

7. Step into one more statement.

   Press F11 once more to see which branch the execution takes. The yellow arrow moves directly back to the label statement at the beginning of the loop, which contains the following text:

   ```
   customer_loop: loop
   ```

   The `IF` test did not return true. The test failed because a comparison of any value to NULL returns NULL. A value of NULL fails the test and the code inside the `IF`...`END IF` statement is not executed.

   At this point, you may realize that the problem is the fact that top_value is not initialized.

Confirm the diagnosis and fix the bug

You can test the hypothesis that the problem is the lack of initialization for top_value right in the debugger, without changing the procedure code.

♦ To test the hypothesis

1. Set a value for top_value.

   In the Local window, click the Value field of the top_value variable, and enter a value of 3000.

2. Step through the loop again.

   Press F11 to step through the instructions to the `IF` statement and check the values of this_value and top_value. You may have to step through several loops until you get a value of this_value greater than 3000.

3. Disable the breakpoint and execute the procedure.

   a. Click the breakpoint so that it turns gray (disabled).

   b. Press F5 to complete execution of the procedure.

      The Interactive SQL window appears again. It shows the correct results.

top_company	top_value
Chadwicks	8076

The hypothesis is confirmed. The problem is that the top_value is not initialized.

♦ To fix the bug

1. From the Mode menu, choose Design mode.
2. Immediately after the line containing the following text

   ```
   OPEN cursor_this_customer;
   ```

   Create a new line that initializes the top_value variable:

   ```
   SET top_value = 0;
   ```
3. Press Ctrl+S to save the modified procedure.
4. 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

This section describes how to use breakpoints to control when the debugger interrupts execution of your source code.

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. With Sybase Central running in Debug mode, display the code where you want to set a breakpoint.
2. Click the line on which you want to insert a breakpoint

   A cursor appears in the line where you clicked
3. Press F9 to set the breakpoint.

   A red circle appears to the left of the line of code.

   You can also double-click the grey column just to the left of the line of code to insert or remove a breakpoint.

♦ To set a breakpoint (Debug menu)

1. With Sybase Central running in Debug mode, choose Debug ► Breakpoints.

   The Breakpoints dialog appears.
2. Click New.

   The New Breakpoint dialog appears.
3. Choose a Procedure name from the dropdown list, and, optionally, enter values for Condition and Count.

   The Condition is a SQL expression that must evaluate to true for the breakpoint to interrupt execution. For example, you can set a breakpoint to apply to a connection made by a specified user, by entering the following condition:

   ```
   CURRENT USER = 'user-name'
   ```

   The Count is a number of times the breakpoint is hit before it stops execution. A value of 0 means that the breakpoint always stops execution.
4. Click OK to set the breakpoint. 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 dialog.

♦ To change the status of a breakpoint

1. Display the source code for the procedure that contains the breakpoint whose status you want to change.
2. Click the breakpoint indicator to the left of the line you want to edit. The status of the line switches from being an active breakpoint being a disabled breakpoint.

♦ To change the status of a breakpoint (Breakpoints dialog)

1. Open the Breakpoints dialog.
2. Edit the breakpoint.

   Alternatively, you can delete a breakpoint by selecting the breakpoint and then pressing Delete.

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. From the Debug menu, choose Breakpoints.

   The Breakpoints dialog appears.
2. Select the breakpoint you want to edit and then click Edit.

   The Edit Breakpoint dialog appears.
3. Enter a condition in the Condition field.

   For example, to set the breakpoint so that it applies only to connections from a specific user ID, enter the following condition:

   ```
   CURRENT USER='user-name'
   ```

   In this condition, *user-name* is the user ID for which the breakpoint is to be active.

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 watch the values of your variables

1. Set a breakpoint in the procedure whose variables you want to examine.

   ☞ For information on setting breakpoints, see “Setting breakpoints” on page 786.

2. Click the Local tab in the Debugger Details pane.
3. Run the procedure. 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 on the Globals tab of the Debugger Details pane.

☞ 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 on the Row tab of the Debugger Details pane.

☞ For more information on triggers, see “Introduction to triggers” on page 735.

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. Set a breakpoint in the procedure whose variables you want to examine.
2. Run the code to the breakpoint.
3. Click the Calls Stack tab in the Debugger Details pane.

   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.

The source code window displays the state for a single connection. To switch connections, double-click a connection on the Connections tab.

A useful technique is to set a breakpoint so that it interrupts execution for a single user ID. You can do this by setting a breakpoint condition of the following form:

```
CURRENT USER = 'user-name'
```

The SQL special value CURRENT USER holds the user ID of the connection.

☞ For more information, see "Editing breakpoint conditions" on page 787, and "CURRENT USER special value" [*SQL Anywhere Server - SQL Reference*].

Index