SQLBase

SQL Language Reference

Trademarks

Centura, Centura Ranger, the Centura logo, Centura Web Developer, Gupta, the Gupta logo, Gupta Powered, the Gupta Powered logo, Fast Facts, Object Nationalizer, Quest, Quest/Web, QuickObjects, SQL/API, SQLBase, SQLConsole, SQLGateway, SQLHost, SQLNetwork, SQLRouter, SQLTalk, and Team Object Manager are trademarks of Centura Software Corporation and may be registered in the United States of America and/or other countries. SQLWindows is a registered trademark and TeamWindows, ReportWindows and EditWindows are trademarks exclusively used and licensed by Centura Software Corporation.

Microsoft, Win32, Windows, Windows NT and Visual Basic are either registered trademarks or trademarks of Microsoft Corporation in the United States of America and/or other countries.

Java is a trademark of Sun Microsystems Inc.

All other product or service names mentioned herein are trademarks or registered trademarks of their respective owners.

Copyright

Contents

Preface

This manual is a reference guide for the SQL commands supported in SQLBase. You can use the SQL commands documented in this manual with the following Centura products:

- SQLTalk
- Team Developer
- SQL/API
- SQLGateways and SQLRouters
- SQLConsole

Consult the manual for the specific product you are using for more information.

Who should read this manual

This manual is intended for:

- Application Developers

    Application developers build client applications that access databases using Centura frontend products like SQLTalk, Team Developer, and the SQL/API.

- Database Administrators (DBAs)

    Database Administrators perform day-to-day operation and maintenance of the database. They design the database, create database objects, load data, control access, perform backup and recovery, and monitor performance.

- End Users

    End users use SQL to query and change data.

This manual assumes you have:

- Knowledge of relational databases and SQL.

Note: This manual is not intended to be a SQL tutorial.

Summary of chapters

This manual is organized in the chapters in the table below. There is also a glossary and index.

1	Introduction to SQL	Shows the SQL command categories and features.
2	SQL Elements	Explains the concepts needed to use SQL.
3	SQL Command Reference	Describes each SQL command. Arranged alphabetically.
4	SQL Function Reference	Lists SQL reserved words.
5	SQL Reserved Words	Lists SQL reserved words.
6	Referential Integrity	Describes SQLBase's implementation of referential integrity.
7	Procedures and Triggers	Describes SQLBase's implementation of procedures and triggers.
8	Optimizing SQL Statements	Describes how to optimize SQL statements for SQLBase performance.
	Appendix A	Provides the description, syntax, and examples for SAL functions supported by SQLBase procedures.

Syntax diagrams

This manual uses syntax diagrams to show how to enter commands.

The syntax for the CREATE INDEX command is used here as an example.
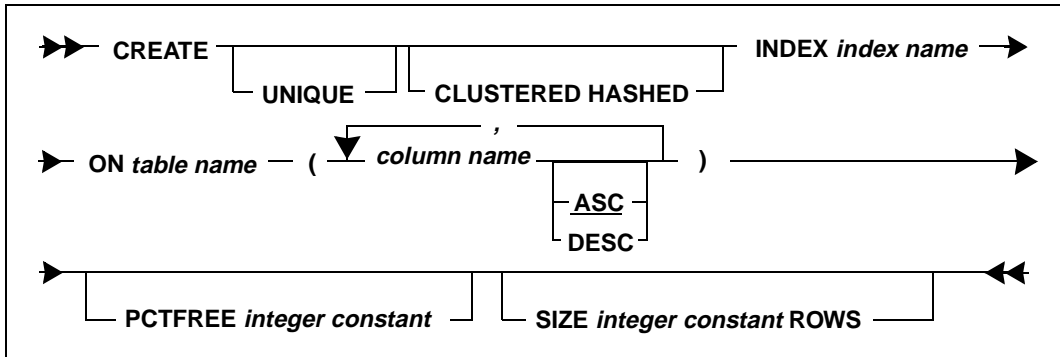
Read the syntax diagram from left to right and top to bottom.

The line with the command name (CREATE) is the main line of the command.
Mandatory keywords and arguments (such as INDEX or ON *table name*) appear on
the main line or a continuation of the main line.

This example diagram could generate the commands shown in these examples:
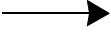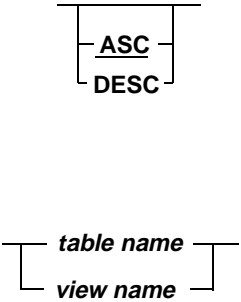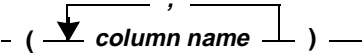
```
CREATE UNIQUE INDEX EMP_IDX ON EMP (EMPNO);

CREATE INDEX ORDER_IDX ON ORDERS (ORDERNO, ORDERDATE);
```

Note that example statements in this manual can appear in bold to distinguish user
entries from a system response:

ROWCOUNT EMP;

```
5 ROWS IN TABLE
```

The following table shows the syntax diagram symbols used in this manual.

Symbol	Description
▶▶	A double arrow pointing right means the start of a command.
⟶	A single arrow pointing right means a continuation line of a command.
◀◀	The double arrow pointing left means the end of a command.
UNIQUE	Optional clauses and keywords (such as UNIQUE) hang off the main or continuation lines.
ASC **DESC**	If there is an optional item with alternate choices, the choices are in a vertical list. In this example, ASC and DESC are alternate non-mandatory options. ASC is underlined, which means it is the default and can be omitted.
table name **view name**	If an item is mandatory, the first alternative is on the main line (this example is from the UPDATE command).
(column name ,)	When you can repeat arguments of the same type (such as a list of column names), an arrow pointing downward is suspended above the argument. A delimiter or operator on this line shows what separates each argument (such as commas separating column names).

Notation conventions

The table below show the notation conventions that this manual uses.

Notation	Explanation
You	A developer who reads this manual
User	The end-user of applications that you write
bold type	Menu items, push buttons, and field names. Things that you select. Keyboard keys that you press.
Courier 9	Builder or C language code example
SQL.INI MAPDLL.EXE	Program names and file names
Precaution	**Warning:**
Vital information	**Important:**
Supplemental information	**Note:**
Alt+1	A plus sign between key names means to press and hold down the first key while you press the second key
TRUE FALSE	These are numeric boolean constants defined internally in Builder:

For the TRUE/FALSE row:

Constant	Value	Meaning
TRUE	1	Successful, on, set
FALSE	0	Unsuccessful, off, clear

Other helpful resources

Centura Books Online. The Centura document suite is available online. This document collection lets you perform full-text indexed searches across the entire document suite, navigate the table of contents using the expandable/collapsible browser, or print any chapter. Open the collection by selecting the Centura Books Online icon from the **Start** menu or by double-clicking on the launcher icon in the program group.

Centura Online Help. This is an extensive context-sensitive online help system. The online help offers a quick way to find information on topics including menu items, functions, messages, and objects.

World Wide Web. Centura Software's World Wide Web site contains information about Centura Software Corporation's partners, products, sales, support, training, and users. The URL is http://www.centurasoft.com.

To access Centura technical services on the Web, go to http:/www.centurasoft.com/support. This section of our Web site is a valuable resource for customers with technical support issues, and addresses a variety of topics and services, including technical support case status, commonly asked questions, access to Centura's Online Newsgroups, links to Shareware tools, product bulletins, white papers, and downloadable product updates.

For information on training, including course descriptions, class schedules, and Certified Training Partners, go to http://www.centurasoft.com/training.

Send comments to...

Anyone reading this manual can contribute to it. If you have any comments or suggestions, please send them to:

Technical Publications Department
Centura Software Corporation
975 Island Drive
Redwood Shores, CA 94065

or send email, with comments or suggestions to:

techpubs@centurasoft.com

Chapter 1

Introduction to SQL

This chapter introduces SQL and its implementation in SQLBase.

What is SQL?

SQL (Structured Query Language) is a complete set of commands that lets you access a relational database. SQL is pronounced *sequel* or *ess-que-ell*.

SQL is the standard interface for many relational databases. It has a simple command structure for data definition, access, and manipulation.

SQL was intended to be used with programming languages, so standard SQL does not have commands for interactive screen dialogue, or for more than very crude report formatting.

SQL is set-oriented. You can perform a command on a group of data rows or on one row.

SQL is non-procedural. When you use SQL you specify *what* you want done, not *how* to do it. To access data you need only to name a table and the columns; you do not have to describe an access method. For example, a single command can update multiple rows in a database without specifying the row's location, storage format, and access format.

SQL has several layers of increasing complexity and capability. End users with little computer experience can use SQL's basic features while programmers can use the advanced features they need.

SQL history

SQL began with a paper published in 1970 by E.F. Codd, a mathematician working at the IBM Research Laboratory in San Jose, California. In this paper, "A Relational Model of Data for Large Shared Data Banks" (*Communications of the ACM*, Vol. 13, No. 6, June 1970) Codd formulated the principles of a relational system for managing a database and described a relational algebra for organizing the data into tables.

Four years later, another important paper followed: "SEQUEL: A Structured English Query Language" (*Proceedings of the 1974 ACM SIGMOD Workshop on Data Description, Access and Control*, May 1974) by D.D. Chamberlin and R.F. Boyce. Both its authors were (like Codd) researchers at IBM's San Jose Research Laboratory. Their paper defined a language (the ancestor of SQL) designed to meet the requirements of Codd's relational algebra.

Two years after that, Chamberlin and others developed a version of the language, SEQUEL/2, and shortly after that IBM built a prototype system called System R that implemented most of its features. Around 1980 the name changed to SQL. Note that today SQL is often pronounced "sequel."

Both the American National Standards Institute (ANSI) and the International Organization for Standardization (ISO) have committees dedicated to establishing

and reviewing SQL standards. The most recent standard released for SQL is known as SQL-92.

Why is SQL used?

SQL's features make it the most widely-used language for relational databases. Here are a few reasons:

- Acceptance

  The American National Standards Institute (ANSI) has approved SQL. The International Standards Organization (ISO) and the U.S. Department of Defense also support SQL. A version of SQL is available on most computers.

- Power

  SQL is powerful. SQL is a complete database language, so you can use it for data definition, data control, and transaction management. SQL commands are simple to use in their basic form, but they have the flexibility to do complex operations.

- Ease of use

  People can easily access and manipulate data without becoming involved with the physical organization and storage complexities of that data.

How you use SQL

You can use SQL in two different ways:

- Interactively through an interface program.
- Embedded in a programming language such as C or SAL (Centura's Scalable Application Language), or in a client application such as a report writer or an application generator.

SQL is not a programming language or even an interactive language. To use SQL, you work through an interface that is part of a proprietary SQL implementation.

You execute SQL commands through a program that provides the interface to the database server and handles things that SQL was not designed to handle. For example, Centura's SQLTalk product handles communications (through a communications library) with the database server when you give SQL commands.

Application end users access the database through business application programs, without the need for prior database knowledge.

Who uses SQL?

End users

End users issue SQL commands to retrieve, insert, update, or delete data either through an interactive command interface or a client application.

Application developers

Developers write programs containing SQL commands to allow end users to access SQLBase data without having to know how the data is accessed. The developers need to know how to write SQL commands and embed them within a program written in C, COBOL, or SAL (Centura's Scalable Application Language).

DBAs

Database administrators (DBAs) use SQL commands to define the database, secure data from unauthorized access, and change data definitions as needed. They use SQL commands to query and report on the database.

Types of SQL commands

With SQL you can:

- Create tables in the database.
- Store data.
- Retrieve data.
- Change data and change the structure of underlying tables.
- Combine and calculate data.
- Provide security.

The SQL commands are grouped into these categories.

Data definition commands (DDL)

These commands create database objects such as tables or views.

```
CREATE DATABASE
CREATE DBAREA
CREATE EXTERNAL FUNCTION
CREATE INDEX
CREATE STOGROUP
CREATE SYNONYM
CREATE TABLE
CREATE TRIGGER
CREATE VIEW
PROCEDURE
```

SQL DROP commands exist for most of these objects, which allow the object to be deleted.

Data manipulation commands (DML)

These commands add, update, or delete data.

> DELETE
> INSERT
> UPDATE

Data query commands (DQL)

The SELECT command retrieves data.

SQL lets you build complex queries with relational operators (such as >, <, =, >=, or <>) that enable you to express a search condition. A query can use a join to pull data from different tables and correlate it by matching on a common row that is in all the tables.

The input to one query can be the output of another query. A nested query is called a *subselect*.

Queries can be nested within INSERT, UPDATE, and DELETE commands to specify the scope of the operation.

Transaction control commands

These commands ensure data integrity when changing data. They ensure that a logically-related sequence of actions that accomplish a particular result in an application (a logical unit of work) are either performed or cancelled in their entirety.

> COMMIT
> ROLLBACK
> SAVEPOINT

Data administration commands

These commands help you analyze system performance and operations.

> AUDIT MESSAGE
> START AUDIT
> STOP AUDIT

Data control commands

In addition to the data definition language (DDL) commands that allow you to create and maintain database objects, the following data control commands include the following maintenance tasks:

- Assigning users to databases and tables.
- Altering database object definitions
- Maintaining databases and partitions

> ALTER DATABASE
> ALTER DBAREA
> ALTER EXTERNAL FUNCTION
> ALTER PASSWORD
> ALTER STOGROUP
> ALTER TABLE
> ALTER TRIGGER
> CHECK DATABASE
> CHECK INDEX
> CHECK TABLE
> COMMENT ON
> DBATTRIBUTE
> DEINSTALL DATABASE
> GRANT
> GRANT EXECUTE ON
> INSTALL DATABASE
> LABEL
> LOAD
> LOCK DATABASE
> REVOKE
> REVOKE EXECUTE ON
> ROWCOUNT
> SET DEFAULT STOGROUP
> UNLOAD
> UNLOCK DATABASE
> UPDATE STATISTICS

Example of a SQL command

The following example shows a SQL query both in conversational English and actual SQL syntax.

English	Give me a list of everyone who works at the Albany location who has the same job as someone who works at the Utica location.
SQL	SELECT LNAME, FNAME, EMPNO FROM EMP WHERE LOC = 'ALBANY' AND JOB IN (SELECT JOB FROM EMP WHERE LOC = 'UTICA');

Some other examples of SQL commands are:

```
SELECT LNAME FROM EMP;
CREATE TABLE FRIENDS (NAME CHAR(15));
SELECT * FROM EMP, EMPSAL
    WHERE EMP.EMPNO = EMPSAL.EMPNO;
ALTER TABLE FRIENDS RENAME TABLE FOLKS;
DROP TABLE FOLKS;
```

What are SQL objects?

With SQL, you can create and use the following SQL objects that allow you to organize and maintain your data:

- Databases
- Tables
- Columns
- Indexes
- Views
- Synonyms
- Stored commands
- Stored procedures
- External functions
- Triggers

Database

A *database* is a set of SQL objects. When you define a database you give a name to an eventual collection of tables and associated indexes.

A single database can contain all the data associated with one application or with a group of related applications. Collecting data into one database lets you start or stop access to all the data in one operation and grant authorization for access to all the data as a unit.

Tables

A database contains one or more *tables*. Each table has a name and contains a specific number of *columns* (vertical) and unordered *rows* (horizontal). Each column in a row is related in some way to the other columns in the same row.

	Column	
CUST_NO	CONTACT	CREDIT
========	=======	=======
46372986	E. Smith	$3000.00
12162344	R. Vince	$1500.00
98121735	G. Handle	$ 580.00
Row 55421888	B. Harty	$2000.00
89923942	S. Jones	$ 550.00

Each column has a name and a data type. Each column contains a data value at the intersection of a row and a column.

In theory, no row in a table should be a duplicate of any other row. For instance, if you define a table of sales orders, the columns might be ORDER_DATE, CUSTOMER_ID, PRODUCT_CODE, and QUANTITY.

If a customer orders 10 widgets one day and then orders another 10 widgets on the same day, there would be 2 duplicate rows in the table. You could either store the time when the order was placed, or have a unique sequence number (such as an invoice number) to identify each order. In each case there will be a column or combination of columns which is different for each order, and so uniquely identifies it.

A *join* retrieves rows from more than one table. This operation is called a join because the rows retrieved from the different tables are joined on one or more columns that appear in two or more of the tables.

A table can have a *primary key* which is a column or a group of columns whose value uniquely identifies each row. Columns of other tables may be *foreign keys*, whose values must be equal to values of the primary key of the first table. The rule that a

value of a foreign key must appear as a value of some specific table is called a *referential constraint*.

SQLBase uses SQL commands to add new columns to an existing table or make an existing column wider. The change takes effect immediately and no database reorganization is needed.

Indexes

An *index* is an ordered set of pointers to the data in a table, stored separately from the table. Each index is based on the values of data in one or more columns of a table.

Users accessing a table need not be aware that SQLBase is using an index. SQLBase decides whether to use an index to access a table.

An index provides two benefits:

- **Improves performance.** Access to data is faster.
- **Ensures uniqueness.** A table with a unique index cannot have two rows with the same values in the column or columns that form the index key.

Views

A *view* is an alternate way of representing data that exists in one or more tables. A view can include all or some of the columns from one or more *base tables*. You can also base a view on other views or on a combination of views and tables.

A view looks like a table and you can use it as though it were a table. You can use a view name in a SQL command as though it were a table name. You cannot do some operations through a view, but you do not need to know that an apparent table is actually a view.

A table has a storage representation, but a view does not. When you store a view, SQLBase stores the definition of the view in the system catalog, but SQLBase does not store any data for the view itself because the data already exists in the base table or tables.

A view lets different users view the same data in different ways. This allows programmers, database administrators, and end users to see the data as it suits their needs.

Synonyms

A *synonym* is another name for a table, view, or external function. When you access a table, view, or external function created by another user (once you have been granted the privilege), you must fully-qualify the table name by prefixing it with the owner's name, unless a synonym for the table or view is available. If one is available, you can refer to the user's table or view without having to fully qualify the name.

Stored commands and procedures

A *stored command* is a compiled query, data manipulation command, or procedure that is stored for later execution. SQLBase stores the command's or procedure's execution plan as well, so subsequent execution is very fast.

A SQLBase procedure is a set of Scalable Application Language (SAL) and SQL statements that is assigned a name, compiled, and optionally stored in a SQLBase database. Procedures reduce network traffic and simplify your applications since they are stored and processed on the server. They also provide more flexible security, allowing end users access to data which they otherwise have no privilege to access.

SQLBase procedures can be static or dynamic. *Static procedures* must be stored (at which time they are parsed and precompiled) before they are executed. *Dynamic procedures* contain dynamic embedded SQL statements, which are parsed and compiled at execution time. For this reason, they do not have to be stored before they are executed.

SQLBase also provides preconstructed procedures as useful tools to help you maintain your database. See *Appendix B* of the *Database Administrator's Guide* for a description of SQLBase-supplied procedures.

External functions

An *external function* is a user-defined function that resides in an "external" DLL (Dynamic Link Library) that is invoked within a SQLBase stored procedure. SQLBase accepts external functions in a language of your choice as C, C++, etc. The SQLBase server converts data types of parameters that are declared in stored procedures into their external representation.

Using external functions enhances the power of the SQLBase server, allowing you to achieve maximum flexibility and performance with minimal programming effort. It extends the functionality of stored procedures with no impact on the application or the server. When external functions are called, they are dynamically plugged in and behave like built-in functions. For details, read *Chapter 8*, *External Functions*.

Triggers

A *trigger* activates a stored or inline procedure that SQLBase automatically executes when a user attempts to change the data in a table. You create one or more triggers on a table, with each trigger defined to activate on a specific command (an INSERT, UPDATE, or DELETE). You can also define triggers on stored procedures.

Triggers allow actions to occur based on the value of a row before or after modification. Triggers can prevent users from making incorrect or inconsistent data changes that can jeopardize database integrity. They can also be used to implement

referential integrity constraints. For details on referential integrity, read *Chapter 6, Referential Integrity.*

For details on the trigger execution order before a single data manipulation statement is executed, read the Section *DML Execution Model* at the end of this chapter.

System catalog tables

For each database, there is a *system catalog* that contains tables created and maintained by SQLBase. These tables contain information about the tables, views, columns, indexes, synonyms, external functions, and security privileges for the database. The system catalog is sometimes called a *data dictionary.*

When you create, change, or drop a database object, SQLBase changes rows in the system catalog tables that describe the object and tell how it is related to other objects.

A system catalog contains the name, size, type, and valid values of each column stored in a table. A system catalog also holds information about the tables and views that exist in the database and how they are accessed. A user can query the data dictionary tables just like any other table.

Read the *Database Administrator's Guide* for information on the system catalog tables.

SQL command processing

There are four basic phases of SQL command processing:

1. Parse:

    • Check that the command is formulated correctly.

    • Break the statement into components for the optimizer.

    • Verify names of columns and tables in the system catalog.

2. Optimize:

    • Replace view column names and table names with real names.

    • Gather statistics on data storage from the system catalog.

    • Identify possible access paths.

    • Calculate the cost of each alternate path.

    • Choose the best path.

    For details on the SQLBase Optimizer, read the following section.

3. Generate execution code:

    • Produce an application plan for execution.

4. Execute the command.

For details on the execution model of any DML statement, read *DML Execution Model* on page *1-12*.

Optimizer

In SQLBase, you specify the data you want through a SQL command and SQLBase determines how the data will be accessed by using the *optimizer*. SQLBase chooses an access path based upon the available indexes, catalog statistics, and the composition of the SQL command.

There are several basic choices:

- **Index access without reading the data table.**

    If all the needed data is in an index, this is the most efficient access.

- **Index access in addition to reading the data table.**

    In this situation, the qualifier of the command is matched against an index and only qualified rows are read from the table. There are cases where SQLBase uses an index although data in the index does not match the data specified in the qualifier of the command.

- **Table scan.**

    All pages and rows will be read.

There are many variations of the options listed. If a query involves several tables, processing can be complex and involve internal sorting and creation of intermediate result tables which are transparent to the user.

DML Execution Model

SQLBase performs a number of validation checks before executing data manipulation statements (INSERT, UPDATE, or DELETE). Following is the execution order for data validation, trigger execution, and integrity constraint checking for a single DML statement:

1. Check for number of bind data.

2. Validate values if they are part of the statement (that is, not bound). This includes null value checking, data type checking (such as numeric), etc.

3. Perform security checks.

4. If a trigger is defined, execute BEFORE statement trigger.

5. Loop for each row affected by the SQL statement.

    For each row, perform the following actions this order

- Validate values if they are bound in. This includes null value checks, data type checking, and size checking (for example, character string too long).

  Note that size checking is performed even for values that are not bound.

- Fire BEFORE ROW trigger.

- Perform checks for duplicate values.

- Perform referential integrity checks on invoking DML.

- Execute INSERT/UPDATE/DELETE.

- Fire AFTER ROW trigger.

6. Execute AFTER statement trigger.

Note: A trigger itself can cause DML to be executed, which will apply to the steps shown in this model.

Chapter 2

SQL Elements

This chapter describes the following SQL elements:

- Names
- Data types
- Constants
- System keywords
- Database sequence objects
- Functions
- Expressions
- Predicates
- Search conditions
- Bind variables

Names

A name is called an *identifier* in SQL. User names, table names, column names, and index names are examples of identifiers.

An identifier can be an *ordinary identifier* or a *delimited identifier.*

- An ordinary identifier begins with a letter or one of the special characters (#, @ or $). The rest of the name can include letters, numeric digits, underscore (_) and special characters. An exception is a database identifier, which can only start with an alphabetic character, and contain only alphanumeric characters.

- A delimited identifier can contain *any* character including special characters such as blanks and periods. Also, a delimited identifier can start with a digit. A delimited identifier is case-sensitive.

  Delimited identifiers must be enclosed in double quotes:

  ```
  "7.g identifier"
  ```

  SQL reserved words can be used as identifiers if they are delimited, but this is *not* recommended.

  If a delimited identifier contains double quotes, then two consecutive double quotes ("") are used to represent one double quote (").

Names are *long* or *short* identifiers, or identifiers *qualified* by other identifiers. The maximum length of a long identifier is 18 characters. The maximum length of a short identifier is 8 characters.

Names of database objects (such as a table or column) are generally case-insensitive. Identifiers such as passwords or user names are usually case-sensitive. Read the following section, *Types of names* on page *2-3*, which describes the different SQLBase identifiers; a name is case-insensitive unless stated otherwise.

Note: Even though a name may be case-insensitive, it is stored in upper-case in the system catalog. For example, a query on the SYSADM.SYSTABLES table must specify the table name in uppercase, unless you enclose it in single quotes, even though you created it in lower case.

Examples of names

Examples of names are:

```
CHECKS
AMOUNT_OF_$
:CHKNUM
$500
```

```
"NAME & NO."
#CUSTOMER
:3
```

Types of names

The following objects have names:

- Authorization ID
- Columns
- Commands
- Correlations
- Databases
- External functions
- Indexes
- Passwords
- Bind Variables
- Commands
- Stored Procedures
- Synonyms
- Tables
- Triggers
- Views

Authorization ID (user name)

This is a short identifier that designates a user. Authorization ID is also called *user name* in this manual. The system keyword USER contains the user name.

An authorization ID is an implicit part of all database object names. To name a database object explicitly, add the authorization ID and a period to the beginning of the identifier. For example, the table name CUST created by user JOE has the explicit name JOE.CUST. The implicit name CUST is used most often.

A user name is case-sensitive.

Examples of authorization IDs are JOE and USER1.

Column name

This is a qualified or unqualified long identifier that names a column of a table or view.
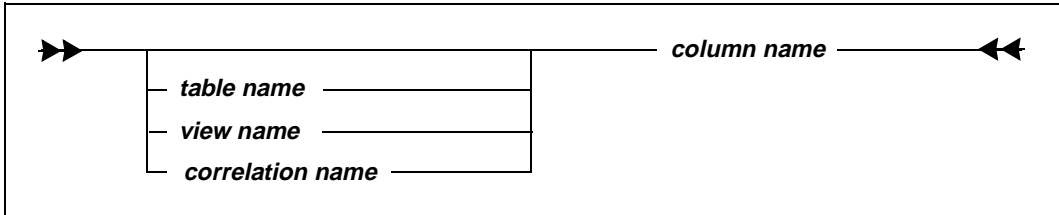
The qualified form is preceded by a table name, a view name, or correlation name and a period (.).

Examples of column names are EMPNO and EMPLOYEES.EMPNO.

Correlation name

This is a long identifier that designates a table or view within a command.

Examples of correlation names are X and TEMP.

Database name

This is a short identifier that designates a database.

Database names can only contain alphanumeric characters (A-Z, a-z, 0-9), and must start with a letter.

Do not specify an extension for a database name. For example, *demo.xyz* is invalid. SQLBase automatically assigns a database name extension of *.dbs*. SQLBase will store a database called DEMO in a file named *demo.dbs*.

Examples of database names are DEMO and COMPANY.

External function name

This is an unqualified ordinary long identifier (maximum 64 characters) that names an external function. An example is MyFunc().

A function name must start with an alpha upper or lowercase letter. It cannot be the same as procedure, or a name used in any of the SQLBase aggregate functions.

Index name

This is a qualified or unqualified long identifier that names an index.

```
►► ──────────┬─────────────────────┬──────────── index name ──────── ◄◄
             └── authorization ID ──┘
```
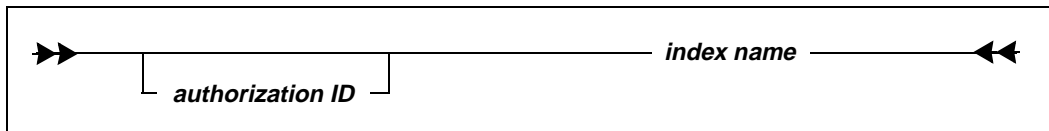
The qualified form is preceded by an authorization ID and a period.

An unqualified index name is implicitly qualified by the authorization ID of the user who gave the command.

Examples are EMPX and JOE.EMPX.

Password

This is a short identifier that is a password for an authorization ID. It is case-sensitive.

Examples are PWD1 and X2381.

Procedure name

This is a qualified or unqualified long ordinary identifier that names a procedure. An example is JOE.PROC.

A procedure name can be different from the name under which it is stored. However, a procedure name cannot be the same name as an external function name.

Bind variable name

Bind variable names in a SQL command must always be ordinary identifiers or digits preceded by a colon (:).

Command name

This is a long identifier that designates a user-defined command. An example is QUERY1.

Synonym name

This is a long identifier that designates a table or view. A synonym can be used wherever a table name or view name can be used to refer to a table or view.

An example of a synonym is EASY.

Table name

This is a qualified or unqualified long identifier that names a table.

```
▶▶─────────────────────────────────── table name ───────────────◀◀
        └─ authorization ID ─┘
```
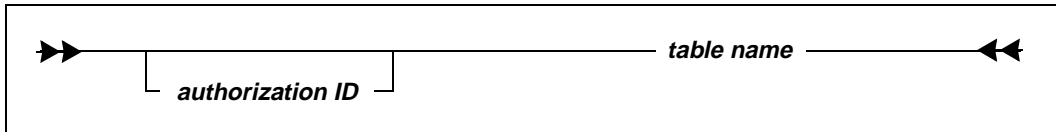
An unqualified table name is implicitly qualified by the authorization ID of the user who created the table.

The qualified form is preceded by an authorization ID and a period.

Examples are EMP and JOE.EMP.

Trigger name

This is a qualified or unqualified long ordinary identifier that names a trigger. An example is JOB_UPDT.

View name

This is a qualified or unqualified long identifier that designates a view.

```
▶▶─────────────────────────────────── view name ────────────────◀◀
        └─ authorization ID ─┘
```
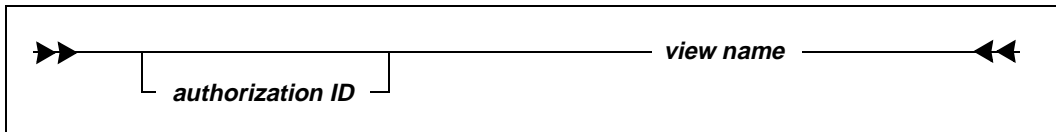
An unqualified view name is implicitly qualified by the authorization ID of the user who gave the command.

The qualified form is preceded by an authorization ID and a period.

Examples of view names are MYEMP and DEPT10.MYEMP.

Summary of naming requirements

The following table lists the naming requirements for any type of name.

Type of Identifier	Maximum Length	Qualifiable?
Authorization ID	8	No
Bind Variable	18	N/A
Column	18	Yes
Command	18	Yes

Type of Identifier	Maximum Length	Qualifiable?
Correlation	18	No
Database	8	No
External function	64	No
Index	18	Yes
Password	8	No
Procedure	18	Yes
Synonym	18	No
Table	18	Yes
Trigger	18	Yes
View	18	Yes

Data types

The general data types that SQLBase uses to store data are:

- Character
- Numeric
- Date and time

The data type determines the following information:

- The value and length of the data as stored in the database.
- The display format when the data is displayed.

The data type for a column is specified in the CREATE TABLE command.

Null values

A *null value* indicates the absence of data. Any data type can contain a null value. A null value has no storage.

Null is *not* equivalent to zero or to blank; it is the same as *unknown*. A value of null is *not* greater than, less than, or equivalent to any other value, including another value of null. To retrieve a field on a null match, the NULL predicate must be used.

NULL is equal to NULL when you insert two of them into a uniquely constrained column.

Empty strings have a null value.

Read the section, *Search conditions* on page *2-24* to understand more about how nulls are treated.

Character data types

A character string is a sequence of letters, digits, or special characters. All character data is stored in SQLBase as variable-length strings.

For DB2 SQL compatibility, SQLBase allows several alternative keywords to declare the same data type.

An empty string has a null value.

All character data types can store binary data.

Character data is stored as case-sensitive. To search for case-insensitive data, you can issue a SELECT statement with the @UPPER or @LOWER functions. For example, the following query returns only upper-case SMITHS:

```
SELECT LNAME FROM EMP
    WHERE @UPPER(LNAME) = 'SMITH';
```

CHAR (or VARCHAR)

A length *must* be specified for this data type. The length determines the maximum length of the string. The length cannot exceed 254 bytes.

You can use CHAR columns in comparison operations with other characters or numbers and, and also in most functions and expressions. Wild-card search operators can be used in the LIKE predicate for character-only comparisons.

This data type is defined in the system catalog as CHAR and VARCHAR.

Examples:

```
CHAR (11)
VARCHAR(25)
CHAR(10)
```

LONG VARCHAR (or LONG)

This data type stores strings of any length. The difference between a CHAR (VARCHAR) and a LONG (LONG VARCHAR) data type is that a LONG type can store strings longer than 254 bytes, and is not specified with a length attribute.

Both text and binary data can be stored in LONG VARCHARs. However, only character data can be retrieved through SQLTalk.

LONG VARCHAR columns can be stored, retrieved, or modified, but cannot be used in a comparison operation in a WHERE clause. LONG VARCHAR columns cannot be used in expressions or in most functions.

You can use LONG VARCHAR as a BLOBS equivalent.

You can store a bitmap file as a LONG field. SQLBase stores the entire file in the field with no compression, which means that all of the file's data is present in the database file. If the bitmap file is large, you can store it outside the database file to save space. To do this, store only the file name in the database, and use a program to access the bitmap file through its stored file name.

A LONG datatype is stored as a linked list of pages. Since it is variable length, no space is pre-allocated. This means that if no data is entered, no pages are allocated, and if data is entered, only enough pages to hold the long are allocated. However, there is a minimum allocated space of one page for non-null values. Space is allocated by page.

Example:

```
LONG VARCHAR
```

Numeric data types

SQLBase allows these numeric data types:

Exact Data Types	Approximate Data Types
DECIMAL (or DEC) INTEGER (or INT) SMALLINT	DOUBLE PRECISION NUMBER FLOAT REAL

SQLBase uses its own internal representation of numbers described in the *SQL/API Reference Manual.* Data is cast on input and output to conform to the restrictions of the data type.

Precision and scale are maintained internally by SQLBase:

- *Precision* refers to the total number of allowable digits
- *Scale* refers to the number of digits to the right of the decimal point.

Numbers with up to 15 decimal digits of precision can be stored in the exact data types. Numbers in the range of 1.0e-99 to 1.0e+99 can be stored in the approximate data types.

SQLBase supports integer arithmetic. For example:

```
INTEGER / INTEGER = INTEGER
```

Number columns can be used in any comparison operations with other numbers and can occur in all functions and expressions.

NUMBER

NUMBER is a superset of all the other numeric data types and supports the widest range of precision and scale (up to the maximum allowed by SQLBase numeric types). The NUMBER data type supports up to 22 precision digits.

Example:

```
NUMBER
```

Use NUMBER in either of the following situations:

- You do not need to control precision or whole numbers.
- You do need SQLBase to automatically give you the largest precision available.

DECIMAL (or DEC)

This data type is associated with a particular scale and precision. Scale is the number of fractional digits and precision the total number of digits. If precision and scale are not specified, SQLBase uses a default precision and scale of 5,0.

Use the DECIMAL data type when you need to control precision and scale, such as in currency.

The position of the decimal point is determined by the precision and the scale of the number. The scale, which is the number of digits in the fractional part of the number, cannot be negative or greater than the precision. The maximum precision is 15 digits.

This data type can store a maximum of 15 digits. The valid range is:

```
-999999999999999 to +999999999999999
```

Another way to express the range is to say that the value can be *-n* to *+n*, where the absolute value of *n* is the largest number that can be represented with the applicable precision and scale.

The DEC notation is compatible with DB2.

Following are some DECIMAL examples:

```
DECIMAL (8,2)
DECIMAL (5,0)        (same as INTEGER precision)
DECIMAL
DEC
```

SQLBase truncates input values to the precision of the column definition. For example:

- Entering 29.994 in a DECIMAL(10,2) stores 29.99.

- Entering 29.995 in a DECIMAL(10,2) also stores 29.99.

SQLBase truncates decimals as DB2 does with 2 exceptions:

- Floating point numbers that are used as bind variables.

- For positive numbers that contain more than 21 digits and negative numbers that contain than 19 digits, SQLBase rounds up the last digit.

Calculating precision for addition/subtraction

For two numbers A and B with precision and scale of (p1,s1) and (p2,s2) respectively, the following rules calculate the precision and scale for subtraction and division.

Precision:

Precision of result (A+B) or (A-B)	=	The minimum value of either the maximum precision of SQLBase (15) or the following equation: $\max(p1-s1, p2-s2) + \max(s1, s2) +1$

Scale:

Scale of result (A+B) or (A-B)	=	The maximum value of the two scales s1 and s2.

Calculating precision for division

For division, the following rules calculate the precision and scale of the result.

Precision:

Precision of result	=	Maximum precision of SQLBase (15)

Scale:

Scale of result	= Maximum precision	- Precision of first input number	+ Scale of first input number	- Scale of second input number

For example, if you have the following two columns:

```
D1    DECIMAL (10,2)
D2    DECIMAL (10,2)
```

and you divide D1 by D2, you get the following precision and scale:

```
precision=  15
scale=      15 - 10 + 2 - 2 = 5
```

Some functions change the maximum precision. For example, SUM changes the maximum precision to 15. Therefore, this equation:

```
SUM(D1)/SUM(D2)
```

results in the following precision and scale:

```
precision=15
scale=15 - 15 + 2 - 2 = 0
```

Calculating for multiplication

For two numbers A and B with precision and scale of (p1,s1) and (p2,s2) respectively, the following rules calculate the precision and scale.

Precision:

Precision of product (A*B)	=	The minimum value of either the maximum precision of SQLBase (15) or the sum of the precisions (p1 + p2)

Scale:

Scale of product (A*B)	=	The minimum value of either the maximum precision of SQLBase (15) or the sum of the scales (s1 + s2)

For example, if you have the following two columns:

```
D1 DECIMAL (10,2)
D2 DECIMAL (10,2)
```

and you multiply D1 by D2, then you get the following precision and scale:

```
precision =  min(15, 20) = 15
scale=       min (15, 4) = 4
```

Some functions change the maximum precision. For example, the SUM function uses the following rule:

```
precision =  min(15, max(p1-s1, p2-s2) + max(s1, s2) + 1)
scale  =  max(s1,s2)
```

So, for the following sum:

```
SUM(D1)*SUM(D2)
```

you get the following precision and scale:

```
precision=min(15, max (8, 8) + max (2,2)+ 1)= min (15, 11)=11
scale=    max(2,2) = 2
```

Currency

SQLBase does not have a specific CURRENCY data type, so you can use DECIMAL instead. A suggested setting is DECIMAL (15,2).

INTEGER (or INT)

This data type has no fractional digits. Digits to the right of the decimal point are truncated.

An INTEGER can have up to 10 digits of precision:

```
-2147483648 to +2147483647
```

The INT notation is compatible with DB2.

Examples:

```
INTEGER
INT
```

SMALLINT

This data type has no fractional digits. Digits to the right of the decimal point are truncated. Use this number type when you need whole numbers.

A SMALLINT can have up to 5 digits of precision:

```
-32768 to +32767
```

SQLBase does not store a SMALLINT value relative to the size of a 16- or 32-bit integer, but approximates it with the same number of digits. C programmers should check for overflow.

Example:

```
SMALLINT
```

DOUBLE PRECISION

This data type specifies a column containing double-precision floating point numbers.

Example:

```
DOUBLE PRECISION
```

FLOAT

This data type stores numbers of any precision and scale.

A FLOAT column can also specify a precision:

```
FLOAT (precision)
```

If the specified precision is between 1 to 21 inclusive, the format is single-precision floating point. If the precision is between 22 and 53 inclusive, the format is double-precision floating point.

Note: Although, SQLBase allows you specify a precision up to 53, the actual maximum supported precision is 22.

If the precision is omitted, double-precision is assumed.

Examples:

```
FLOAT
FLOAT (20)
FLOAT (53)
```

REAL

This data type specifies a column containing single-precision floating point numbers.

Example:

```
REAL
```

Date/Time data types

SQLBase supports these data types for date and time data:

- DATETIME (or TIMESTAMP)
- DATE
- TIME

You can use date columns in comparison operations with other dates. You can also use dates in some functions and expressions. The supported range of dates is 01-jan-0000 through 31-dec-9999.

Internally, SQLBase stores all date and time data in its own floating point format. The internal floating point value is available through an application program API call.

This format interprets a date or time as a number with the form:

```
DAY.TIME
```

DAY is a whole number that represents the number of days since December 30, 1899. December 30, 1899 is 0, December 31, 1899 is 1, and so forth.

TIME is the fractional part of the number. Zero represents 12:00 AM.

March 1, 1900 12:00:00 PM is represented by the floating point value 61.5 and March 1, 1900 12:00:00 AM is 61.0.

Anywhere a date/time string can be used in a SQL command, a corresponding floating point number can also be used.

SQLTalk and SQLBase provide extensive support for date/time values. Read the section *Date/Time values* on page *2-33* for more information.

DATETIME (or TIMESTAMP)

This data type is used for columns which contain data that represents both the date and time portions of the internal number.

You can input DATETIME data using any of the allowable date and time formats listed for the DATE and TIME data types.

When a part of an input date/time string is omitted, SQLBase supplies the default of 0, which converts to December 30, 1899 (date part) 12:00:00 AM (time part).

TIMESTAMP can be used instead of DATETIME for DB2 compatibility.

Examples:

```
DATETIME
TIMESTAMP
```

The time portion of DATETIME has resolution to the second and microsecond. The time portion of TIMESTAMP has resolution to the microsecond.

DATE

This data type stores a date value. The time portion of the internal number is 0. On output, only the date portion of the internal number is retrieved.

Example:

```
DATE
```

TIME

This data type stores a time value. The date portion of the internal number is 0. On output, only the time portion of the internal number is retrieved.

Example:

```
TIME
```

TIME has resolution to the second.

Data type conversions

This section describes how SQLBase converts data types.

Data type conversions in assignments

SQLBase is flexible in the data types it accepts for assignment operations:

Source Data Type	Target Data Type	Comment
Character	Numeric	Source value must form a valid numeric value (only digits and standard numeric editing characters).
Numeric	Character	Single quotes are not needed.
Date/Time	Numeric	
Numeric	Date/Time	
Date/Time	Character	Single quotes are not needed.
Character	Date/Time	Source value must form a valid date/time value.

Data type conversions in functions

Usually, functions accept any data type as an argument if the value conforms to the operation that function performs. SQLBase will automatically convert the value to the required data type.

For example, in functions that perform arithmetic operations, arguments can be character data types if the value forms a valid numeric value (only digits and standard numeric editing characters).

For date/time functions, an argument can be a character or numeric data type if the value forms a valid date/time value.

Constants

A constant (also called a literal) specifies a single value. Constants are classified as:

- String constants.
- Numeric constants.
- Date and time constants.

String constants

A string is a sequence of characters. A string constant must be enclosed in *single* quotes (apostrophes) when used in a SQL command.

To include a single quote in a string constant, use two adjacent single quotes.

Numeric constants

A numeric constant refers to a single numeric value.

A numeric constant is specified with digits. The value can include a leading plus or minus sign and a decimal point.

A numeric constant can be entered in E notation.

Date/Time constants

Date and time values can be used as constants. Read the section *Date/Time values* on page *2-33* for more information.

Examples of constants

Constant Type	Example	Explanation
Character String	'CHICAGO'	Character string must be enclosed in single quotes.
	'DON''T'	To include a quote character as part of a character string, use two consecutive single quotes.
	''	Two consecutive single quotes with no intervening character represents a null value.
	'1492'	If digits are enclosed in quotes, it is assumed to be a character string and not a number.
Numeric	2580	Digits not enclosed in quotes are assumed to be numeric values.
	1249.57	Numeric constant with decimal point.
	-1249	Leading plus or minus signs may be used on numerics.
	4.00E+7	E-notation can be used to express numeric values.
Date/time	10-27-94	Date/time constants do not need to be quoted.
	27-Oct-1994	

System keywords

Certain keywords have values that can be used in some commands in place of column names or constants. These special keywords are:

NULL   **The absence of a value.** NULL can be used as a constant in a select list or in a search condition. For example:

```
SELECT LNAME FROM EMP
    WHERE DEPTNO IS NULL;
```

ROWID   **The internal address of a row.** ROWID can be used instead of a column name in a select list or in a search condition.

```
SELECT ROWID FROM EMP
    WHERE HIREDATE > 01-JAN-1994;
```

USER	**The authorization ID of the current user.** USER can be specified instead of a constant in a select list or in a search condition.

```
CREATE VIEW MYTABLES AS
SELECT * FROM SYSADM.SYSTABLES
    WHERE CREATOR = USER;
```

SYSDBTRANSID	The current transaction ID of the SQL command. SYSDBTRANSID can be specified instead of a constant or column name. Read the following section Using SYSDBTRANSID keyword for more details.

SQLBase also provides these keywords:

- date/time keywords, such as:

    SYSDATETIME
    SYSDATE
    SYSTIME
    SYSTIMEZONE

    Read the Section *Date/Time values* on page *2-33* for more information.

- database sequence object keywords:

    CURRVAL
    NEXTVAL

    Read the section *Database sequence objects on page 2-20* for more information

Using SYSDBTRANSID keyword

SYSDBTRANSID is an unsigned 4-byte numeric value representing the current transaction ID under which the SQL command was executed. Like other system keywords, you can specify SYSDBTRANSID in a SQL expression in place of a constant or column name. The current transaction ID, which is the value returned by SYSDBTRANSID, remains the same throughout the life of the transaction.

For example, assume you want to "capture" and store the transaction ID associated with the following UPDATE statement:

```
UPDATE EMPLOYEES SET SALARY = 100000 WHERE NAME = 'JOHN';
```
The following INSERT statement inserts the UPDATE's transaction ID into a table called MYHISTORYTABLE:

```
INSERT INTO MYHISTORYTABLE
(transid,time,changed_by,employee_name,new_salary)
```

```
SELECT SYSDBTRANSID, SYSTIME, USER, NAME, SALARY FROM EMPLOYEES
  WHERE NAME = 'JOHN';
```

Although SYSDBTRANSID never decreases, you may not necessarily see sequential transaction IDs for sequential transactions. For instance, if you get a transaction ID of 20004 for one transaction, you may get an ID of 20010 for the next transaction, instead of 20005. This depends on the nature of the transaction; often times SQLBase has to do several internal transactions for each user transaction. The internal transactions also get their own transaction IDs. All IDs are unique.

Database sequence objects

SYSDBSequence is the name of the Database Sequence Object provided in SQLBase. A Database Sequence Object is an object inherently built into the SQLBase database that can be accessed by any database user for generating sequential numeric values. You can use sequences for automatically generating primary key values. When used as a primary key in a table, the generated sequence numbers also provide a useful way of ordering the rows in the entry sequence order.

Using SYSDBSequence

SYSDBSequence is a permanent persistent object in SQLBase. It is created when a SQLBase database is created and remains as part of the database until the database is dropped. It is persistent through reorganization of databases and can be migrated using the LOAD and UNLOAD database commands.

Initial value of the SYSDBSequence is 0 at the time of database creation and increases by 1 with no practical upper limit.

To access SYSDBSequence object values in SQL statements, use these pseudo columns:

- • NEXTVAL: Obtains the next available sequence number
- • CURRVAL: Obtains the sequence number last retrieved.

These pseudo columns let you obtain the current and incremented next value of the SYSDBSequence object as you would for regular table columns in some DML statements. Since the sequence number are generated independent of tables, they can be used across multiple tables or in general DML statement.

Although SYSDBSequence number never decrease, you may not necessarily see sequential numbers for sequential transactions. For instance, if you get a sequence number of 1000 for one transaction, you may get a number of 1005 for the next sequence, instead of 1001. This occurs since NEXTVAL is used by all transactions. Sequence numbers are always unique and ascending but not necessarily sequential.

You must qualify CURRVAL and NEXTVAL with the database sequence name SYSDBSequence. For example:

```
SYSDBSequence.CURRVAL
        or
SYSDBSequence.NEXTVAL
```

You can use SYSDBSequence by accessing its value with CURRVAL and NEXTVAL pseudo columns in these places:

- SELECT list of a SELECT statement
- VALUES clause of an INSERT statement
- SET clause of an UPDATE statement

The following semantic rules apply for the usage of sequence numbers. Note that all of the following semantics rules apply for the single execution of a SQL statement.

- First reference to NEXTVAL returns the sequence's initial value. Subsequent references to NEXTVAL increment the sequence value by 1 and returns the new value.

- Any reference to CURRVAL always returns the sequence's current value. Before you use CURRVAL for the sequence in your transaction session, you must first increment the sequence with NEXTVAL otherwise an "un-initialized currval" error will be returned.

- Once a NEXTVAL is generated, it can be accessed in the same transaction session till the next NEXTVAL is requested from that transaction session.

- One transaction session can never see the sequence number generated by another transaction session. Once a sequence number is generated by one transaction, that transaction can continue to access that value by using the CURRVAL, regardless of whether the sequence is incremented by another transaction.

- You can only increment the SYSDBSequence once in a single SQL statement.

- If a statement contains more than one reference to NEXTVAL for SYSDBSequence, SQLBase increments the sequence value once and returns the same value for all occurrences of NEXTVAL in that statement.

- If a statement contains references to both CURRVAL and NEXTVAL, SQLBase increments the sequence once and returns the same value for both CURRVAL and NEXTVAL, regardless of their order within the statement.

- Two transactions can concurrently increment the sequence; the sequence number each transaction sees may have gaps because sequence numbers can be generated by the other transactions.

Examples

This example increments the SYSDBSequence and uses its value for a new employee inserted into the employee table:

```
INSERT INTO emp
  VALUES (SYSDBSequence.nextval, 'John', SYSDATE);
```

The following example adds a new order with the next order number to the master order table and then adds suborders with this number to the detail order table.

```
INSERT INTO master_order(orderno, customer, orderdate)
  VALUES (SYSDBSequence.nextval, 'John', SYSDATE);

INSERT INTO detail_order(orderno, part, quantity)
  VALUES (SYSDBSequence.currval, 'HUBCAP', 1);

INSERT INTO detail_order(orderno, part, quantity)
  VALUES (SYSDBSequence.currval, 'SPARKPLUG', 4);

INSERT INTO detail_order(orderno, part, quantity)
  VALUES (SYSDBSequence.currval, 'MUFFLER', 1);
```

Expressions

An expression is:

- • An item that yields a single value.
- • A combination of items and operators that yield a single value.

An *item* can be any of the following:

- • A column name.
- • A constant.
- • A bind variable.
- • The result of a function.
- • A system keyword.
- • Another expression.

The form of an expression is:
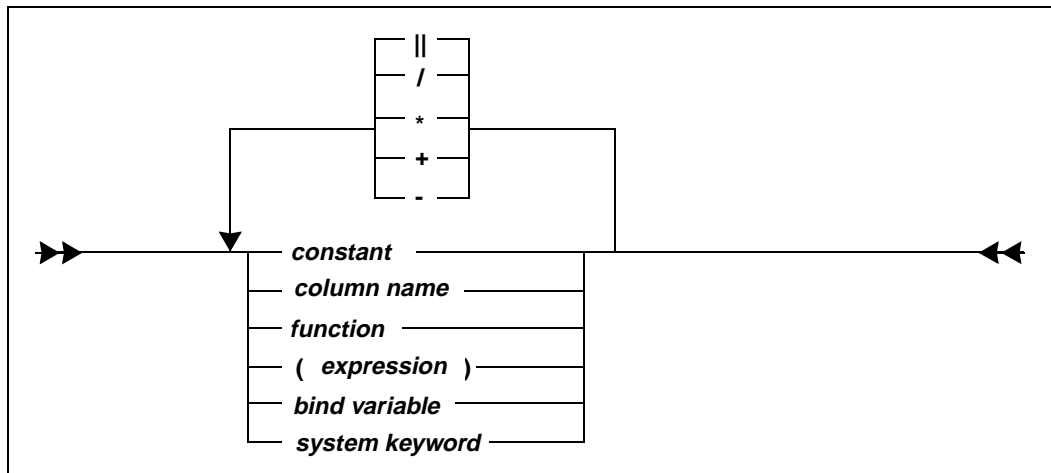
If you do not use arithmetic operators, the result of an expression is the specified value of the term. For example, the result of 1+1 is 2; the result of the expression AMT (where AMT is a column name) is the value of the column.

Null values in expressions

If any item in an expression contains a null value, then the result of evaluating the expression is null (*unknown* or false).

String concatenation operator (||)

This operator (||) concatenates two or more strings:

```
string || string
```

The result is a single string.

For example, if the column PLACE contains the value "PALO ALTO", then the following example returns the string "was born in PALO ALTO".

```
' was born in '|| PLACE
```

The following example prefixes everyone's name with "Mr.":

```
SELECT 'Mr. '||LNAME FROM EMP;
```

Precedence

The following precedence rules are used in evaluating arithmetic expressions:

- Expressions in parentheses are evaluated first.
- The unary operators (+ and -) are applied before multiplication and division.
- Multiplication and division are applied before addition and subtraction.
- Operators at the same precedence level are applied from left to right.

Examples of expressions

The following table lists some sample expressions:

AMOUNT * TAX	Column arithmetic.
(CHECKS.AMOUNT * 10) - PAST_DUE	Nested arithmetic with columns.
HIREDATE + 90	Column and constant arithmetic.
SAL + MAX(BONUS)	Function with column arithmetic.
SAL + :1	Bind variable with column arithmetic.
SYSDATETIME + 4	Date/time system keyword arithmetic.

Search conditions

A search condition in a WHERE clause qualifies the scope of a query by specifying the particular conditions that must be met. The WHERE clause can be used in these SQL commands:

- SELECT
- DELETE
- UPDATE

A search condition contains one or more *predicates* connected by the logical (Boolean) operators OR, AND, and NOT.
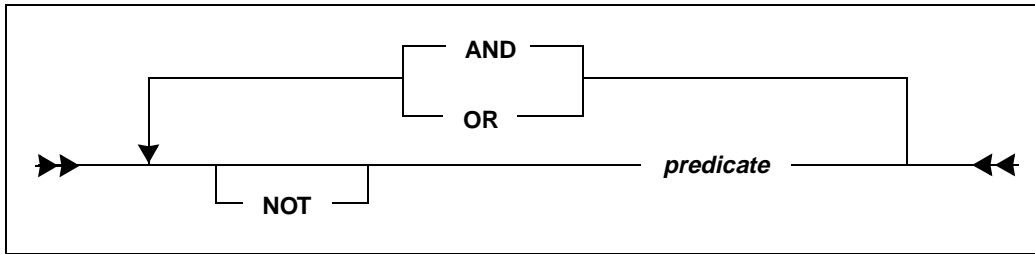
The types of predicates that can be used in a search condition are discussed in section *Predicates on page 2-27*.

The specified logical operators are applied to each predicate and the results combined according to the following rules:

- Boolean expressions within parentheses are evaluated first.

- When the order of evaluation is not specified by parentheses, then NOT is applied before AND.

- AND is applied before OR.

- Operators at the same precedence level are applied from left to right.

A search condition specifies a condition that is *true*, *false*, or *unknown* about a given row or group. NOT (true) means false, NOT (false) means true and NOT (unknown) is unknown (false). AND and OR are shown in the following truth table.

Assume P and Q are predicates. The first two columns show the conditions of the individual predicates P and Q. The next two columns show the condition when P and Q are evaluated together with the AND operator and the OR operator. If an item in an expression in a search condition is null, then the search condition is evaluated as unknown (false).

P	Q	P and Q	P or Q
True	True	True	True
True	False	False	True
True	Unknown	Unknown	True
False	True	False	True
False	False	False	False
False	Unknown	False	Unknown
Unknown	True	Unknown	True

P	Q	P and Q	P or Q
Unknown	False	False	Unknown
Unknown	Unknown	Unknown	Unknown

Using indexes with the OR predicate

SQLBase will use indexes with the OR predicate in the following situations:

- WHERE *column* IN (*literal constants*)
- WHERE *column* IN (*literal constants*) AND (*boolean expression*)
- WHERE *column operator constant1* OR *column operator constant2*..
- WHERE (*column operator constant1* OR *column operator constant2*..) AND (*boolean expression*)

Nulls and search conditions

If a search condition specifies a column that might contain a null value for one or more rows, be aware that such a row is *not* retrieved, because a null value is neither less than, equal to, nor greater than the value specified in the condition. The value of a null is *unknown* (false).

To select values from rows that contain null values, use the NULL predicate (explained later in this chapter):

```
WHERE column name IS NULL
```

SQLBase does not distinguish between a NULL and zero length string on input. Consider the following command that inserts a zero-length string:

```
INSERT INTO X VALUES ('');
```

The following command returns not only the null rows, but also the row with the zero-length string:

```
SELECT X FROM X WHERE X IS NULL;
```

Examples of search conditions

This returns rows for employees who are in department 2500.

```
SELECT * FROM EMP WHERE DEPTNO = 2500;
```

This returns rows for employees who are in department 2500 and were hired Feb. 1, 1994, or returns rows for employees who are programmers.

```
SELECT * FROM EMP WHERE (DEPTNO = 2500
    AND HIREDATE = '01-FEB-1994') OR JOB = 'Programmer';
```

The following WHERE clauses are equivalent.

```
SELECT * FROM EMP WHERE NOT
    (JOB = 'Programmer' OR HIREDATE = '01-FEB-1994');
SELECT * FROM EMP WHERE
    JOB != 'Programmer' AND HIREDATE ! = '01-FEB-1994';
```

Predicates

A predicate in a WHERE or HAVING clause specifies a search condition that is true, false, or unknown with respect to a given row or group of rows in a table.

Predicates use operators, expressions, and constants to specify the condition to be evaluated.

These types of predicates are described in this section:

- Relational
- BETWEEN
- NULL
- EXISTS
- LIKE
- IN

Relational predicate
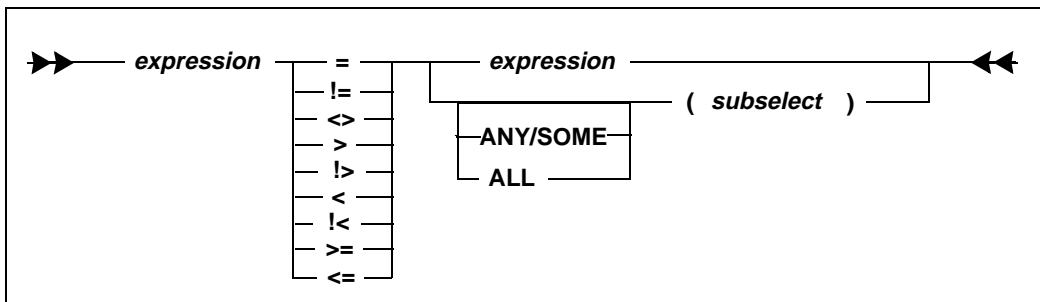
There are two types of relational predicates:

- *Comparison* relational predicate
- *Quantified* relational predicate

Comparison Relational Predicate

A comparison relational predicate compares a value to another value based on standard relational operators. The basic form of a comparison predicate is two expressions connected by a relational operator:

> A > B
>
> col1 != col2

The following are examples of comparison predicates:

```
SELECT * FROM EMP WHERE EMPNO = '50642';
SELECT * FROM EMP WHERE HIREDATE <= '1-Jan-1994';
```

Note: If you omit the keyword ALL or ANY (or SOME which can be used in place of ANY), the comparison relational predicate must return one row, or this error message is displayed: "Subselect resulted in multiple rows."

For example, if you have a table GRADES that contains a column RANK with the values 1, 2, and 3, the following statement *is not* allowed since no rows are returned:

> X >= (SELECT RANK FROM GRADES WHERE RANK >= 4)

Quantified relational predicate

A quantified relational predicate compares the first expression value to a *collection* of values which result from a subselect command.

A SELECT command that is used in a predicate is called a *subselect* or *subquery*. A subselect is a SELECT command that appears in a WHERE clause of a SQL command.

You can use the NOT operator in place of the symbol (!). For example, NOT (a=b) is the same as a!=b.

You cannot use an ORDER BY clause in a subselect. Also, you cannot use a LONG VARCHAR column in a subselect.

ANY/SOME. You can use the ANY keyword as a test with one of the comparison operators. SQLBase also allows the SOME keyword as a alternate for ANY; they are interchangeable.

The ANY test compares a single test value to a column of data values produced by the subquery. SQLBase compares the test value to each value in the column individually. If *any* of the comparisons is TRUE, the entire ANY test is TRUE.

The following table lists the rules describing results of the ANY test when the test value is compared to the column of subquery results:

Comparison Test Value	ANY search value
TRUE for at least one of the data values in the column	TRUE
FALSE for every data value in the column	FALSE
Not TRUE for any data value in the column, but is NULL for one or more of the data values.	NULL
Subquery produces empty column of query results.	FALSE

Be careful when using the ANY keyword, since it involves an entire set of comparisons, not just one. Consider the following syntax:

```
WHERE X < ANY (SELECT Y)
```

It's easy to read this line as "where X is less than any select Y". However, you should read the line as "where, for some Y, X is less than Y".

ALL. Like the ANY keyword, the ALL keyword is a quantified relational test used with the comparison operators. It compares a single test value to each data value in a column, one at a time. If all of the individual comparisons are TRUE, the entire ALL test is TRUE.

Examples of subqueries. Here are some examples of subselects and subqueries.

SALARY is not equal to the average salary:

```
SELECT * FROM EMPSAL WHERE SALARY != (SELECT
    AVG(SALARY) FROM EMPSAL);
SELECT * FROM EMPSAL WHERE SALARY <> (SELECT
    AVG(SALARY) FROM EMPSAL);
```

SALARY is greater than the average salary:

```
SELECT * FROM EMPSAL WHERE
    SALARY > (SELECT AVG(SALARY) FROM EMPSAL);
```

SALARY is less than the average salary:

```
SELECT * FROM EMPSAL WHERE
    SALARY < (SELECT AVG(SALARY) FROM EMPSAL);
```

SALARY is greater than or equal to any salary:

```
SELECT * FROM EMPSAL WHERE
    SALARY >= ANY(SELECT SALARY FROM EMPSAL);
```

BETWEEN predicate

The BETWEEN predicate compares a value with a range of values. The BETWEEN predicate is inclusive.
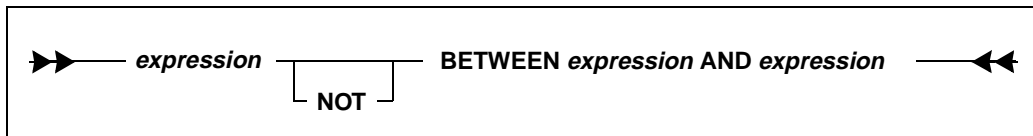
The following line shows a BETWEEN example:

```
SELECT * FROM EMPSAL WHERE
    SALARY BETWEEN 30000 AND 60000;
```

NULL predicate

The NULL predicate tests for null values.
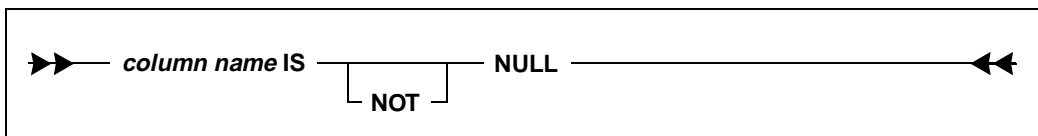
The following line shows a NULL example:

```
SELECT * FROM EMP WHERE DEPTNO IS NULL;
```

EXISTS predicate

The EXISTS predicate tests for the existence of certain rows in a table.
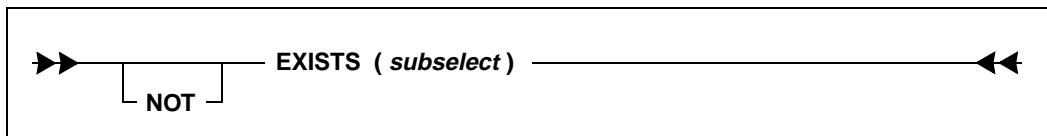
This example retrieves all the rows from the EMP table if a salary matches the value stored in bind variable :1.

```
SELECT * FROM EMP WHERE EXISTS (SELECT * FROM EMPSAL
    WHERE SALARY= :1)
    \
    70000
    /
```

LIKE predicate

The LIKE predicate searches for strings that match a specified pattern. The LIKE predicate can only be used with CHAR or VARCHAR data types.
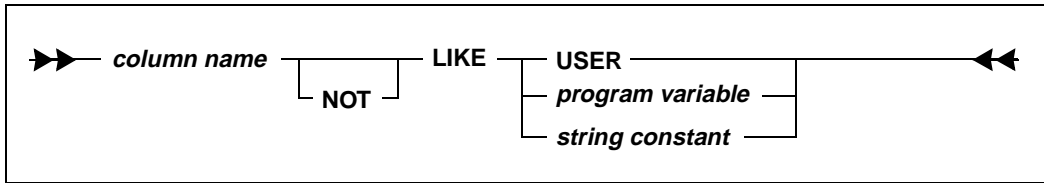
The underscore (_) and the percent (%) are the pattern-matching characters:

_	Matches *any single character.*
%	Matches *zero or more characters.*

The backslash (\) is the escape character for percent (%), underscore (_), and itself.

The following examples show examples of LIKE predicates.

True for any name with the string 'son' anywhere in it.

```
SELECT * FROM EMP WHERE LNAME LIKE '%son%';
```

True for any 2-character job code beginning with 'M'.

```
SELECT * FROM EMP WHERE JOB LIKE 'M_';
```

Returns all rows where the value in the JOB column is 'A24%'.

```
SELECT * FROM EMP WHERE JOB LIKE 'A24\%';
```

Returns all rows where the value in the JOB column begins with 'A24%'

```
SELECT * FROM EMP WHERE JOB LIKE 'A24\%%';
```

IN predicate

The IN predicate compares a value to a collection of values. The collection of values can be either listed in the command or the result of a subselect.

If there is only one item in the list of values, parentheses are not required.
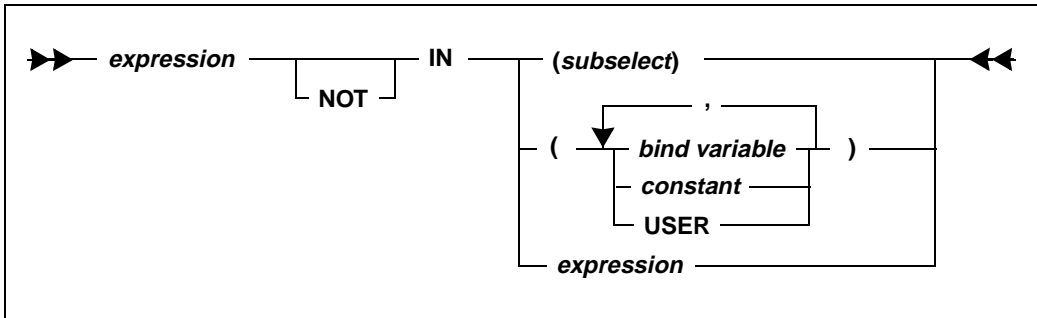
The following examples show IN predicates.

```
SELECT * FROM EMP
    WHERE DEPTNO IN (2500,2600,2700);
SELECT * FROM EMP
    WHERE EMPNO NOT IN  (SELECT EMPNO FROM EMPSAL WHERE
    SALARY< 40000);
SELECT * FROM EMP
    WHERE @LEFT (LNAME, 1) IN ('J', 'M', 'D');
SELECT * FROM EMP
    WHERE LNAME NOT IN (:1,:2,'Jones')
    \
    Johnson, Smith
    /
```

Functions

A function returns a value that is derived by applying the function to its arguments.

SQLBase has many functions that manipulate strings, dates and numbers. Functions are classified as:

- Aggregate functions
- String functions
- Date and time functions
- Logical functions
- Special functions
- Math functions
- Finance functions

The functions are described in *Chapter 4*, *SQL Function Reference*.

Date/Time values

This section describes SQL date and time values, including SQLBase year 2000 (Y2K) support.

Entering date/time values

Although SQLBase stores dates and times in its own internal format, it accepts all conventional date and time input formats, including ISO, European, and Japanese Industrial Time.

Input for a date or time column is a string that contains date or time information. The input string has a date portion and/or a time portion, depending on whether the date/time is a DATE, a TIME or a DATETIME.

A forward slash (/), hyphen (-) or period (.) are used as the delimiter for the parts of a date, as shown in the diagram on the next page. You must be consistent within a single command. A colon (:) or a period (.) are both accepted as the delimiter for times. Case is ignored by SQLBase when entering months. Either a space or a hyphen can separate the date portion from the time portion.

Letter combinations used in the formats below have the following meanings.

yy or yyyy (read the next section, Year and century values for details)	Year
mm (entered with numbers, for example, 01) mon (spelled out, for example, *jan*)	Month
dd	Day
hh	Hours
mi	Minutes
ss	Seconds
999999	Microseconds

Year and century values

SQLBase accepts date/time values in either of the following string formats:

- 4-digit string *yyyy*, which represents a 2-digit century value and a 2-digit year; for example, 1996.

- 2-digit string *yy,* which represents a 2-digit year; for example, 96.

By default, SQLBase always stores 2-digit century values as the current century. To change the default setting, you can specify 1 (one) as the value for the SQLBase keyword *centurydefaultmode* in the server section of SQL.INI. When set to 1, SQLBase applies the algorithm reflected in the following table to determine whether the year is in the current, previous, or, next century.

When last 2-digits of current year are:	When 2-digit entry is 0-49	When 2-digit entry is 50-99
0-49	The input date is in the **current** century	The input date is in the **previous** century
50-99	The input date is in the **next** century	The input date is in the **current** century

Examples:

- Assume the current year is 1996:

  If 05 is entered, the computed date is 2005
  If 89 is entered, the computed date is 1989

- Assume current year is 2014:

  If 05 is entered, the computed date is 2005
  If 34 is entered, the computed date is 2034
  If 97 is entered, the computed date is 1997

- Assume current year is 2065:

  If 05 is entered, the computed date is 2105
  If 70 is entered, the computed date is 2070

Note: Enabling the 2-digit century is a SQLBase feature and has no impact on connectivity routers. If you are using a Centura developed application or a SQL/API application against a non-SQLBase database, read the database documentation for information on how it determines year/century values.

Date/time input formats

Valid input formats for date/time values are:
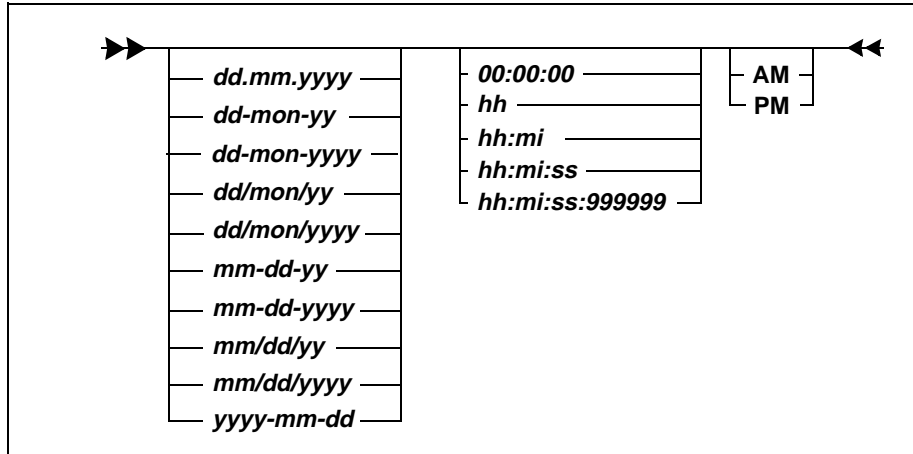
A time string can contain an AM or PM designation. The default is AM. SQLBase recognizes military time (24 hour clock) on input if the AM/PM parameter is omitted.

Some examples of date/time input strings are:

```
12-JAN-94
12/jan/1994 12:15
01-12-94 12
01/12/94 12:15:20
```

Date/time system keywords

Certain system keywords return a date/time values. These system keywords can be used in expressions to specify an interval of a specified type.

The keyword values for SYSDATETIME, SYSDATE, SYSTIME, and SYSTIMEZONE are set at the beginning of execution of a command.

The following table lists system keywords and their meaning. An asterisk (*) means that the keyword is DB2 compatible.

System Keyword	Meaning
SYSDATETIME CURRENT TIMESTAMP * CURRENT DATETIME *	Current date and time.

System Keyword	Meaning
SYSDATE CURRENT DATE *	Current date.
SYSTIME CURRENT TIME *	Current time.
SYSTIMEZONE CURRENT TIMEZONE *	Timezone interval in days. For example, SYSTIMEZONE=.025 means 6 hours.
MICROSECOND[S]	Time in microseconds.
SECOND[S]	Time in seconds.
MINUTE[S]	Time in minutes.
HOUR[S]	Time in hours.
DAY[S]	Time in days.
MONTH[S]	Time in months.
YEAR[S]	Time in years.

Resolution for time keywords

The table below show the resolution in seconds for the time keywords.

Time Keyword	Resolution
CURRENT TIME CURRENT DATETIME	Seconds (hh:mm:ss)
SYSDATE TIMESYSTIME CURRENT TIMESTAMP	Microseconds (hh:mm:ss:999999)
SECOND[S]	Seconds (ss)
MICROSECONDS	Microseconds (ss:999999)

The following command shows an example of a date/time system keyword:

```
INSERT INTO CALLS (DATE) VALUES (SYSDATETIME)
```

Time zones

The keyword SYSTIMEZONE returns the time zone for the system as a time interval in days. For example, if SYSTIMEZONE returns 0.25, the time interval is 6 hours.

The time interval is the difference between local time and Greenwich Mean Time:

```
TIMEZONE interval = LOCAL TIME - GMT
```
This interval is set with the *timezone* keyword in *sql.ini*. The default value is 0 (Greenwich Mean Time).

For instance, GMT is 5 hours later than EST (Eastern Standard Time). If the time was 5:00 A.M. EST, then

```
TIMEZONE interval = 5 - 10 = -5
TIMEZONE= -5
```

To get the current time in GMT, use the following expression:

```
(SYSTIME - SYSTIMEZONE)
```

Date/Time expressions

Addition or subtraction operators can be applied to dates. The results are as follows:

- Date + Number (of days) is DATETIME.
- Date - Number (of days) is DATETIME.
- Date - Date is a number (of days).

Note that if you add or subtract a non-date/time value to or from DATE, the result is a DATETIME. To make the result a DATE, use an expression like this:

```
Date + Number DAYS
```

where *Number* is a numeric value.

The system keywords that represent time intervals (such as MONTH or MICROSECOND) can be added to or subtracted from other date and time quantities to get new date and time quantities.

For example, the following expression yields a new DATETIME value.

```
SYSDATETIME + 3 MINUTES
```

If you do not specify the type of interval, the number is assumed to be DAYS. The following example adds one day to the current date.

```
SYSDATE + 1
```

You could also use the expression:

```
SYSDATE + 1 DAY
```

Only a constant can precede a date/time keyword.

Microseconds, seconds, minutes, hours, and days behave like numbers. MONTH and YEAR intervals however, are special cases since they do not have a fixed value in terms of the number of days in the month or year. February has 28 or 29 days, March has 30; a year can be 365 or 366 days.

Use the following rules for MONTH and YEAR intervals:

- MONTH and YEAR intervals can only be added to or subtracted from a DATE or a DATETIME quantity.

  ```
  Valid: (SYSDATE + 3 DAYS) + 1 YEAR
  Invalid:SYSDATE + (3 DAYS + 1 YEAR)
  ```

- When MONTHs are added, the month number (and if necessary the year number) is incremented. If the day represents a day beyond the last valid day for the month and year, it is adjusted to be a valid date.

- SQLBase ignores fractional parts of MONTHs and YEARs. For example, SQLBase would ignore the fraction part .5 of MONTHS in the following command:

  ```
  SELECT DISTINCT SYSDATETIME, SYSDATETIME + 1.5
    MONTHS FROM SYSTABLES
  ```

Examples of date/time expressions

The following table lists some sample date/time expressions and their results:

Date/Time Expression	Result
31-Jan-1993 + 1 MONTH	28-Feb-1993
20-Jan-1993 + 1 MONTH	20-Feb-1993
3 1-Jan-1993 + 1 MONTH - 1 MONTH	28-Jan-1993

Joins

A *join* pulls data from different tables and compares it by matching on a common row that is in all the tables.

You cannot perform an operation with the CURRENT OF clause on a result set that you formed with a join.

The following example demonstrates a join.

Example:

The primary key for a table is a value that has a match in another table. For example, the following CUSTOMER table contains these columns: name and address. Also, each customer has a unique identifying number.

CUSTNO	NAME	ADDRESS
1	ABC INC.	13 A St.
2	XYZ INC.	1 B St.
3	A1 INC.	12 C St.

There is another table called ORDERS that contains the order number, order date, and sales rep for each order. The table also includes a key that contains the customer number. This is the same number that is in the CUSTOMER table.

CUSTNO	ORDERNO	ORDERDATE	SALES REP
1	3001	01-JUL-94	Jill
1	3002	03-JUL-94	Jill
1	3003	06-JUL-94	Tom
2	3004	06-JUL-94	Tom
3	3005	07-JUL-94	Jill

You can *join* customer information with order information without unnecessary data repetition.

The following SQL command uses these tables to find the name and order numbers of the sales made by Tom.

```
SELECT NAME, ORDERNO FROM
    CUSTOMER, ORDERS WHERE
    CUSTOMER.CUSTNO = ORDERS.CUSTNO
    AND SALES REP = 'Tom';
```

This produces the following result:

NAME	ORDERNO
ABC Inc.	3003
XYZ Inc.	3004

Types of joins

SQLBase supports the following types of joins:

- Equijoins
- Outer joins
- Self joins
- Non-equijoins

Equijoin

The following query matches customer names and order numbers. Two tables are used: CUSTOMER and ORDERS.

```
SELECT NAME, ORDERNO FROM CUSTOMER, ORDERS
    WHERE CUSTOMER.CUSTNO = ORDERS.CUSTNO;
```

Each result row contains the customer name and an order number. If customer number 1 made three orders, three rows would result. The single customer row containing the customer's name and number would be "joined" to each of the three order rows.

The ORDERS rows are related to the CUSTOMER using the key column, CUSTNO, which appears in both the CUSTOMER table and the ORDERS table.

This type of search condition, which specifies a relationship between two tables based on an *equality*, is called an equijoin.

Cartesian product

Specifying a join condition as a relational predicate in the search condition is necessary to avoid a Cartesian product. A Cartesian product is the set of all possible rows resulting from a join of more than one table. For example, suppose we specified the previous query as follows:

```
SELECT NAME, ORDERNO FROM CUSTOMER, ORDERS;
```

The result would be the product of the number of rows in the customer table and the number of rows in the orders table. If CUSTOMER had 100 rows, and ORDERS had

500 rows, the Cartesian product would be every possible combination, or 50,000 rows, which is probably not the desired result.

The correct way to get each customer and order listed (a set of 500 rows) is with an equijoin, as follows:

```
SELECT NAME, ORDERNO FROM CUSTOMER, ORDERS
    WHERE CUSTOMER.CUSTNO = ORDERS.CUSTNO;
```

Outer join

In the previous example of the equijoin, the search condition specified a join on customers and orders. What happens if customer NEWACCOUNT has not yet made an order? The above query does not retrieve that customer.

An outer join produces a result that joins each row of one table with either a matching row or a null row of another table. The result includes *all* the rows of one table regardless of whether they have a match with any of the rows of the table to which they are being joined.

Outer join semantics. In the WHERE clause, add a plus sign (+) to the join column of the table that might *not* have rows to satisfy the join condition.

SQLBase supports an outer join on only one table per SELECT statement, and it must be a *one-way* outer join. You cannot add the plus sign (+) to both sides of the join condition. You can, however, specify an outer join on more than one column of the same table, like this example:

```
SELECT t1.col1, t2.col1, t1.col2, t2.col2
FROM t1, t2
WHERE t1.col1 = t2.col1 (+)
AND t1.col2 = t2.col2 (+);
```

The next example lists customer names and their order numbers, including customers who have made no orders.

```
SELECT CUSTOMER.CUSTNO, NAME FROM CUSTOMER, ORDERS
WHERE CUSTOMER.CUSTNO = ORDERS.CUSTNO(+);
```

When SQLBase sees the plus sign (+) after ORDERS.CUSTNO, it temporarily adds an extra row containing all null values to the ORDERS table. SQLBase then joins this null row to rows in the CUSTOMER table which do not have matching orders. Therefore, all customer numbers are retrieved.

SQLBase adheres to both the ANSI and industry standard implementation of an outer join. According to the ANSI standard, the correct semantics of an outer join must display all the rows of one table that meet the specified constraints on that table, regardless of the constraints on the other table.

Oracle Outer Join. If you need to use the Oracle-style outer join result, you can specify the *oracleouterjoin* keyword in the relevant server section of your *sql.ini* file. For example, if you are using the SQLBase Server for Windows NT, specify *oracleouterjoin* in the [dbntsrv] section:

```
[dbntsrv]
oracleouterjoin=1
```

The following example shows how the two standards differ in output. These examples use the following tables and SELECT statement:

```
Table A (a int)    Table B (b int)
---------------    ---------------
1                  1
2                  2
3                  3
4
5

SELECT a, b
  FROM A, B
  WHERE A.a = B.b (+)
  AND B.b IS NULL;
```

The ANSI standard gives the following result:

```
a     b
---   ---
1
2
3
4
5
```

Using the ORACLE style outer join yields a different result:

```
a     b
---   ---
4
5
```

Self join

A self join lets you join a table to itself, as though it was two separate tables. To do this, the self-join table is given a correlation name. The example below finds all dates on which more than one order was placed:

```
SELECT A.ORDERNO, A.ORDERDATE
  FROM ORDERS A, ORDERS B
  WHERE A.ORDERDATE = B.ORDERDATE
  AND A.ORDERDATE <> B.ORDERNO;
```

The ORDERS table is treated as two tables, using the correlation names A and B. An order date is retrieved from correlation table A. Then this order date is used as a search condition for table B.

This same information can be retrieved using a subquery. Read the following section *Subqueries* on page *2-43* for more information.

Non-equijoin

A non-equijoin joins tables to one another based on comparisons other than equality. Any of the relational operators can be used, (such as >, <, !=, BETWEEN, or LIKE).

The following example specifies a join using the BETWEEN operator.

```
SELECT NAME, ORDERNO, ORDERDATE
  FROM CUSTOMER, ORDERS
  WHERE CUSTOMER.CUSTNO = ORDERS.CUSTNO
  AND ORDERDATE BETWEEN 01-JUL-94 AND 30-SEP-94;
```

Number of joins

SQLBase allows you to join 10 tables in a select statement. However, think carefully before you join this many tables. With each table added in a JOIN, the time needed to process the statement increases. Having 10 tables in a join can slow the database performance considerably.

A carefully designed database model should rarely need to join 10 tables in a statement. Instead of using this many tables, reconsider your database design.

Subqueries

SQL is recursive. The input to one query can be the output of another query. A query can be nested within another SQL command to define the scope of a command. This nested query is called a *subquery*.

A subquery is a search condition that is a nested SELECT command (also called a subselect). The subquery specifies a result table from one or more tables or views, in the same manner as any other SELECT. Each result row of the subselect is used as a basis for qualifying a candidate result row in the outer select.

You cannot use an ORDER BY clause in a subquery. Also, you cannot use a LONG VARCHAR in a subquery.

Examples of subqueries

For example, find all the orders that were placed on the same day as the order from customer number 2.

```
SELECT ORDERNO, ORDERDATE FROM ORDERS
  WHERE ORDERDATE = (SELECT ORDERDATE FROM ORDERS
  WHERE CUSTNO = 2);
```

First, the order date of customer number 2 is retrieved and this value is used to complete the search condition of the main or outer query. In this example, the subquery was executed once to retrieve a single value used by the main query. In the following SELECT command, called a correlated subquery, the subquery is executed once for each candidate row in the main query.

For example, find all employees whose salary is larger than the average salary of other employees.

```
SELECT * FROM EMPSAL WHERE
  SALARY (SELECT AVG (SALARY) FROM EMPSAL);
```

Bind variables

A bind variable refers to a data value associated with a SQL command. Bind variables associate (bind) a syntactic location in a SQL command with a data value that is to be used in that command.

Bind variables can be used wherever a data value is allowed:

- WHERE clause.
- VALUES clause in an INSERT command.
- SET clause of in UPDATE command.

Bind variable names start with a colon (:). The name can be:

- The name of a variable that is declared in a program (such as :ARG1).
- A number that refers to the relative position among the data items associated with the SQL command (such as :1, :2, :3).

Read the manual for the client application that you are using (such as the Centura's SQL/API, SQLTalk, or Team Developer) for the details on how to use bind variables.

Chapter 3

SQL Command Reference

This chapter contains the syntax, description, and examples of each SQL command. This chapter is organized alphabetically by command name.

SQL command summary

Command	Function
ALTER DATABASE	Changes storage group or log for database.
ALTER DBAREA	Changes the size of a database area.
ALTER EXTERNAL FUNCTION	Changes an external function definition.
ALTER PASSWORD	Changes a password.
ALTER STOGROUP	Adds or drops a database area from a storage group.
ALTER TABLE	Changes the description of a table.
ALTER TABLE (error messages)	Makes error messages specific to a particular referential integrity violation.
ALTER TABLE (referential integrity)	Adds or drops primary and foreign keys.
ALTER TRIGGER	Enables and disables triggers defined on tables.
AUDIT MESSAGE	Writes a message string to an audit file.
CHECK DATABASE	Checks database for integrity.
CHECK INDEX	Checks specified index for integrity.
CHECK TABLE	Checks specified table for integrity.
COMMENT ON	Replaces or adds a comment to the description of a table, view, column, or external function in the system catalog.
COMMIT	Ends a logical unit of work and commits database changes made by it.
CREATE DATABASE	Physically creates a database.
CREATE DBAREA	Creates a database area.
CREATE EXTERNAL FUNCTION	Creates an external function.
CREATE INDEX	Creates an index on a table.
CREATE STOGROUP	Creates a storage group.

Command	Function
CREATE SYNONYM	Defines an alternate name for a table, view, or external function.
CREATE TABLE	Defines a table.
CREATE TRIGGER	Creates a trigger.
CREATE VIEW	Defines a view of one or more tables or views.
DBATTRIBUTE	Sets database-specific attributes
DEINSTALL DATABASE	Takes a database off the network, making it unavailable to users.
DELETE	Deletes one or more rows from a table.
DROP DATABASE	Physically deletes a database.
DROP DBAREA	Physically deletes a database area.
DROP EXTERNAL FUNCTION	Deletes an external function.
DROP INDEX	Removes an index.
DROP STOGROUP	Deletes a storage group.
DROP SYNONYM	Deletes a synonym.
DROP TABLE	Physically deletes table from the database.
DROP TRIGGER	Deletes a trigger.
DROP VIEW	Deletes a view.
GRANT (database authority)	Grants database authority or privileges.
GRANT (table privileges)	Grants one or more specified privileges for a table or view.
GRANT EXECUTE ON	Grants execute privilege on stored procedures and external functions to other users.
INSERT	Inserts one or more rows into an existing table.
INSTALL DATABASE	Puts a database on the network, making it accessible to users.

Command	Function
LABEL	Adds or changes labels in catalog descriptions
LOAD	Loads one or more tables into a database.
LOCK DATABASE	Places an exclusive lock on the database, preventing connections from other users.
PROCEDURE:	Creates a procedure.
REVOKE	Revokes database authority or privileges.
REVOKE EXECUTE ON	Revokes a user's execute privilege on a stored procedure or external function.
ROLLBACK	Terminates a logical unit of work and backs out database changes made during the last transaction.
ROWCOUNT	Counts the number of rows in a table.
SAVEPOINT	Assigns a checkpoint within a transaction.
SELECT	Queries tables or views.
SET DEFAULT STOGROUP	Specifies the default storage group.
START AUDIT	Starts a database audit.
STOP AUDIT	Stops a database audit.
UNLOAD	Unloads a database to an external file.
UNLOCK DATABASE	Releases the exclusive lock on the database from the LOCK DATABASE command.
UPDATE	Updates the values of columns in a table or view.
UPDATE STATISTICS	Updates the statistics for an index in a table.

ALTER DATABASE

```
>>── ALTER DATABASE database name ──┬── STOGROUP ──┬── stogroup name ──◄◄
                                    └── LOG ───────┘
```

This command changes the storage group for a database or its log files. ALTER DATABASE only affects future allocations of space. Existing databases or log files are not moved or affected.

Clauses

database name
The name of the database to be changed.

STOGROUP
Changes the storage group for a database.

LOG
Changes the storage group for the database's log files.

stogroup name
The name of the storage group to be changed. A storage group is a list of database areas, which are similar to files or a partition.

Example

```
ALTER DATABASE ACCTPAY STOGROUP ACCTDEPT;
```

See also

CREATE DATABASE
CREATE STOGROUP
DELETE STOGROUP

ALTER DBAREA

```
▶▶─ ALTER DBAREA dbarea name ──── SIZE megabytes ─────────────◀◀
```

This command changes the size of a database area. When increasing the size of a database area, available space is checked at the time of the operation.

Clauses

dbarea name
The name of the database area to be changed.

SIZE megabytes
The size of the database area in megabytes.

Example

```
ALTER DBAREA ACCT1 SIZE 10;
```

See also

ALTER DATABASE
CREATE DBAREA
DROP DBAREA

ALTER EXTERNAL FUNCTION

```
▶▶─ ALTER EXTERNAL FUNCTION function name ──────────────────▶

▶── ┬─ LIBRARY ────── library-name ──────┬──────────────◀◀
    ├─ EXTERNAL NAME ── external-name ─────┤
    ├─ CALLSTYLE ─┬─ PASCAL/STDCALL ─┬─────┤
    │             └─ CDECL ──────────┘
    └─ EXECUTE IN ─┬─ SEPARATE PROCESS ─┬──
                   └─ SAME THREAD ──────┘
```

Use this command to alter an external function. You must have DBA authority to execute this command.

This command allows you to alter those properties of an external function that do not invalidate dependent objects. Those properties are library name, external name, callstyle, and execution mode.

Each clause in this command is optional, but you must at least specify one clause.

Clauses

function name

Specify the name of the external function that you want to alter. This is the name that refers to the function within SQLBase.

LIBRARY library-name

Specify the dynamic linked library (DLL) name if you want to change the existing library name where the function resides. You must provide a fully qualified path name for the file, or else be sure the PATH environment variable is set to point to the location of the file in your operating system.

Specify the library name as a string with up to 254 characters. You can include special characters in the string. If the library name contains spaces, you must delimit the name in single quotes (for example, 'lib name').

EXTERNAL external-name

Specify this clause if you want to change the external name or provide an external name for the function. An external name lets you create a function name that references the function in a DLL by another name. Thus, the function has a calling name that is separate from the name used to reference the same function in the DLL.

Specify the external name as a string with up to 254 characters. You can include special characters in the string. The external name is case-sensitive and must be identical to the exported function name in the DLL.

Note that if you do not supply an external name, the function name is the same name that is used in the DLL.

CALLSTYLE

Specify this clause if you want to change the compiler style that is required to invoke the external function. For details, read *Chapter 9*, *External Functions*.

Note: Be sure to specify the correct callstyle for your platform. An incorrect callstyle can result in server failure.

PASCAL/STDCALL

PASCAL applies only to 16-bit platforms and is the call style for Windows API calls. STDCALL applies only to 32-bit platforms and is the call style for all 32-bit Windows API calls.

CDECL

This is the default compiler callstyle and applies to both 16-bit and 32-bit platforms.

EXECUTE IN

Specify this clause only if you are using a 32-bit platform and want to change the execution mode to SAME THREAD or SEPARATE PROCESS.

For details on execution mode, read *Chapter 9*, *External Functions*.

Examples

```
CREATE EXTERNAL FUNCTION MYFUNC
    LIBRARY TEST.DLL;
ALTER EXTERNAL FUNCTION MYFUNC
    LIBRARY MYFUNC.DLL;
```

See also

CREATE EXTERNAL FUNCTION
DROP EXTERNAL FUNCTION

ALTER PASSWORD

ALTER PASSWORD *old password* ———— **TO *new password***

This command changes your password.

The password is stored in the system catalog and can be read by a user with SYSADM or DBA privileges. Note that passwords are encrypted when transmitted across a network.

Clauses

old password
Your current password.

TO new password
The new password you wish to implement.

Examples

```
ALTER PASSWORD OLDSTUFF TO NEWSTUFF;
```

See also

GRANT
REVOKE

ALTER STOGROUP
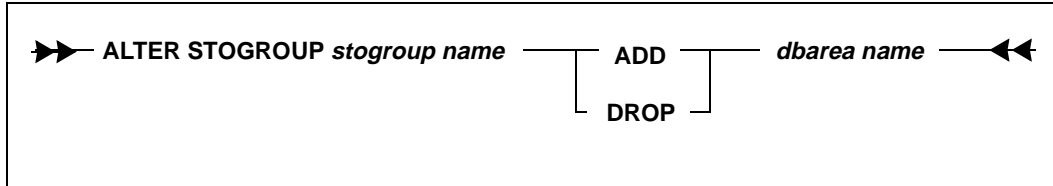
This command changes the storage group for a database area. The command only affects future allocations of space. Existing databases or log files are not moved or affected.

Clauses

stogroup name

A storage group is a list of database areas.

ADD dbarea name

Adds a database area to the storage group.

DROP dbarea name

Drops a database area from the storage group.

Example

```
ALTER STOGROUP ACCTDEPT ADD ACCT4;
ALTER STOGROUP ACCTDEPT DROP ACCT4;
```

See also

CREATE STOGROUP
DROP STOGROUP

ALTER TABLE
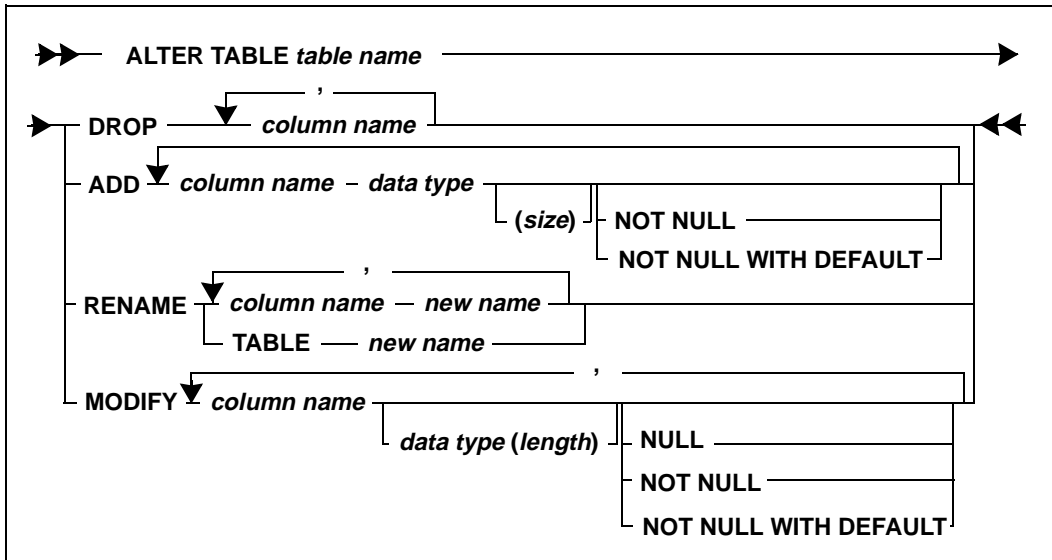
Use this command to perform the following functions:

- Add, drop, or modify a column.
- Rename a column or table.

Views that reference dropped or renamed columns or tables are automatically dropped. Views that reference modified columns are also dropped.

Precompiled commands that reference dropped or renamed columns or tables are not dropped. Such precompiled commands could become invalid and return an error if executed.

You must have the ALTER privilege on the table to execute this command.

You cannot alter tables that have triggers defined on them. If you need to do this, you must drop the triggers, alter the table, and then create the triggers for the table.

Like all DDL commands, this command locks system tables while executing.

Clauses

ADD

This adds a column to a table. Columns are defined the same way as in the CREATE TABLE command. Read the section *Data types* on page *2-7* for more information.

Adding a column does not effect existing views or precompiled commands.

You can add columns to user tables or to system catalog tables. However, if you add columns to system catalog tables, they are not maintained by SQLBase. For example, UNLOAD will ignore any user-defined columns in a system catalog table.

ADD is the default clause if no clause is specified.

DROP

This removes a column from a table. If the column has data, the data is lost.

You cannot drop any of the following:

- An indexed column.
- A column belonging to a primary or foreign key.
- System defined columns in the system catalog.

MODIFY

This changes the attributes for a column.

You can increase the length of a character column, but you cannot decrease the length. You specify the data type when you increase the length of a character column.

You *cannot* change the data type of a column.

You *cannot* change the length of a numeric column.

NULL

This removes a NOT NULL attribute for a column.

NOT NULL

This adds a NOT NULL attribute to a column that currently accepts nulls.

If the column contains NULL values, you cannot redefine the column as NOT NULL.

You cannot modify system-defined columns in the system catalog.

NOT NULL WITH DEFAULT

This clause prevents a column from containing null values and allows a default value other than the null value. The default value used depends on the data type of the column, as follows:

Data Type	Default Value
Numeric	0 (zero)
Date/Time	Current date/time
Character	Blank

The NOT NULL WITH DEFAULT clause causes an INSERT to use the above defaults. SQLBase puts a 'D' in the NULLS columns of the SYSCOLUMNS table and treats it like a NOT NULL field.

The ALTER TABLE command *does not* allow the addition of a column defined as NOT NULL if rows of data already exist. The ALTER TABLE command *does* allow a new column defined as NOT NULL WITH DEFAULT to be added if no rows of data exist for the specified table.

To add columns defined as NOT NULL or NOT NULL WITH DEFAULT when rows of data already exist, do the following steps:

1. Add the column with ALTER TABLE, but do *not* specify a NOT NULL or NOT NULL WITH DEFAULT clause.

2. Update the values in the new column to some value other than NULL.

3. Change the column to NOT NULL or NOT NULL WITH DEFAULT with the ALTER TABLE command.

The NOT NULL WITH DEFAULT clause is compatible with DB2.

RENAME

This renames a table or column. System catalog tables and system-defined columns in the system catalog cannot be renamed.

Examples

Add a new column called JOB that contains a maximum of 20 characters to the EMP table:

```
ALTER TABLE EMP ADD JOB VARCHAR(15);
```

Increase the size of the column JOB to 40 characters and make it a NOT NULL column:

```
ALTER TABLE EMP
MODIFY JOB VARCHAR(40) NOT NULL;
```

Drop the columns JOB and HIREDATE.

```
ALTER TABLE EMP DROP JOB, HIREDATE;
```

Change the name of EMP to EMPLOYEE.

```
ALTER TABLE EMP RENAME TABLE EMPLOYEE;
```

Add the NOT NULL attribute to the HIREDATE column:

```
ALTER TABLE EMP MODIFY HIREDATE NOT NULL;
```

Now drop the NOT NULL attribute:

```
ALTER TABLE EMP MODIFY HIREDATE NULL;
```

Drop the primary key:

```
ALTER TABLE EMP DROP PRIMARY KEY;
```

See also

CREATE TABLE
DROP TABLE

ALTER TABLE (Error Messages)
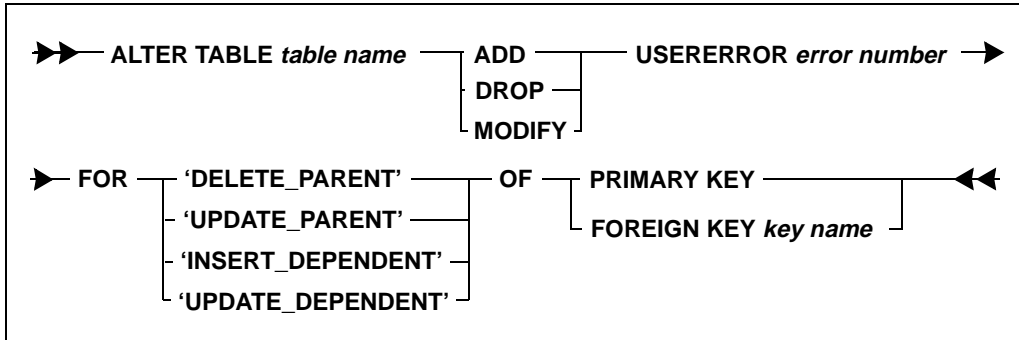
To make error messages specific to a particular violation of referential integrity, you can edit the *error.sql* file and use ALTER TABLE statements. For more information, read *Chapter 6, Referential Integrity*.

Clauses

ADD

This adds a specific error message for referential integrity. You must include an error number from the *error.sql* file.

DROP

This deletes a specific error message. Do not enter the user error number for a DROP command.

MODIFY

This modifies the error message number for referential integrity.

USERERROR error number

This specifies the error number in the *error.sql* file. You can modify the *error.sql* file to add an appropriate error message.

'DELETE_PARENT'

This specifies that a deletion failed because there were dependent rows in the dependent table.

'UPDATE_PARENT'

This specifies that an update failed because there were dependent rows in the dependent table (dependent on the values to be updated).

'INSERT_DEPENDENT'

This specifies that an insertion failed because there was no parent row in the parent table.

'UPDATE_DEPENDENT'

This specifies that an update failed because there was no parent row in the parent table for the new set of values.

Example

A user may attempt to delete the employee number of an employee who still works for the company. You can avoid this problem by editing the *error.sql* file:

```
20000 xxx xxx Employee number cannot be deleted while employee
still works for this company.
```

Then, use the ALTER TABLE statement to add the new error message:

```
ALTER TABLE EMP ADD USERERROR 20000 FOR 'DELETE_PARENT' OF
PRIMARY KEY;
```

If a user now attempts to delete the employee number, the new error message appears:

```
DELETE FROM EMP where EMPNO = 1234;


Error: Employee number cannot be deleted while employee still
works for this company.
```

See also

CREATE TABLE

ALTER TABLE (Referential Integrity)
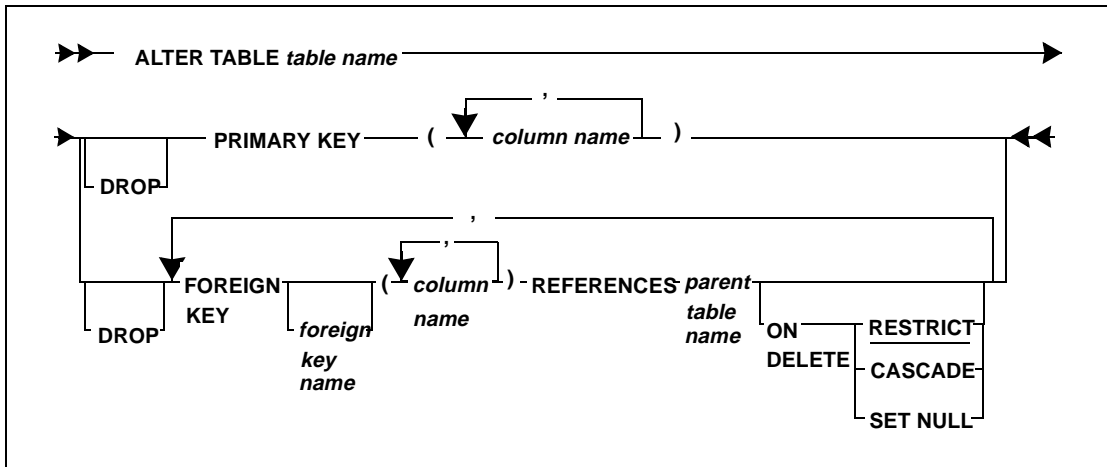
When you use ALTER TABLE with referential constraints, you can add or drop primary and foreign keys. For more information, read *Chapter 6, Referential Integrity*.

Clauses

(ADD)/DROP

You do not specify ADD since it is the default, but you must specify DROP if this is your intention.

Before dropping the primary key, consider the effect this will have on the application programs. The programs must then enforce referential constraints without the primary key.

PRIMARY KEY

In a database with referential integrity, this adds or drops the primary key of the table. If you drop the primary key, the table continues to exist with a unique index on the same list of columns (if the table has a unique index). The relationship between the tables is dropped if the table has a dependent.

To drop the primary key, you must have the ALTER privilege on both the parent and dependent tables.

The following rules apply to primary keys:

- If a table has a primary key, you must also create a unique index on the primary key columns to make the table complete. See the CREATE INDEX command for more information.

- The primary key format must obey the following rules:

    - Cannot contain more than 16 columns.

    - Sum of the column length attributes cannot be greater than 255 bytes.

    - Cannot contain LONG or LONG VARCHAR columns.

- You cannot use an UPDATE WHERE CURRENT clause with a primary key column.

- In a self-referencing row, you cannot update the primary key value. If a row is a *self-referencing row,* its foreign key value is the same as its primary key value.

- The values of the primary key must be unique; no two rows of a table can have the same key values.

- A table can have only one primary key.

- The primary key can be made up of one or more columns in a table. This is called a composite primary key. Separate the columns with a comma.

- Each column in the primary key must be classified with the NOT NULL constraint. However, you should not use the NOT NULL WITH DEFAULT option unless the primary key column(s) has a data type of TIMESTAMP or DATETIME.

- An updateable view defined on a table with a primary key must include all columns of the primary key. Although this is only required if you use the view in an INSERT statement, the resulting unique identification of rows is also useful if the view is used for updating, deleting, or selecting.

    If you try to insert a row into a view that does not contain values for all of the primary key columns, the following message appears:

    ```
    NOT ENOUGH NON-NULL VALUES
    ```

    This message appears because all the primary key columns are defined as NOT NULL (since a primary key cannot contain NULL values).

FOREIGN KEY

This adds or drops a foreign key to an existing table. The values in the foreign and primary keys must conform with referential integrity. Otherwise, the command is rejected. To drop a foreign key, you must have the ALTER privilege on both the parent and dependent tables.

Before you drop a foreign key, consider carefully the effect this will have on application programs. Dropping a foreign key drops the corresponding referential relationship and delete rule. Without the foreign key, programs must enforce these constraints.

The following rules apply to foreign keys:

- **Matching columns.** A foreign key must contain the same number of columns as the primary key. The data types of the foreign key columns must match those of the primary key on a one-to-one basis, and the matching columns must be in the same order.

  However, the foreign key can have different column names and default values. It can also have NULL attributes. If an index is defined on the foreign key columns, the index columns can be in ascending or descending order, which may be different from the order of the primary key index.

- **Using primary key columns.** A column can belong to both a primary and foreign key.

- **Foreign keys per table.** A table can have any number of foreign keys.

- **Number of foreign keys.** A column can belong to more than one foreign key.

- **Number of columns.** A foreign key cannot contain more than 16 columns.

- **Parent table.** A foreign key can only reference a primary key in its parent table. This parent table must reside in the same database as the foreign key.

- **NULL values.** A foreign key column value can be NULL. A foreign key value is NULL if any column in the foreign key is NULL.

- **Privileges.** You must grant ALTER authority on a table to all users who need to define that table as the parent of a foreign key.

- **System catalog table.** The foreign key cannot reference a system catalog table.

- **Views.** A foreign key cannot reference a view.

- **Self-referencing row.** In a self-referencing row, the foreign key value can only be updated if it references a valid primary key value. If a row is a *self-referencing row,* its foreign key value is the same as its primary key value.

foreign key name

You can assign a name to the foreign key to identify it. This name is called a constraint name. If you do not specify a name yourself, SQLBase generates a constraint name from the name of the first foreign key column.

A foreign key constraint name can have up to 18 characters. This means that if the first foreign key column name is more than 18 characters, you must assign a constraint name yourself that does violate this limit. Otherwise, SQLBase will not create the foreign key.

If there are multiple foreign keys referencing the same table, each foreign key must have a unique name. This ensures that every referential constraint is uniquely identified by a table name/constraint name combination.

REFERENCES

This identifies the parent table in a relationship and defines the necessary constraints. The REFERENCES clause must accompany the FOREIGN KEY clause.

ON DELETE

This specifies the DELETE rules for the table.

The DELETE rules are optional.

The default is RESTRICT.

DELETE rules are only used to define a foreign key.

CASCADE

This deletes the selected rows first, and then deletes the dependent rows, honoring the delete rules of their dependents.

RESTRICT

This specifies that a row can be deleted if no other row depends on it. If a dependent row exists in the relationship, the delete will fail.

SET NULL

This specifies that for any delete performed on the primary key, matching values in the foreign key are set to null.

Examples

Add a foreign key to the EMPSAL table:

```
ALTER TABLE EMPSAL
FOREIGN KEY (EMPNO) REFERENCES EMP
ON DELETE RESTRICT;
```

Add a primary key to the EMPLOYEE table:

```
ALTER TABLE EMP
PRIMARY KEY (EMPNO);
```

See also

CREATE TABLE

ALTER TRIGGER

▶▶ **ALTER TRIGGER** *trigger name* ── **ENABLE** ─────────────────── ◀◀
　　　　　　　　　　　　　　　　 └─ **DISABLE** ─┘
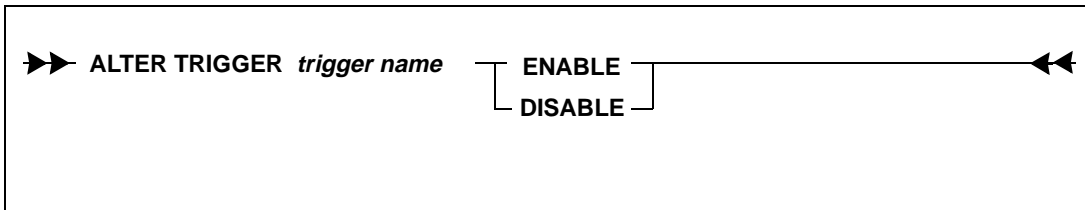
This command enables or disables triggers defined on tables. To execute the ALTER TRIGGER command, you must be the owner of the table, or have SYSADM or DBA authority.

A database trigger has a status of either enabled or disabled. If a trigger is enabled, SQLBase fires the trigger when an activating DML statement is issued. If disabled, SQLBase does not fire the trigger when an activating DML statement is issued. When you create a trigger with the CREATE TRIGGER command, SQLBase enables it automatically.

The ALTER TRIGGER does not change the definition of an existing trigger. It updates the status of the existing trigger. To redefine a trigger, you must drop the trigger with the DROP TRIGGER command, and then create it with the CREATE TRIGGER command.

You may need to disable triggers on tables for the following reasons:

- To bypass errors in which a trigger refers to an object that is unavailable.
- To load a large database without firing triggers so the load can proceed quickly.
- To reload a database.

Warning: Do NOT disable triggers on replicate tables.SQLBase replication uses a trigger-based mechanism to capture changes to replicate tables; therefore, if you disable triggers on replicate tables, the RSA cannot track changes made to them.

As an alternative to using this command, SQLBase provides a stored procedure that lets you easily disable or enable all triggers defined on a table. For more information, read *Triggers* on page *7-54*.

Clauses

trigger name
The name of the trigger to be enabled or disabled.

You can provide a fully-qualified trigger name by prefixing the creator's name. If you omit the creator's name, the current user is assumed to be the creator.

ENABLE

Enables the trigger.

DISABLE

Disables the trigger.

Example

```
ALTER TRIGGER JOB_UPDT DISABLE;
```

See also

CREATE TRIGGER
DROP TRIGGER

AUDIT MESSAGE
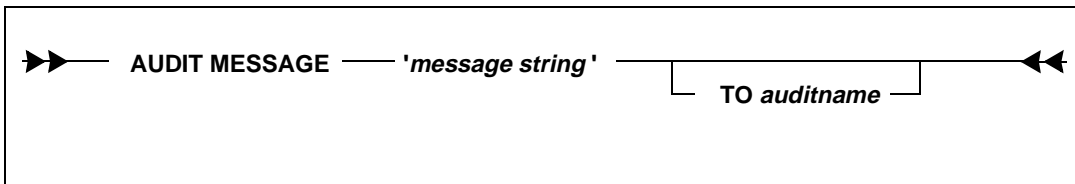
Use this command to write a message in all or specified audit files. This command requires either a cursor or server handle.

In the audit file, a message follows this record layout:

```
998,datetime,database,username,clientname,audit message
```

Clauses

message string

This is the message to be included in the audit file. It can be no longer than 254 characters, and must be enclosed in single quotes.

TO auditname

This is the name of the audit created with START AUDIT. SQLBase writes the message to the audit file associated with this audit identifier.

This clause is optional. If you do not enter an audit name, the message is written to all active audits.

Examples

The following example is issued from a SQL script that you can run from SQLTalk:

```
AUDIT MESSAGE 'Start SQL script';
```

CHECK DATABASE

```
▶▶── CHECK DATABASE ──────────────────────────────── ◀◀
                              └ SYSTEM ONLY ┘
```
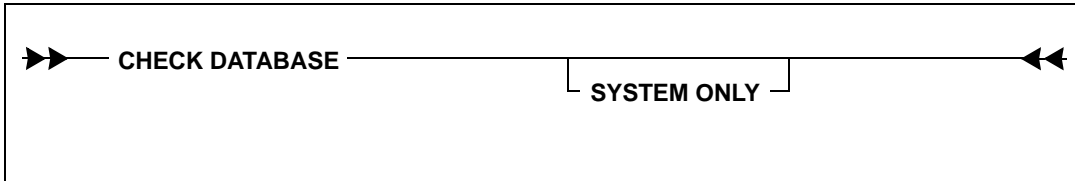
This command performs integrity checks on the entire database. Integrity checking consists of the following:

- Checking the integrity of the system and group free space data structures

- Checking the system data and allocation structures

- Verifying the table row count against the actual number of rows

- Cross-checking each index against its base table

- Checking the integrity of each row and index page

- Ensuring that each page is part of an allocated structure or is on a free page list

This command reads and places a shared lock on every page of the database. For this reason, you should run CHECK DATABASE only when there are no concurrent updates being performed.

If your database is very large, this can be a time-consuming operation. A CHECK DATABASE command is equivalent to the following command sequence:

```
CHECK DATABASE SYSTEM ONLY;
CHECK INDEX <each user index>;
CHECK TABLE <each user table> WITHOUT INDEXES;
```
or:
```
CHECK DATABASE SYSTEM ONLY;
CHECK TABLE <each user table>;
```

Clauses

SYSTEM ONLY

Verify only system-defined tables and indexes; ignore user-created tables and indexes.

Integrity checking consists of:

- Checking the integrity of the table free space data structures
- Checking the system data and allocation structures

Example

```
CHECK DATABASE;
```

See also

REORGANIZE

CHECK INDEX

▶▶ ——— **CHECK INDEX** *index name* ——————————————————◀◀

Use this command to perform an integrity check on a specific index. Integrity checking consists of:

- Checking the integrity of index pages
- Cross-checking each index against its base table

You must specify the index name as *creator.indexname*. If you omit the *creator* portion of the name, it defaults to your username.

If SQLBase finds an integrity violation, it stops the integrity check and reports an error to the user. If the problem occurred on a user-defined object, SQLBase identifies the name of the object. If the problem occurred on a system-defined object, SQLBase provides either a description of the object or the name of the object.

Read the SET ERRORLEVEL documentation in the *SQLTalk Command Reference* for an explanation of the error message detail that SQLBase displays.

Examples

```
CHECK INDEX EMP_IDX;
```

CHECK TABLE

▶▶ ——— **CHECK TABLE** *table name* ——————————————————◀◀
                                    └ **WITHOUT INDEXES** ┘

Use this command to perform an integrity check on a specific table. Indexes associated with the table are also checked.

You can specify a view name instead of a table name. In this case, SQLBase does not check the row data pages or any user-defined indexes of the underlying table or tables. Only system indexes related to the view are checked.

You must specify the table or view name as *creator.tablename*. If you omit the *creator* portion of the name, it defaults to your username.

Integrity checking consists of:

- Verifying the table row count against the actual number of rows
- Cross-checking each index against its base table
- Checking the integrity of each row and index page

If SQLBase finds an integrity violation, it stops the integrity check and reports an error to the user. If the problem occurred on a user-defined object, SQLBase identifies the name of the object. If the problem occurred on a system-defined object, SQLBase provides either a description of the object or the name of the object.

Read the SET ERRORLEVEL documentation in the *SQLTalk Command Reference* for an explanation of the error message detail that SQLBase displays.

Clauses

WITHOUT INDEXES

Prevent SQLBase from verifying any indexes associated with the specified table.

Examples

```
CHECK TABLE EMP;
CHECK TABLE EMP WITHOUT INDEXES;
```

COMMENT ON
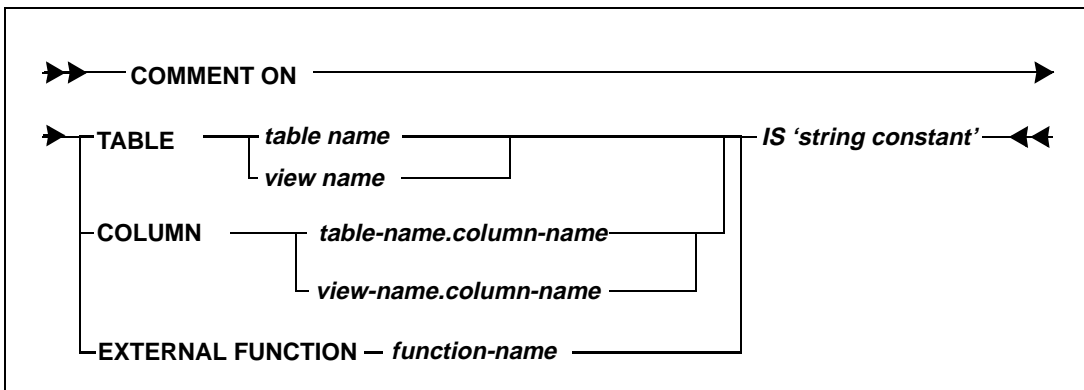
This command places a comment in the REMARKS column of the following tables: SYSTABLES, SYSCOLUMNS, or SYSEXTFUN. A comment can be added for a table, view, column, or external function.

You must have ALTER privileges on the table to use this command.

In SQLTalk, the REMARKS column is not displayed on the screen unless you enter the command COLUMN 4 WIDTH 20.

The COMMENT ON command is like the LABEL ON command. The difference is that the REMARKS columns (maintained by COMMENT ON) is 254 characters long while the LABEL column (maintained by LABEL ON) is 30 characters long.

Clauses

TABLE table name or view name

This specifies the name of a table to which to add a comment.

COLUMN table name.column name or view name.column name

This specifies the name of a column to which to add a comment.

TABLE table name or view name

This specifies the name of a table to which to add a comment.

EXTERNAL FUNCTION function name

This specifies the name of an external function to which to add a comment.

IS 'string-constant '

The comment cannot be longer than 254 characters (maximum length of a VARCHAR column). The comment must be enclosed in *single* quotes.

Examples

```
COMMENT ON TABLE EMP
    IS 'CONTAINS EMPLOYEE PERSONAL INFORMATION';

COMMENT ON COLUMN EMP.JOB
    IS 'CONTAINS JOB TITLE FOR EMPLOYEE';

COMMENT ON EXTERNAL FUNCTION MYFUNC
    IS 'CONTAINS MYFUNC INFORMATION';
```

See also

LABEL ON

COMMIT
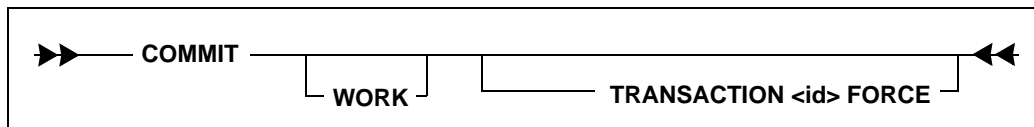
This command ends the current transaction (logical unit of work). A transaction has one or more SQL commands that must either all be executed or none at all.

This command commits all changes made to the database since either the last COMMIT or ROLLBACK, or the initial user connection, if there were no commands issued. This command commits the work for all cursors that the SQLTalk session or application has connected to the database.

Connecting to a database causes an implicit commit of a transaction. After establishing this connection to the database, SQLBase issues a COMMIT to establish the starting point of the first transaction in the logging system. However, subsequent connections to other cursors are not specifically database connections, and do not cause SQLBase to issue a COMMIT or activate any transaction control devices. Also, they do not alter the flow of the current transaction and destroy compiled commands.

The COMMIT operation applies to all SQL commands including data definition commands (CREATE, DROP, ALTER) and data control commands (GRANT, UPDATE, DELETE).

Locks are always released after a COMMIT except when cursor-context preservation is on.

Any user with CONNECT authority can execute this command.

Clauses

WORK

This is a noise word that can be coded, but it has no effect. It is provided for DB2 compatibility.

TRANSACTION <ID> FORCE

This clause forces a manual COMMIT of an in-doubt distributed transaction. Generally, the automatic recovery feature of the commit server daemon will resolve all transactions; you should only force a COMMIT as a last resort. The <ID> value is the transaction's global ID in the SYSPARTTRANS table.

Only COMMIT a transaction that has a status of COMMITTING.

Examples

```
COMMIT;  (signals end of transaction and start of new one)
<SQL Command ...>
<SQL Command ...>
<SQL Command ...>
COMMIT; (commits previous three SQL commands)
```

See also

ROLLBACK
SAVEPOINT

CREATE DATABASE

```
►► CREATE DATABASE database name ───────────────────────────         ◄◄
                                    ┌─ IN stogroup name ──┐
                                    └─ LOG TO stogroup name ─┘
```
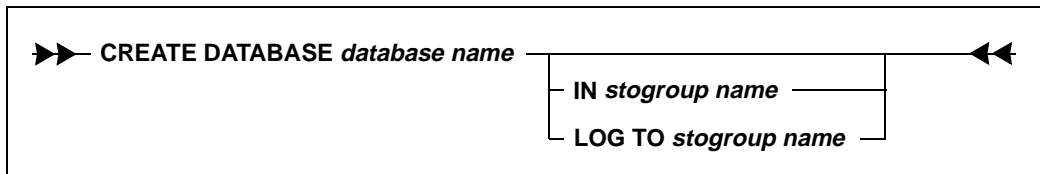
This command physically creates and installs a database. If you are not using a single engine SQLBase product, SQLBase creates the database on the server specified by the last SET SERVER command, and installs the database on the network.

About new databases

SQLBase creates a new database in the first directory on the *dbdir* path or in the current directory if *dbdir* is not specified. SQLBase also adds the dbname keyword to *sql.ini*.

In SQLBase, a database contains a database file placed in a subdirectory. The database file must have the extension *.dbs*, for example, *demo.dbs*. The name of the subdirectory must be the same as the database file name without the extension, for example, *\demo*.

Usually the database sub-directory is placed in the *centura* directory. This is the default, but you can change to any location using the *dbdir* keyword in SQL.INI.

SQLBase expects the name of the *.dbs* file to be exactly the same as the name of the subdirectory.

Note: The above rules *only* apply to non-partitioned databases.

Clauses

database name

The name of the new database to be created. The maximum length of the database name is 8 characters. Unlike other ordinary identifiers, you cannot use special characters in a database name, and the first letter must be alphabetic.

Do not specify an extension for a database name, such as *demo.xyz*. SQLBase automatically assigns a database name extension of *.dbs*. SQLBase will store a database called *demo* in a file named *demo.dbs*. These rules do not affect partitioned databases.

Do not specify the name *main* for your database, since this is used to store control information for partitioned databases. Also, do not specify the name of an existing server for your database.

If a database file with the same name already exists (and the PAUSE option is turned ON), SQLTalk prompts you with the message:

```
Database file already exists. Overwrite it (Y/N)?
```

This lets you decide if you really want to remove the existing database.

IN stogroup name

You can specify a storage group for the database and a separate storage group for the log. If you do not specify a storage group, the default storage group in the main database is used if one exists. If either the *main* database does not exist or you do not specify a default storage group, the database is created and allocated like an ordinary single-file database.

LOG TO stogroup name

You can place the log file on a disk separate from the database for better performance and integrity. If you specify a database storage group, but not one for the log, the log file space is allocated using the database storage group.

Examples

```
CREATE DATABASE SAMPLE;

CREATE DATABASE ACCTPAY IN ACCTDEPT
    LOG TO ACCTDEPT;
```

See also

CREATE DBAREA
CREATE STOGROUP
DROP DATABASE
INSTALL DATABASE
SET SERVER

CREATE DBAREA

```
  ▶▶── CREATE DBAREA dbarea name ────────────────────────────────▶

  ▶── AS [']filename['] / [']raw device ['] ──────────────────────◀◀
                                    └─ SIZE megabytes ─┘
```
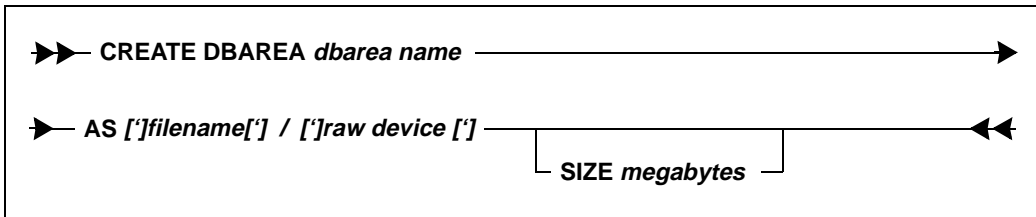
This command physically creates a database area of a specified size either on the server or in a raw partition. Commands or characters enclosed in brackets ([ ]) are optional.

The default size for a database area is 1 megabyte. The maximum size is limited by available disk space. If you are creating a database area on a raw device, you do not need to specify the size.

An error message appears if the file already exists or the disk space is already being used for another database area. An error also appears if a file of the specified size cannot be created or if the actual size of the raw device is smaller than the specified size of the area.

This command requires either an existing *Centura* directory or a valid setting for the DBDIR parameter.

Clauses

dbarea name

The name of the new database area that you create. The maximum length of the database area name is 18 characters.

AS filename/raw device

Allows you to create a database area in a specified filename or raw device. If the filing system in use allows embedded blanks, you must use single quotes around the filename or raw device.

SIZE megabytes

Allows you to specify the size of the database area in megabytes. Do not attempt to create a database area which is larger than the actual amount of free disk space available on the specified device.

Example

```
CREATE DBAREA ACCT1 AS PAYROLL SIZE 5;
```

See also

ALTER DBAREA
DROP DBAREA
DROP DATABASE
INSTALL DATABASE

CREATE EXTERNAL FUNCTION

The syntax diagram shows:

CREATE EXTERNAL FUNCTION *function name*

PARAMETERS **(** *external type* , **)**

RETURNS **(** *external type* **)**

LIBRARY *library-name*

EXTERNAL NAME *external-name*

CALLSTYLE **PASCAL/STDCALL** / **CDECL**

EXECUTE IN **SEPARATE PROCESS** / **SAME THREAD**
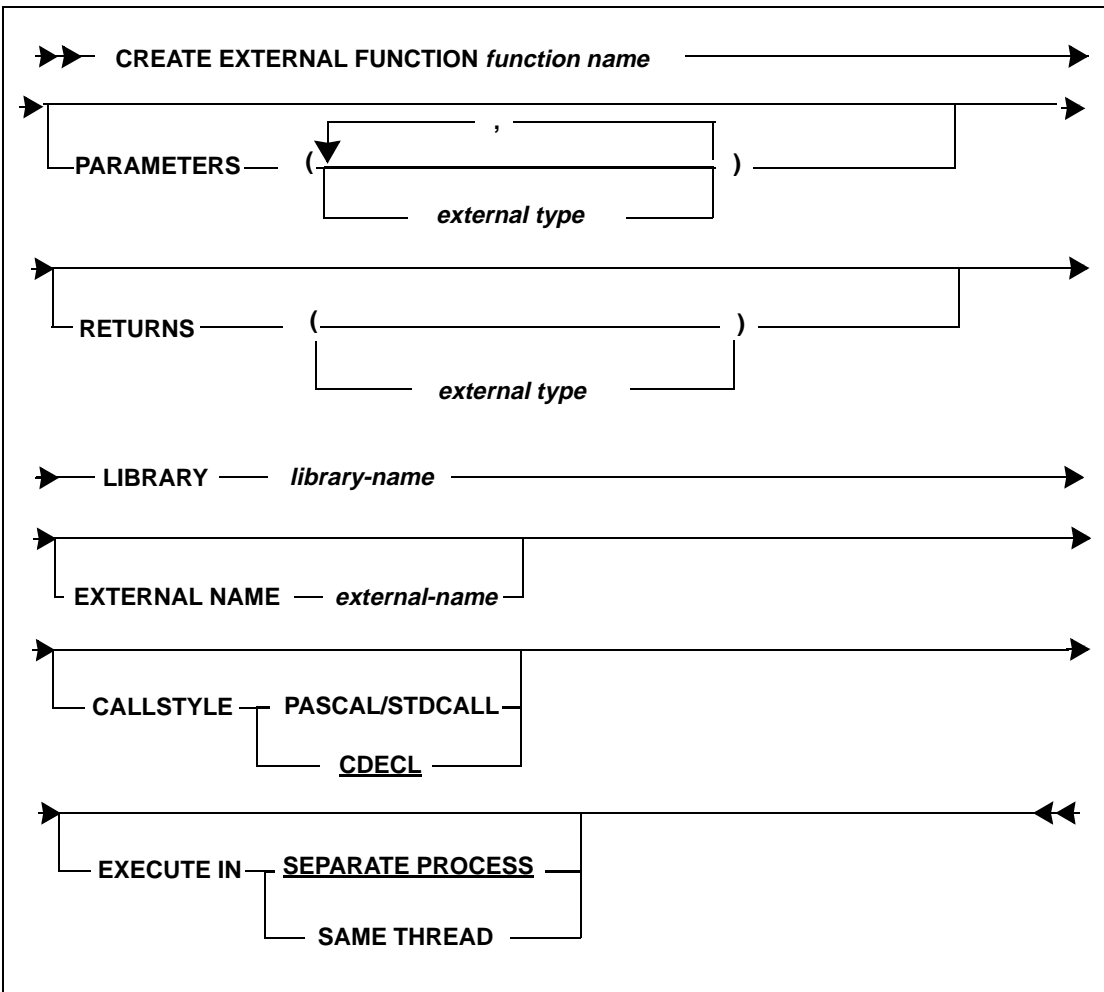
Use this command to create an external function, a user-defined function that resides in an "external" DLL (Dynamic Link Library) invoked within a SQLBase stored procedure.

DBA authority is required to create external functions.

If a user is granted **execute with creator privileges** on a procedure that calls external functions, then the user does not need execute privileges on any external function invoked within the procedure. Only the CREATOR of the procedure needs to have execute privileges on the external function.

If the user is granted **execute with grantee privileges** on a stored procedure, the user must also have execute privileges on the external functions invoked within the procedure. For details on setting up security for external functions, see the *Database Administrator's Guide*.

Read Chapter 9, *External Functions* for more information on creating external functions.

Clauses

function name

Specifies the name of the function. This is the name that refers to the function within SQLBase. Function names are similar to other database object names, except they can be up 64 characters in length.

Unless specified within double quotes, a function name must start with an alpha (a - z) character. By default, the characters are uppercased.

You must specify a function name in double quotes if the name contains special characters or starts with a non-alpha character.

Note that if you enclose the name in double quotes, the case of the name is preserved.

Please note the following restrictions:

- Function names cannot be the same as procedure names and vice versa.

- Functions names cannot be the same name used in any of the SQLBase aggregate functions (for example, min, max, avg, etc., or any functions beginning with the @ symbol, such as @ASIN, @ATAN, @CHAR, etc.)

- Function names cannot begin with **SQL**.

- If the external name is not used in the function definition, then the function name must match the exported name in the DLL.

- If the external name is used in the function definition, then the external name must match the exported name in the DLL.

PARAMETERS

Specify this clause if you want to define input parameters to the external function. If there are no parameters for the external function, omit the PARAMETERS clause, or provide empty parentheses **( )** in the declaration.

The data type for parameters tells SQLBase the format (both size and pass by reference value) to use when passing data to the external function.

The external type typically corresponds to a standard Microsoft data type. For details, read Chapter 9, *External Functions*.

To specify an external data type with more than one input parameter, separate each entry by a comma. For example:

```
. . .

PARAMETERS (int, lpint, boolean)

...;
```

RETURNS

Specify this clause if you want to define return values to the external function. If there is no return type from the external function, omit the RETURNS clause, or provide empty parentheses **( )** in the declaration.

The external data type tells SQLBase the format (both size and pass by value) to use when returning a value to an external function. The external type typically corresponds to a standard Microsoft data type. For details, read Chapter 9, *External Functions*.

LIBRARY library-name

Specify the dynamic linked library (DLL) name where the function resides. You must provide a fully qualified path name for the file, or else be sure the PATH environment variable is set to point to the location of the file in your operating system.

Specify the library name as a string with up to 254 characters. You can include special characters in the string. If the library name contains spaces, you must delimit the name in single quotes (for example, 'lib name').

EXTERNAL external-name

Specify this clause if you want to provide an external name for the function. An external name lets you create a function name that references the function in a DLL by another name. Thus, the function has a calling name that is separate from the name used to reference the same function in the DLL.

Specify the eternal name as a string with up to 254 characters. You can include special characters in the string. The external name is case-sensitive and must be identical to the exported function name in the DLL.

Note that if you do not supply an external name, the function name is the same name that is used in the DLL.

CALLSTYLE

Specify this clause if you want to change the compiler style that is required to invoke the external function. For details, read Chapter 9, *External Functions*.

Note: Be sure to specify the correct callstyle for your platform. An incorrect callstyle can result in server failure.

PASCAL/STDCALL

PASCAL applies only to 16-bit platforms and is the callstyle for Windows API calls. STDCALL applies only to 32-bit platforms and is the callstyle for all 32-bit Windows API calls.

CDECL

This is the default compiler callstyle and applies to both 16-bit and 32-bit platforms.

EXECUTE IN

Specify this clause only if you are using a 32-bit platform and want to change the execution mode to SAME THREAD or SEPARATE PROCESS.

For details on execution mode, read Chapter 9, *External Functions*.

Examples

```
CREATE EXTERNAL FUNCTION MYFUNC
  PARAMETERS (int, lpint)
  RETURNS ()
  LIBRARY myfunc.dll
  EXECUTE IN SAME THREAD;
```

See also

ALTER EXTERNAL FUNCTION
DROP EXTERNAL FUNCTION
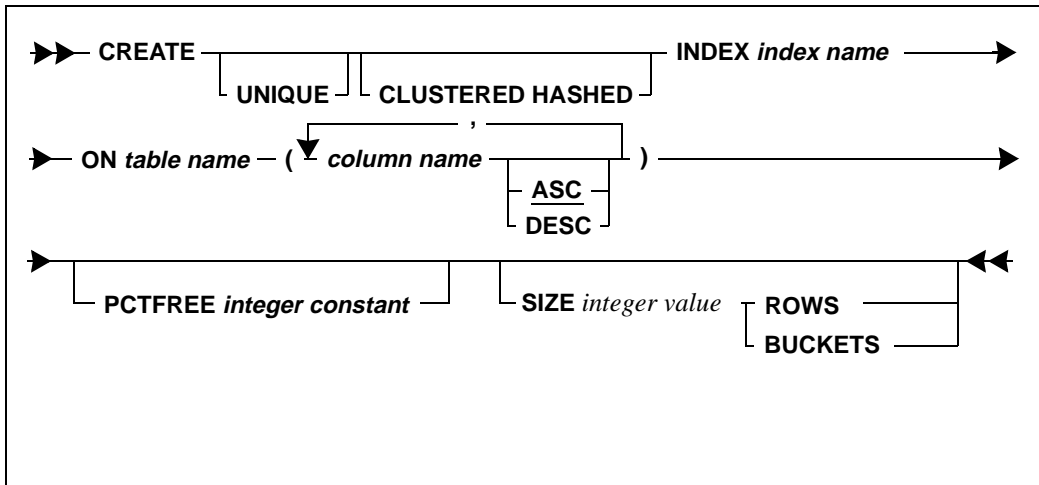
CREATE INDEX

This command creates an index on one or more columns of a table. Indexes optimize data retrieval since the data can be found without scanning an entire table. Indexes can also force unique data values in a column.

If an index is created on an empty table, the statistics reflect that the index is empty and SQLBase does not use the index in the queries. Therefore, be sure to always run UPDATE STATISTICS after the table is populated so the statistics accurately reflect the data. (You can create indexes at any time. When an index is created, statistics are gathered regarding the index and its associated values.)

There is no limit on the number of indexes per table.

You cannot update the key of a clustered hash index.

Like all DDL commands, this command locks system tables while executing.

If you create a table with a primary key with CREATE TABLE, you must create a unique index on the primary key's columns.

Index size

The maximum number of columns in an index cannot exceed 16. If this limit is reached, SQLBase issues an error message.

The maximum size of an index key is:

| 6 | + | number of columns in index | + | sum of lengths of all columns in index | <= | 255 |

Note that SQLBase issues an error message if an index key size has a length greater than 255.

The length of each column depends on its data type. For example, a CHAR(10) column is 10 bytes; any numeric column is 12 bytes; and any date/time column is 12 bytes.

Consider the following columns:

```
LASTNAME    CHAR20)
FIRSTNAME   CHAR(20)
MI          CHAR(1)
```

Create the concatenated index:

```
LASTNAME    CHAR(20)
FIRSTNAME   CHAR(20)
MI          CHAR(1)
            41
```

The following calculation shows the size of the index key:

| 6 | + | 3 | + | 41 | = | 50 |

Since this length is less than 255, it is valid.

As another example, consider the index on the following single column:

```
LARGECHAR(249)
```

The index is 249. Adding the number of columns and the sum of their lengths results in the following sum:

| 6 | + | 1 | + | 249 | = | 256 |

This length is not allowed, since 256 > 255.

You must have the INDEX privilege on the table to execute this command.

Functions in Indexes

An index can be created for one or more column values resulting from applying a function to the column. Functions for an index cannot be nested. Not all functions can be used to create an index.

Indexes created in this manner are used when the respective function is used in the WHERE clause. For functions which have arguments in addition to the table column (such as @SUBSTRING), all arguments must agree exactly between the CREATE INDEX and WHERE clause invocations in order for the index to be used.

A case-insensitive index results from applying the @UPPER or @LOWER function to the column in the CREATE INDEX command. When you query a column containing names that were entered using mixed case, and use the respective function in the WHERE clause to constrain the query, the rows returned include those in upper and lower case.

The following functions are allowed in CREATE INDEX.

@CHAR	@QUARTER
@CODE	@QUARTERBEG
@DATEVALUE	@RIGHT
@DAY	@SECOND
@HOUR	@STRING
@LEFT	@SUBSTRING
@LENGTH	@TIMEVALUE
@LICS	@TRIM
@LOWER	@UPPER
@MICROSECOND	@VALUE
@MID	@WEEKBEG
@MINUTE	@WEEKDAY
@MONTH	@YEAR
@MONTHBEG	@YEARBEG
@PROPER	@YEARNUM

Clauses

index name

Each index name is a long identifier prefixed by an implicit qualifier which is the authorization-id of the index creator. The index name (including the qualifier) must be unique within a database.

table name

View names *cannot* be used in the creation of an index.

UNIQUE

This keyword enforces unique key values within the table. It specifies that no combination of indexed columns in the table can be identical. If this uniqueness property is violated during index creation, or during an insert or update, an error is returned.

CLUSTERED HASHED

This clause stores the data rows in locations based on the key hash value (clustering). A clustered hashed index speeds random access to rows in a table. If a table has a unique key that identifies each row, declaring a clustered hashed index on that key usually allows rows to be accessed with 1 disk read.

SQLBase uses a clustered hash index when both of the following situations are true:

- All of the key columns are in the WHERE clause
- The columns only use the equals (=) condition, such as C1=10.

If this clause is not specified, a *B-tree (non-clustered)* index is created.

The table can grow or shrink, but clustered hashed indexes are intended for tables which are static or where an upper bound for the size of the table can be specified. A clustered hashed index can be specified for a non-unique key, but access only improves if there are relatively few rows for each key value.

Only one clustered hashed key can be created for a table, and it cannot be updated.

A CREATE INDEX command that specifies a clustered hashed index must be given after the CREATE TABLE command and *before* any data is added to the table.

You cannot drop and then recreate a clustered hashed index on an empty table if rows existed previously but were then deleted. You must first drop and then recreate the table before you recreate the index.

ASC
DESC

This specifies whether the index is in ascending or descending order. ASC is the default order. This clause is only relevant for B-tree indexes.

PCTFREE integer constant

The PCTFREE (percent free) clause specifies how much free space to allocate in each index entry when the index is initially built. After the index is built, key insertions and deletions can make the actual free space vary between 0% and 50%. If not specified, the default free space is 10%.

The PCTFREE keyword is followed by a number (between 0 and 99 inclusive) that specifies the percentage of free space to be left in each index entry when the index is first built.

Normal values are 0-50%. Specifying 90-99% makes a binary index tree (2 entries per page) which results in the maximum height B-tree. This degrades retrieval performance.

This clause is ignored for a CLUSTERED HASHED index.

SIZE integer constant

Specify this clause in conjunction with either ROWS or BUCKETS. This controls the "expected" size of the index and is specified as a number of rows or buckets. If the size is too small, overflow pages are used and performance degrades. If the size is too large, overflow pages are not used, but disk space is wasted. This clause is only relevant for clustered hashed indexes.

You must specify this clause if you specify the CLUSTERED HASHED clause.

ROWS

Use this clause in conjunction with the SIZE *integer value* clause to specify the number of rows to store a clustered hashed index. If you use SIZE..ROWS instead of SIZE...BUCKETS, SQLBase calculates the actual number of primary buckets and round up to the nearest prime number for the hash based on the number of rows specified, the size of the row from the sum of all declared column widths, and the SQLBase page size.

BUCKETS

Use this clause in conjunction with the SIZE *integer value* clause to directly specify the number of primary bucket pages to store clustered hashed index and data. SQLBase allocates primary buckets in SQLBase pages to store the clustered hashed index and its data. The primary buckets are the direct entries into the hash table, and require only one I/O for access. SQLBase will round up the specified BUCKETS value to the nearest prime number to obtain better hash distribution.

The number of pages for the primary buckets is stored in the SYSADM.SYSINDEXES.PRIMPAGECOUNT column. However, when you specify the clustered hashed index in buckets, the SYSADM.SYSINDEXES.IXSIZE column containing the number of ROWS can be null.

Examples

Create an index named HIRE_IDX using the HIREDATE column.

```
CREATE INDEX HIRE_IDX ON EMP (HIREDATE);
```

Create a concatenated index composed of LNAME and FNAME.

```
CREATE INDEX NAME_IDX ON EMP (LNAME, FNAME);
```

Create a descending index on the EMP_IDX column of the EMP table. Disallow duplicate part numbers.

```
CREATE UNIQUE INDEX EMP_IDX ON EMP (EMPNO DESC);
```

This example illustrates the creation and use of a case insensitive index.

```
CREATE INDEX LN_IDX ON EMP (@UPPER(LNAME));
```

In the above example, an upper case index is created for LNAME. This index is used when the @UPPER function is specified in the WHERE clause of a SELECT, thereby using case insensitive sort order and using the index. The following example illustrates this.

```
SELECT LNAME FROM EMP WHERE
@UPPER(LNAME) = 'JONES' ORDER BY 1;

NAME
====
JONES
Jones
jones

3 Rows Selected
```

Create an index on the first 3 characters of a column.

```
CREATE INDEX CODE_IDX ON EMP
     (@LEFT(DEPTNO, 3));
```

The select command that uses this index must agree with the definition of CODE_IDX.

Get all the rows for people in the '250' division.

```
SELECT * FROM EMP WHERE
     @LEFT(DEPTNO,3) = '250';
```

See also

CREATE TABLE

CREATE STOGROUP

```
>>-- CREATE STOGROUP stogroup name --------------------------->

                                ,
>-- USING --------- dbarea-name --------------------------<<
```
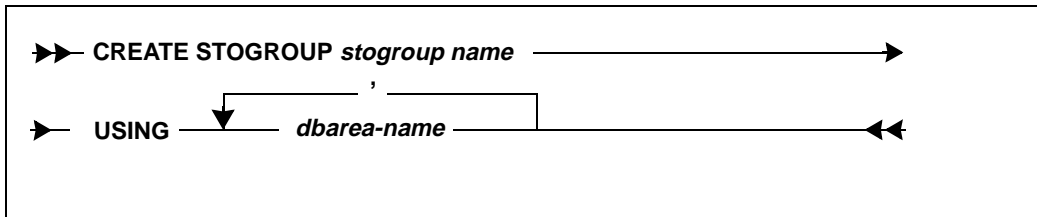
This command creates a storage group. If the volumes containing the database areas are not mounted, an error occurs when you try to create a database.

Clauses

stogroup name

This names the storage group that you create. The maximum length of the storage group name is 18 characters.

USING dbarea name

This is a list of database areas. Database areas must already exist.

Example

```
CREATE STOGROUP ACCTDEPT USING ACCT1, ACCT2;
```

See also

ALTER STOGROUP
DROP STOGROUP

CREATE SYNONYM

```
>>── CREATE ──┬──────────┬── SYNONYM synonym-name ──────────────>
              └─ PUBLIC ─┘

>── FOR ──┬─── TABLE ───────────┬── object name ──────<<
          ├── EXTERNAL FUNCTION ─┤
          ├──── COMMAND ────────┤
          └──── PROCEDURE ──────┘
```
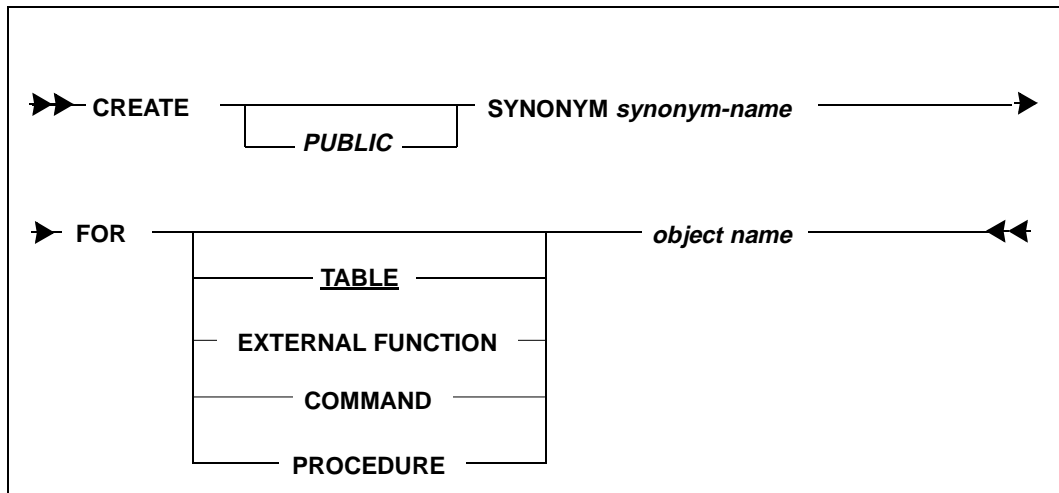
This command defines an alternate name for a table, view, external function, stored command, or stored procedure. Alternate names let you reference another user's tables, views, external functions, stored commands, or stored procedures without having to use the qualified name (*auth-id.table-name or auth-id.external function-name*).

You can create synonyms for a table, view, external function, stored command, or stored procedure, if you own the given object. If you own an external function, you can also grant/revoke execute privileges on that function. If execute authority is granted on a synonym for a function, the base name is inserted into the SYSOBJAUTH table.

Synonyms for tables are stored in the SYSADM.SYSSYNONYMS system catalog table. Synonyms for external functions, stored commands, and stored procedures are stored in the SYSADM.SYSOBJSYN system catalog table.

Synonyms used in a command can only be executed by the creator of the synonym.

If you create a local synonym with the same name as a PUBLIC synonym, the local definition overrides the public definition.

When an external function, stored command, or stored procedure is invoked, SQLBase looks for the function in this order of precedence:

- functions owned by the creator of the invoking object

- private synonyms

- public synonyms

Clauses

PUBLIC

This allows you to access the table, external function, stored command, or stored procedure through the synonym without fully qualifying the object name with the authorization-id of the owner.

You must own the table, external function, stored command, or stored procedure or be a DBA or SYSADM to create a PUBLIC synonym.

You must have the appropriate privileges on the underlying table, external function, stored command, or stored procedure to access it through a PUBLIC synonym.

synonym name

The synonym is named in the same manner as a table, view, external function, stored command, or stored procedure. It must not be the same as any other synonym, table, view, external function, stored command, or stored procedure that you own. The same rules for naming tables, views, external functions, stored command, or stored procedure also apply to synonyms.

You can create synonyms for tables called "TABLE", "EXTERNAL", "COMMAND", or "PROCEDURE."

Note: The called name of an external function can be the synonym name for the function rather than the actual function name. The SYSDEPENDENCIES catalog maintains dependencies between a stored procedure and the called name of an external function.

object name

The name of an object type can be a table, view, external function, stored command, or stored procedure. The table-name can name an existing view or table in the database. The view-name must name an existing view or table in the database. The external function name, stored command name, or stored procedure name must name, respectively, an existing external function, stored command, or stored procedure in the database.

TABLE

If the object type is omitted, the default is TABLE.

EXTERNAL FUNCTION

You must specify EXTERNAL FUNCTION after the FOR keyword when creating a synonym for an external function.

COMMAND

You must specify COMMAND after the FOR keyword when creating a synonym for a stored command.

PROCEDURE

You must specify PROCEDURE after the FOR keyword when creating a synonym for a stored procedure.

Examples

```
CREATE PUBLIC SYNONYM SN2 FOR EXTERNAL FUNCTION MYFUNC;

CREATE SYNONYM SN2 FOR EXTERNAL FUNCTION MYFUNC;
```
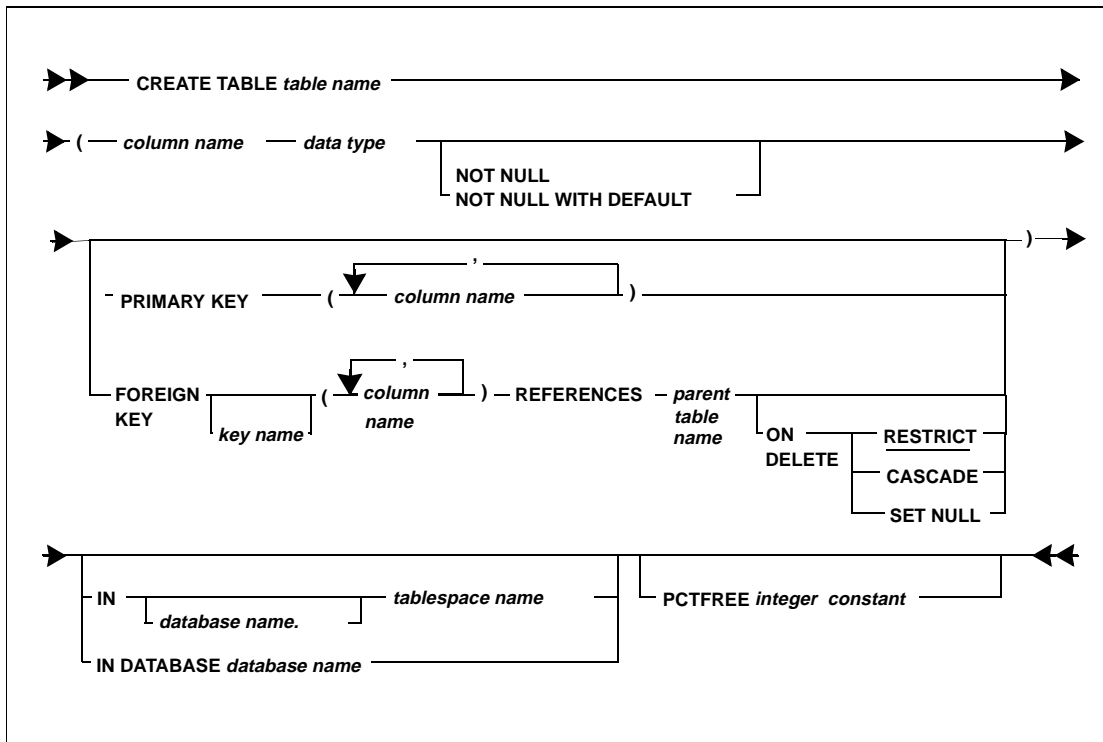
Note that in the following examples, since no object type is included, the object type defaults to TABLE.

```
CREATE SYNONYM ES FOR USER1.EMPSAL;

CREATE SYNONYM ES FOR EMPSAL;

CREATE PUBLIC SYNONYM ES FOR SYSADM.EMPSAL;
```

See also

CREATE TABLE
CREATE EXTERNAL FUNCTION
GRANT EXECUTE ON
REVOKE EXECUTE ON

CREATE TABLE

```
►►──── CREATE TABLE table name ──────────────────────────────►

►─( ── column name ── data type ──────────────────────────────►
                         NOT NULL
                         NOT NULL WITH DEFAULT

►──────────────────────────────────────────────────────────)─►
                              ,
    PRIMARY KEY ──── ( ─── column name ─── ) ───

                              ,
    FOREIGN ───── ( ─ column ─ ) ─ REFERENCES ─ parent ─
    KEY    key name   name                      table
                                                name   ON   RESTRICT
                                                       DELETE
                                                              CASCADE

                                                              SET NULL

►─────────────────────────────────────────────◄◄
    IN ──────────────── tablespace name ──  PCTFREE integer constant
        database name.
    IN DATABASE database name
```

This command creates a table with the specified columns. You can define a maximum of 253 columns for each table. You must have RESOURCE, SYSADM, or DBA authority to execute this command.

When you use CREATE TABLE with referential constraints, you should define a foreign key with the same specifications as the primary key of the parent table. A referential constraint defines the rules for a relationship between the primary key of a parent table and a foreign key of a dependent table. A referential constraint requires that for each row in a dependent table, the value of the foreign key must appear as the primary key of a row in the parent table.

You must designate the parent table name when you define the foreign key. This parent table must have a primary key and a primary index. You can also specify the delete rule of the referential constraint. The default rule is RESTRICT.

Like all DDL commands, this command locks system tables while executing.

Clauses

table name

A fully-qualified SQL table name has the form:

> authorization-id.*table-name*

The authorization-id is a qualifier denoting the creator of the table. The combined authorization-id.table name must form a unique name which does not identify any existing table, view, or synonym in the database.

When you create a table, if you do not specify the authorization-id, your default authorization-id is automatically prefixed to the table name.

column name

A column name must begin with a letter (A through Z and the special characters #, @ and $) and must not exceed 18 characters.

A fully-qualified column name has the form:

> *table name . column name*

You can use the unqualified column name when you define the table, and it must be a long identifier (18 characters maximum). Each column name must be unique within a table.

data type

A column can be one of the following data types. These data types are described in the section *Data types* on page *2-7*.

> CHAR (length)
> VARCHAR (length)
> DECIMAL [(precision, scale)]
> FLOAT
> INTEGER
> LONG VARCHAR
> NUMBER
> SMALLINT
> DATE
> DATETIME
> TIME
> TIMESTAMP

Columns defined as CHAR or VARCHAR require a length attribute.

Columns defined as DECIMAL have a default size attribute of 5,0; any other precision and scale must be declared in parentheses.

SQLBase does not allocate the full space for a row when it is inserted with null columns. An application that inserts a row with uninitialized columns and later writes values to those columns will expand the row with extent pages. To avoid the extent pages, the application should write blank-filled columns on the first INSERT of each row.

PRIMARY KEY

This creates the primary key for the table. The following rules apply to primary keys:

- If a table has a primary key, you must also create a unique index on the primary key columns to make the table complete. See the CREATE INDEX command for more information.

- The primary key format must obey the following rules:

  - Cannot contain more than 16 columns.

  - Sum of the column length attributes cannot be greater than 255 bytes.

  - Cannot contain LONG or LONG VARCHAR columns.

- You cannot use an UPDATE WHERE CURRENT clause with a primary key column.

- In a self-referencing row, you cannot update the primary key value. If a row is a *self-referencing row,* its foreign key value is the same as its primary key value.

- The values of the primary key must be unique; no two rows of a table can have the same key values.

- A table can have only one primary key.

- The primary key can be made up of one or more columns in a table. This is called a composite primary key. Separate the columns with a comma.

- Each column in the primary key must be classified with the NOT NULL constraint. However, you should not use the NOT NULL WITH DEFAULT option unless the primary key column(s) has a data type of TIMESTAMP or DATETIME.

- An updateable view defined on a table with a primary key must include all columns of the primary key. Although this is only required if you use the view in an INSERT statement, the resulting unique identification of rows is also useful if the view is used for updating, deleting, or selecting.

  If you try to insert a row into a view that does not contain values for all of the primary key columns, the following message appears:

  ```
  NOT ENOUGH NON-NULL VALUES
  ```

  This message appears because all the primary key columns are defined as NOT NULL (since a primary key cannot contain NULL values).

If you decide later to change the order of the primary key columns, you must use the following steps:

1. Run ALTER TABLE (referential integrity) and drop the primary key.

2. Drop the primary index with DROP INDEX.

3. Recreate a unique index on the new primary key columns with CREATE INDEX.

4. Run ALTER TABLE (referential integrity) again to re-add the primary key with the new column order.

FOREIGN KEY

This specifies the foreign key for a table. Every value in a foreign key must match some value in the primary key from which the foreign key column originates.

The parent table must have a unique index on the primary key.

The following rules apply to foreign keys:

- *Matching columns.* A foreign key must contain the same number of columns as the primary key. The data types of the foreign key columns must match those of the primary key on a one-to-one basis, and the matching columns must be in the same order.

  However, the foreign key can have different column names and default values. It can also have NULL attributes. If an index is defined on the foreign key columns, the index columns can be in ascending or descending order, which may be different from the order of the primary key index.

- *Using primary key columns.* A column can belong to both a primary and foreign key.

- *Foreign keys per table.* A table can have any number of foreign keys.

- *Number of foreign keys.* A column can belong to more than one foreign key.

- *Number of columns.* A foreign key cannot contain more than 16 columns.

- *Parent table.* A foreign key can only reference a primary key in its parent table. This parent table must reside in the same database as the foreign key.

- *NULL values.* A foreign key column value can be NULL. A foreign key value is NULL if any column in the foreign key is NULL.

- *Privileges.* You must grant ALTER authority on a table to all users who need to define that table as the parent of a foreign key.

- *System catalog table.* The foreign key cannot reference a system catalog table.

- *Views.* A foreign key cannot reference a view.

- **Self-referencing row.** In a self-referencing row, the foreign key value can only be updated if it references a valid primary key value. If a row is a *self-referencing row,* its foreign key value is the same as its primary key value.

key name

You can assign a name to the foreign key to identify it. This name is called a constraint name. If you do not specify a name yourself, SQLBASE generates a constraint name from the name of the first foreign key column.

A foreign key constraint name can have up to 18 characters. This means that if the first foreign key column name is more than 18 characters, you must assign a constraint name yourself that does violate this limit. Otherwise, SQLBase will not create the foreign key.

If there are multiple foreign keys referencing the same table, each foreign key must have a unique name. This ensures that every referential constraint is uniquely identified by a table name/constraint name combination.

REFERENCES

This identifies the parent table in a relationship and defines the necessary constraints. The REFERENCES clause must accompany the FOREIGN KEY clause.

NOT NULL

If you declare a column NOT NULL, it requires data to be present in the column every time a row is added to the table. If omitted, the column can contain null values, and its default value is the null value.

NOT NULL WITH DEFAULT

The NOT NULL WITH DEFAULT clause prevents a column from containing null values and allows a default value other than the null value.

The default value used depends on the data type of the column, as follows:

Data Type	Default Value
Numeric	0 (zero)
Date/Time	Current date/time
Character	One blank

The NOT NULL WITH DEFAULT clause causes the INSERT to insert the above defaults. If the column is not specified in the INSERT command, SQLBase puts a 'D' in the NULLS columns of the SYSCOLUMNS table and treats it like a NOT NULL field.

The NOT NULL WITH DEFAULT clause is compatible with DB2.

IN DATABASE database name

IN [database name] tablespace name

SQLBase accepts these clauses but ignores them. The IN clauses are compatible with DB2.

ON DELETE

This specifies the DELETE rules for the table.

The DELETE rules are optional.

The default is RESTRICT.

DELETE rules are only used to define a foreign key.

CASCADE

This deletes the selected rows first, and then deletes the dependent rows, honoring the delete rules of their dependents.

RESTRICT

This specifies that a row can be deleted if no other row depends on it. If a dependent row exists in the relationship, the delete will fail.

SET NULL

This specifies that for any delete performed on the primary key, matching values in the foreign key are set to null.

PCTFREE integer constant

This sets the free space left in each table row when it is first filled. The default free space is 10 percent.

If you plan to expand rows later by adding more columns or increasing the width of existing columns, this feature leaves space for expansion so that extension pages are not needed.

Also, for small, heavily-accessed tables where page locking can cause contention, the PCTFREE option can force fewer rows to be assigned to each page which reduces contention.

The PCTFREE value must be between 0 and 99.

Examples

Create a table EMP for storing employee data and EMPSAL for keeping a salary history.

```
CREATE TABLE EMP
    (EMPNO INTEGER NOT NULL,
    LNAME VARCHAR(15),
    FNAME CHAR(10),
```

```
                DEPTNO SMALLINT,
                HIREDATE DATE,
                JOB VARCHAR (15));

CREATE TABLE EMPSAL
                (EMPNO INTEGER NOT NULL,
                SALARY DECIMAL(5,9,2),
                REVIEW LONGVARCHAR;
```
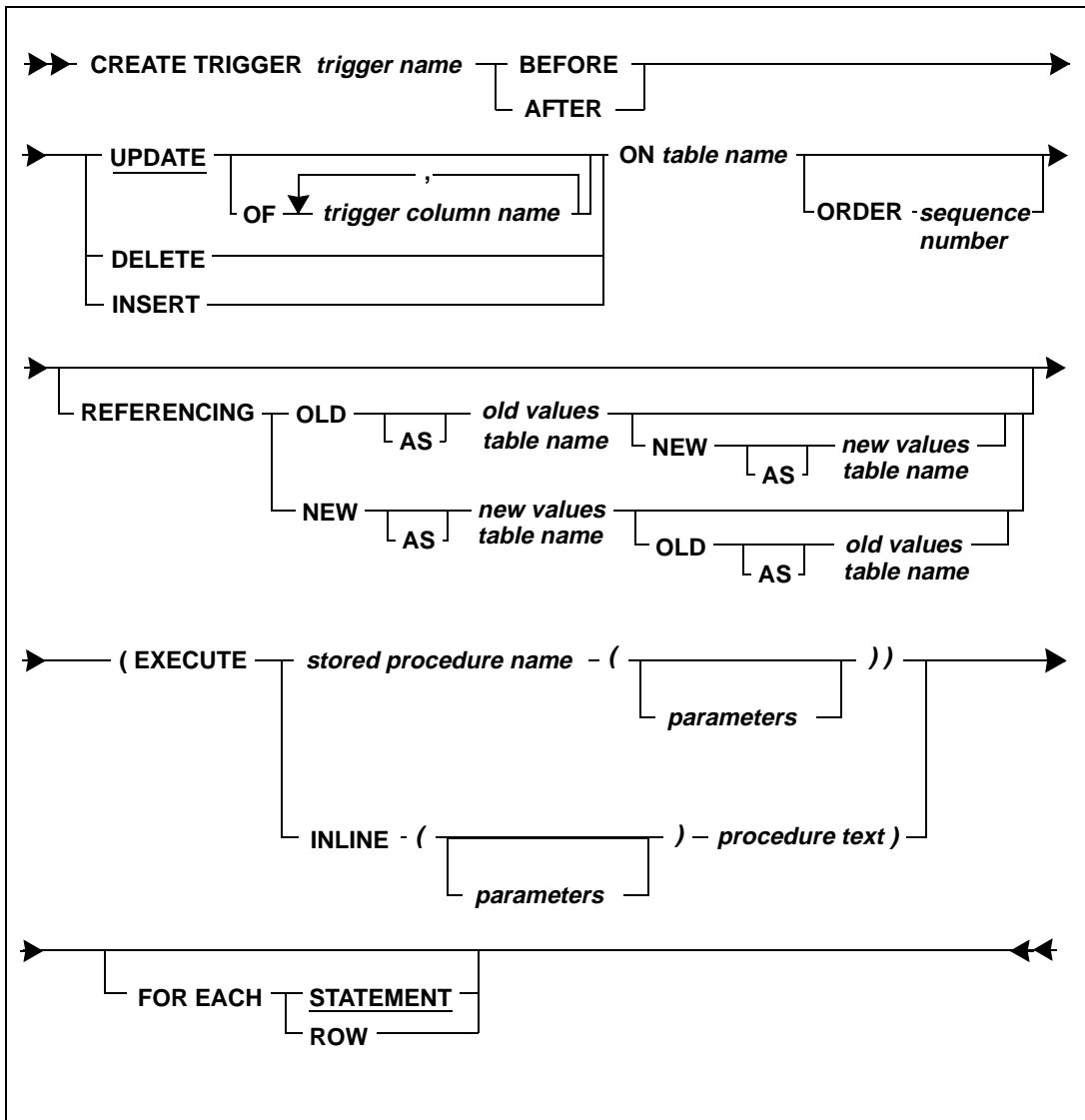
Create a table that allows the foreign key EMPNO in the EMPSAL table to reference EMPNO in the EMP table, with a DELETE CASCADE rule.

```
CREATE TABLE EMP
                (EMPNO INT NOT NULL,
                LNAME VARCHAR(15),
                FNAME CHAR(10),
                DEPTNO SMALLINT,
                HIREDATE DATE,
                JOB VARCHAR (15)
                PRIMARY KEY (EMPNO));

CREATE UNIQUE INDEX EMP_IDX ON EMP (EMPNO);

CREATE TABLE EMPSAL (EMPNO INTEGER, SALARY DECIMAL (9,2),
                REVIEW LONG VARCHAR,
                FOREIGN KEY (EMPNO) REFERENCES EMP
                ON DELETE CASCADE);
```

See also

ALTER TABLE
CREATE INDEX
DELETE
UPDATE

CREATE TRIGGER

```
►► CREATE TRIGGER  trigger name  ── BEFORE ──────────────────────────►
                                     └─ AFTER ─┘

►── UPDATE ──────────────────────── ON table name ──────────────────►
        └─ OF ──►── trigger column name ──┘      └─ ORDER ─ sequence ─┘
                       ,                                      number

►──── REFERENCING ── OLD ───────────── old values ───────────────────►
                      └─ AS ─┘        table name ── NEW ── new values ──
                                                      └─ AS ─┘ table name
                    └─ NEW ──────────── new values ──
                        └─ AS ─┘       table name ── OLD ── old values ──
                                                       └─ AS ─┘ table name

►──── ( EXECUTE ─── stored procedure name ─ ( ──────── ) ) ──────────►
                                              └ parameters ┘
                 └─ INLINE ─ ( ──────── ) ─ procedure text ) ─┘
                                └ parameters ┘

►──────────────────────────────────────────────────◄◄
   └─ FOR EACH ─ STATEMENT ─┘
                 └─ ROW ────┘
```

This command creates a trigger on a table. You cannot define a trigger on a view even if the view is based on a single table. You must be the owner of the table, or a user with SYSADM or DBA authority, to create a trigger on the table. For a general description of triggers and their use, read *Chapter 7, Procedures and Triggers*.

You can define up to sixteen triggers for each combination of table, event (INSERT, UPDATE, DELETE), time (BEFORE and AFTER), and frequency (FOR EACH ROW and FOR EACH STATEMENT).For details, read the section "BEFORE and AFTER" under *Clauses* on page *3-57*.

Because triggers can be activated by *any* user's attempt to INSERT, UPDATE, or DELETE data, no privileges are required to execute them. When a trigger is activated, the action statements are executed on behalf of the table owner, not the user who activates the trigger. However, to create a trigger which uses a stored procedure, one of the following conditions must be true:

- You have SYSADM or DBA authority.
- You own the table and the stored procedure,

    or

- You own the table and have been granted execute authority for the stored procedure.

If you have either SYSADM or DBA authority and create a table for another user, SQLBase assumes that unqualified names specified in your TRIGGER statement belong to the user. For example, assume you execute the following command as SYSADM:

```
CREATE TRIGGER A.TRIG BEFORE UPDATE on EMP...
```

Since table EMP is unqualified, SQLBase assumes that the qualified table name is A.EMP, not SYSADM.EMP.

Triggers do not need a commit from the invoking transaction in order to fire; DML statements by themselves cause triggers to fire.

If a procedure returns a non-zero value to a trigger, the trigger causes its invoking DML command to fail with the error message associated with the return code in *error.sql*. For example, if you have an INSERT trigger that calls a procedure, and the procedure returns 906 to the trigger, the INSERT command invoking the trigger will fail with error code 906 (Invalid table name). However, any commands in the transaction prior to the (failed) invoking DML statement are not rolled back.

Note: Triggered SQL statements are a part of the invoking transaction. If the invoking DML statement fails due to either the trigger or another error generated outside the trigger, all SQL statements within the trigger are rolled back along with the failed invoking DML command.

It is the responsibility of the invoking transaction to commit or rollback any DML statements executed within the trigger's procedure. However, this becomes irrelevant if the DML command invoking the trigger fails as a result of the associated trigger. In

this situation, any DML statements executed within that trigger's procedure are automatically rolled back.

In certain situations, SQLBase allows you to drop a table that has a dependent trigger or stored procedure defined on it. SQLBase does not issue a warning, but instead, during execution, issues a runtime error that the table does not exist. For example, assume you create two tables, A and B. You then create an update trigger (TRIG_A) on table A that calls a stored procedure (SP_A) to insert data into table B.

If you attempt to drop table B, SQLBase accepts your DROP TABLE command without warning you that TRIG-A is a dependent object of table B. If you go on to update table A, the trigger issues a runtime error that table B does not exist. SQLBase rolls back the command and does not permit you to update table A until you recreate table B or drop the trigger. Note that similar behavior in this example occurs if you drop stored procedure SP_A.

SQLBase also issues an error when you attempt to load triggers or static procedures that reference dropped or altered objects. To prevent the error:

- Recreate any referenced object that you drop, or
- Restore any referenced object you changed back to its original state (known by the procedure or trigger).

Following are restrictions to note when creating triggers:

- You cannot alter a table that has a trigger defined on it.
- You cannot create a trigger on a system catalog table.
- If a a DML statement updates a row that causes a trigger to be fired, you cannot update the same row again within that trigger.

    In the following example a trigger updates a row that was just inserted by a DML statement and causes SQLBase to generate an error:

    ```
    CREATE TABLE EMP (c1 int);

    CREATE TRIGGER TRIG1 after insert on EMP
    (EXECUTE inline (EMP.rowid)
    PROCEDURE P1
      string: RowIdentifier

    Action
      call SqlImmediate('update EMP set \
        (c1 = c1 + 5 where EMP.rowid =: RowIdentifier'))

    for each row;
    ```

    If you use a trigger to perform such actions like the one in the previous example, SQLBase returns error 848 ("Row being processed for a DELETE/

UPDATE was modified by triggered actions") when the invoking DML is executed.

For restrictions on setting default/derived column values when using receive parameters in triggered stored procedures, read *Using receive parameters on page 3-55*.

Triggers and Procedures

Triggers can call stored procedures and cause SQLBase to execute other triggers. You can nest triggers up to 8 levels deep. If a trigger gets into an infinite loop, SQLBase detects this recursive action when the 8-level nesting maximum is reached and returns an error to the user. For example, you could activate a trigger by attempting to insert into the table T1 and the trigger could call a stored procedure which also attempts to insert into T1, recursively activating the trigger.

If a set of nested triggers fails at any time, SQLBase rolls back the command which originally activated the triggers.

By defining a trigger to a procedure, you can set default or derived column values in INSERT and UPDATE operations. When you create the trigger for this purpose using the CREATE TRIGGER command, the trigger must comply with these rules:

- The trigger must be executed BEFORE the INSERT or UPDATE operation.

    You can modify column values only with a BEFORE...ROW trigger. Because the column value must be set before the INSERT or UPDATE operation, using the AFTER...ROW trigger to set column values is meaningless. Note also that the DELETE operation does not apply to modifying column values.

- For an UPDATE operation, the REFERENCING clause must contain a NEW column value for modification.

    Note that it is meaningless to modify the OLD column value. For an INSERT operation, all values are new by default.

- The trigger must be specified with the FOR EACH ROW clause.

    Column values cannot be passed to triggers that are specified with the FOR EACH STATEMENT clause. Note that you must change the default which is FOR EACH STATEMENT.

- You must pass the column you want to modify as a receive parameter to the procedure that is triggered. See examples in section that follows.

    Note when using procedure logic, the column value may be modified under some conditions, or left the same under other conditions.

Using receive parameters

When you set default/derived column values when using receive parameters in triggered stored procedures, note these restrictions:

- Only columns can be used for receive parameters.

- Delete triggers cannot use receive parameters.

- After triggers cannot use receive parameters.

- Old column values cannot be passed as receive parameters.

In this example, a trigger modifies a column value from null to five.

```
CREATE TABLE t1 (c1 int);

CREATE TRIGGER tg1 before insert on t1
(EXECUTE inline (c1)
PROCEDURE p1 static
parameters
  receive number : n
actions
  set n = 5)
for each row;

insert into t1 values (null);
```

This next example applies the same rule as the first example when setting a column with derived values. In the example, column c3 is set to the sum of the other two columns.

```
CREATE TRIGGER tg1 before insert on t1

(EXECUTE inline (c1, c2, c3)
PROCEDURE p1 static
parameters
  number : c1
  number : c2
  receive number: c3
actions
  set c3 = c1 + c2)
for each row;
```

General restrictions

Following are general restrictions when creating triggers with stored or inline procedures:

- You cannot use dynamic stored procedures with triggers.

- A trigger cannot call a non-stored procedure. You must always store static procedures with the STORE command.

- You cannot pass SQL @ functions as parameters to procedures that are referenced by triggers.

- You cannot pass expressions and constants as receive parameters to triggered procedures. Only columns can be passed as receive parameters to modify their values.

- It is recommended that you not define a trigger to a procedure that modifies any table referenced in a DML statement with a subselect or WHERE clause. For example:

```
DELETE FROM T1 WHERE C1 > 3
```

  In the example, note that a triggered procedure cannot modify Table T1 since it is referenced by the WHERE clause. The actions of the trigger can cause the subselect or WHERE clause of the invoking DML statement to return unpredictable results.

Triggers and Referential Integrity

Triggers are useful for implementing referential integrity constraints that are not supported by standard declarative SQLBase referential integrity (described in *Chapter 6, Referential Integrity*). For example, you can use triggers to implement an UPDATE CASCADE or UPDATE SET NULL constraint.

You can also use triggers to enforce DELETE constraints instead of implementing SQLBase declarative referential integrity DELETE constraints. For example, you can specify a delete rule for each parent/dependent relationship. This delete rule tells SQLBase what to do when a user tries to delete a row of the parent table.

Be aware, however, that SQLBase does not perform cycle or conflict checks if you use triggers to enforce referential integrity rules instead of using the SQLBase declarative referential integrity feature. Also be aware, that if you want to implement UPDATE CASCADE using triggers, you must remove the referential integrity constraints you have defined since the UPDATE statement is not allowed.

The following actions summarize the order of operation for referential integrity checks:

1. BEFORE STATEMENT trigger

2. Referential integrity check to see if a particular DML is allowed

3. BEFORE ROW trigger

4. Referential integrity operation (delete cascade, set null or restrict) that is required. For example, in a delete cascade referential integrity, the dependent rows are deleted, or in a update set null referential integrity, the dependent rows will be set null.

5. DML (update, delete)

6.   AFTER ROW trigger

7.   AFTER STATEMENT trigger

This matrix describes what triggers can be created on a table with the following declarative referential integrity rules:

Declarative RI Constraint Type	INSERT TRIGGER	DELETE TRIGGER	UPDATE TRIGGER
On Delete Cascade	Yes	No	Yes
On Delete Restrict	Yes	Yes	Yes
On Delete Set Null	Yes	Yes	*No

This is "NO" only if the column list referenced in the trigger definition contains a column which is all or a part of the foreign key for that table.

Clauses

trigger name

The name of the trigger. This can contain up to 18 characters.

BEFORE or AFTER

Specify whether to execute the trigger before or after the data modification (the invoking DML statement). In some circumstances, the BEFORE and AFTER clauses are interchangeable. However, there are some situations where you should use one clause instead of the other:

- Using the BEFORE clause is more efficient than the AFTER clause when performing data validation such as domain constraint and referential integrity checking.

- The AFTER clause can provide additional processing of table rows which do not yet exist but become available from the invoking DML statement. Conversely, it can also confirm data deletion after the invoking DELETE statement.

You can define up to sixteen triggers for each combination of table, event (INSERT, UPDATE, DELETE), time (BEFORE and AFTER), and frequency (FOR EACH ROW and FOR EACH STATEMENT). For example, you can define sixteen triggers for each BEFORE EACH STATEMENT, BEFORE EACH ROW, AFTER EACH ROW, and AFTER EACH STATEMENT, providing a total of 64 triggers. In addition, if you provide INSERT, UPDATE, and DELETE triggers to these combinations, you can have a total maximum of 192 triggers.

The following example shows trigger **tgibr01** defined BEFORE INSERT ON table **t1** FOR EACH ROW. You can define 15 more CREATE TRIGGER statements with the

same combination. You can define for example triggers **tgibr02** through **tgibr15** with inline procedures **Pibr02** through **Pibr15**.

```
CREATE TRIGGER tgibr01 BEFORE INSERT ON t1
(EXECUTE INLINE()
PROCEDURE Pibr01 STATIC
LOCAL VARIABLES
  NUMBER      n1
  NUMBER      n2
ACTIONS
  ON PROCEDURE EXECUTE
    set n2 = 11101
    call SQLImmediate('insert into t2 values (:n2) ')
    )
FOR EACH ROW
```

Note that if more than one trigger is created on the same combination of table, event, time, and frequency, be sure to use the ORDER clause. If you do not use the ORDER clause, SQLBase randomly assigns a firing order for the set of triggers.

The BEFORE and AFTER clauses have different implications and advantages for each DML operation. Here are some examples:

UPDATE	BEFORE	Can be used to verify that updated data adheres to integrity constraint rules before performing an UPDATE. If you use the REFERENCING NEW AS *new values tablename* clause of the CREATE TRIGGER command with the BEFORE UPDATE clause, then the updated values are accessible to the triggered SQL statements.
		In the trigger, you can set default column values or derived column values before performing an UPDATE. The column to be modified must be passed as a receive parameter to the triggered procedure. Read the section *Triggers and Procedures* on page *3-54*.
	AFTER	Can be used to perform operations on data just updated. For example, you can compile new projected regional sales figures after updating the address of one of your large distributors following a recent move.
		If you use the REFERENCING OLD AS *old values tablename* clause of the CREATE TRIGGER command with the AFTER UPDATE clause, then the values that existed prior to the invoking update are accessible to the triggered SQL statements.

INSERT	BEFORE	Can be used to verify that inserted data adheres to integrity constraint rules before performing an INSERT. Column values passed as parameters are visible to the triggered SQL statements but the inserted rows are not.
		In the trigger, you can set default column values or derived column values before performing an INSERT. The column to be modified must be passed as a receive parameter to the triggered procedure. Read the section *Triggers and Procedures* on page *3-54*.
	AFTER	Can be used to perform operations on the data just inserted. For example, after inserting a customer's order, you can calculate the total price of all the items ordered to see whether it exceeds the customer's credit limit.
		Both column values passed as parameters and inserted rows are visible to the triggered SQL statements.
DELETE	BEFORE	Can be used to perform operations based on the soon-to-be-deleted data. Both column values passed as parameters and deleted rows are visible to the triggered SQL statements.
	AFTER	Can be used to confirm the deletion of data. Column values passed as parameters are visible to the triggered SQL statements, but the deleted rows are not.

Commits and autocommits from the invoking transaction of INSERT, UPDATE, and DELETE statements on tables which have triggers occur *after* trigger-related processing.

INSERT

Specify that the trigger is to be activated by an INSERT on the table.

Loading data is considered inserting.

DELETE

Specify that the trigger is to be activated by a DELETE on the table.

UPDATE

Specify that the trigger is to be activated by an UPDATE on the table.

You cannot reference the same column by more than one update trigger.

SQLBase allows you to recursively update the same table, and does not prevent you from recursively updating the same row.

If multiple update triggers are defined on a table, you can use the ORDER clause to specify the firing order for the set of triggers. If you do not specify the ORDER clause, SQLBase decides a random order in which to execute them. Be sure to use the ORDER clause when you create triggers that depend on a particular execution order.

SQLBase does not detect situations where the actions of different triggers cause the same data to be updated. For example, assume two update triggers on different columns, Col1 and Col2, of the table Tbl1. When you attempt to UPDATE all the columns of Tbl1, the two triggers are activated. Both triggers call stored procedures which update the same column, Col3 of a second table, Tbl2. The first trigger updates Tbl2.Col3 to 10 and the second trigger updates Tbl2.Col3 to 20.

Likewise, SQLBase does not detect situations where the result of an UPDATE which activates a trigger conflicts with the actions of the trigger itself. For example, consider the following SQL statement:

```
UPDATE t1 SET c1 = 10 WHERE c3 = 5;
```

If the trigger activated by this UPDATE then calls a procedure that contains the SQL statement:

```
UPDATE t1 SET c1 = 7 WHERE c1 = 10;
```

the result of the UPDATE which activated the trigger is overwritten.

Note: This example can lead to recursive trigger execution and should be avoided.

OF trigger column name

Activates the trigger when a user attempts to update the specified columns.

Each column can appear in at most one BEFORE and one AFTER trigger.

If you do not specify one or more column names, SQLBase assumes all of the table's columns.

ORDER sequence number

Use this clause in conjunction with a sequence number to specify the order you want a given set of triggers to be fired. The order for each set is specified in ascending order. For example, a BEFORE row action trigger with the number 0 is the first trigger to be fired for the row BEFORE the action; the trigger with the order number of 999 is the last to be fired.

If you omit the order clause for a trigger, SQLBase randomly assigns the trigger a sequence number (400 through 599) that does not conflict with any existing sequence number assignments. From that number on, the ordering remains valid.

You can define a maximum of 16 triggers for each combination of table, event, time, and frequency. In the case of UPDATE triggers, the limit is applied without regard to the column list specification. If you define more than one trigger for the same time, event and frequency, you must specify a different order number for each trigger.

Valid values for the ORDER clause sequence number are 0 through 399 and 600 through 999.

REFERENCING

Use this clause only when defining a trigger on an UPDATE operation. The REFERENCING clause provides you with a way to reference both the old column values and the new updated column values by aliasing the table on which the UPDATE operation takes place.

You cannot specify both a REFERENCING clause and a FOR EACH STATEMENT clause in the trigger definition. Because there may be multiple rows or no rows that meet the criteria, there is no one single value for SQLBase to use.

If you specify neither the old values table name nor the new values table name, SQLBase decides the values by trigger action time depending on whether you specified that the trigger should execute before or after the data modification. If you specified that the trigger should execute *before* data modification, SQLBase assumes old values. If you specified that the trigger should execute *after* data modification, SQLBase assumes new values.

OLD AS old values table name NEW AS new values table name

This is a subclause of the REFERENCING clause. It allows you with to reference the values of columns both before and after an UPDATE operation. It produces a set of old and new values which can be passed to an inline or stored procedure which contains logic used to evaluate these parameter values. An example is domain constraint checking.

Use the OLD AS clause to alias the table's column values as they existed *before* the UPDATE. Use the NEW AS clause to alias the table's column values as they exist *after* the UPDATE.

You cannot use the same name for the *old values table name* and the *new values table name*.

NEW AS new values table name OLD AS old values table name

This is a subclause of the REFERENCING clause. It provides you with the means to reference the values of columns both before and after an UPDATE operation.

Use the NEW AS clause to alias the table's column values *after* the UPDATE. Use the OLD AS clause to alias the table's column values as they existed *before* the UPDATE.

You cannot use the same name for the *new values table name* and the *old values table name*.

(EXECUTE...)

This command executes a stored or inline procedure. The procedure must be static.

stored procedure (parameters)

INLINE (parameters) procedure text

A stored procedure is a previously-compiled and named set of SQL statements that can contain flow control language. Read *Chapter 7, Procedures and Triggers* for detailed information on stored procedures. The procedure requires parenthesis to indicate a parameter set, even if the parameter set is empty.

Bind variables cannot be passed as parameters.

Columns of LONG VARCHAR data type are not supported. This means that you cannot pass a LONG VARCHAR column as a parameter to a procedure. You can only pass column names (of the table associated with the trigger) and constants. You can pass rowids as parameters. You cannot pass aggregate functions or non-aggregate functions (those that begin with an "@").

Instead of specifying a stored procedure name, you can also type in an inline procedure here. If the procedure expects input, you can pass parameter values in the *(parameters)* part of the command line.

If the procedure returns a non-zero return code, the trigger nullifies its invoking INSERT, UPDATE, or DELETE command, and the command fails with the error message associated with that return code.

An ON PROCEDURE FETCH statement is executed only if it contains receive parameters.

When a procedure is called by a trigger, SQLBase returns a runtime error if the stored procedure contains any of these commands:

> COMMIT
> ROLLBACK
> SAVEPOINT
> SET ISOLATION

FOR EACH STATEMENT or ROW

Specify whether the stored procedure should be executed on a per-row or per-statement basis. FOR EACH STATEMENT is the default.

A trigger defined with a FOR EACH ROW clause is activated only when the WHERE clause of an INSERT, UPDATE, or DELETE statement evaluates to TRUE and one or more rows qualify.

A trigger defined with a FOR EACH STATEMENT clause is always activated whenever a user attempts to INSERT, UPDATE, or DELETE a row of the table (even if no rows qualify for the operation's WHERE clause).

For example, assume you define the following trigger:

```
CREATE TRIGGER trg_update
AFTER UPDATE ON t1 REFERENCING OLD AS oldt1
NEW AS newt1 (EXECUTE sp1 (oldt1.c1, oldt1.c2, newt1.c1,
newt1.c2)) FOR EACH ROW;
```

If you attempt to update the table and no rows meet the conditions specified in the UPDATE statement's WHERE clause, SQLBase does *not* execute sp1.

Now assume that you defined the trigger with a FOR EACH STATEMENT clause, and the trigger called a procedure sp2 that inserts an aggregate total into a table called SUMMARY:

```
CREATE TRIGGER trg_update AFTER UPDATE ON t1 (EXECUTE sp2())
FOR EACH STATEMENT;
```

If you attempt to update the table t1 and no rows meet the conditions specified in the UPDATE statement's WHERE clause, SQLBase still executes sp2.

You cannot specify both a REFERENCING clause and a FOR EACH STATEMENT clause in the trigger definition. Because there may be multiple rows or no rows that meet the criteria, there is no one single value that SQLBase can use for the evaluation.

You cannot pass column names as parameters to stored or inline procedures called by a trigger with a FOR EACH STATEMENT clause.

INSERT statements with multiple bind value rows are treated as multiple insert statements. This is important to remember when you use the FOR EACH STATEMENT clause. For example, FOR EACH STATEMENT clause considers the following INSERT statements as three INSERT statements:

```
Insert into T1 values (:1)
\
1
2
3
/
```

Examples

These trigger examples use the following tables EMP and JOB:

```
CREATE TABLE EMP (EMP_NO integer,
    EMP_NAME   varchar(18), EMP_SALARY decimal(8,2),
EMP_JOB_NO integer);

CREATE TABLE JOB (JOB_NO integer,
JOB_DESC varchar(18), JOB_MIN_SALARY decimal(8,2),
JOB_MAX_SALARY decimal(8,2));

INSERT INTO JOB values (:1,:2,:3,:4)
```

```
\
102,Programmer, 40000,55000
103,Junior Programmer,30000,45000
/
```

The triggers call a stored procedure SALARY_RULE2 which validates a salary for a given job classification.

```
STORE SALARY_RULE2
PROCEDURE: SALARY_RULE2 static
Parameters
    Number: nJob
    Number: nSalary
Local Variables
    Sql Handle: hSql
    Number: nFetchStatus
    Number: nMax
    Number :nMin
Actions
    Call SqlConnect(hSql)
Call SqlPrepare(hSql,'SELECT JOB_MAX_SALARY,\
    JOB_MIN_SALARY from JOB \
    where JOB_NO = :nJob into :nMax, :nMin')
Call SqlExecute(hSql)
Call SqlFetchNext(hSql, nFetchStatus)
Call SqlDisconnect(hSql)

! If the salary is out of range, return the user-defined
! error code 20000 to the SQL statement which invokes the
! trigger.

If nSalary < nMin
  Return 20000
Else if nSalary > nMax
  Return 20000
Else
  Return 0;
```

INSERT. The following trigger is invoked when you run INSERT. The trigger calls SALARY_RULE2. This procedure checks to see that the inserted values fall in a range established by SALARY_RULE2.

```
CREATE TRIGGER EMP_ISRT before insert on EMP
    (EXECUTE SALARY_RULE2 (EMP.EMP_JOB_NO, EMP.EMP_SALARY)
    for each row;
```

The following insert fails because 25000 does not fall in the salary range for this job number.

```
SINSERT INTO EMP values (1, 'Bill Bates', 25000, 103);
```

This example corrects the salary so that the insert will succeed:

```
INSERT into EMP values (1, 'Bill Bates', 30000, 103);
COMMIT;
```

UPDATE. This next trigger checks the salary of an employee who are changing jobs to verify that the employees' salaries are within the salary range of the newly-assigned job. This update trigger uses the same stored procedure as the previous insert trigger.

```
CREATE TRIGGER JOB_UPDT
before update of EMP_JOB_NO on EMP
referencing old as OLD_EMP new as NEW_EMP
(execute SALARY_RULE2
(NEW_EMP.EMP_JOB_NO, NEW_EMP.EMP_SALARY))
for each row;
```

This update fails because the employee does not have the salary required for the job classification

```
UPDATE EMP set EMP_JOB_NO = 102 where EMP_NO = 1;
```

DELETE. This trigger invokes an inline stored procedure which provides referential integrity checking. It ensures no rows can be deleted from the JOB table (parent table) without first checking the dependent EMP table for dependent rows. If the inline procedure detects dependent rows, it returns user-defined error code 20001 (defined in *error.sql*) to the trigger, which causes the invoked DELETE to fail with error 20001.

```
CREATE TRIGGER JOB_DELETE before delete on JOB
  (execute inline (JOB.JOB_NO)
PROCEDURE: RI_RULE static
Parameters
     Number: nJobNo
   Local Variables
     Boolean: bExists
   Actions
     Call SqlExists('SELECT EMP_JOB_NO from EMP \
       where EMP_JOB_NO = :nJobNo', bExists)

   ! User defined error code in error.sql
   ! You cannot delete record(s) from the JOB table that
   ! have dependent record(s) in the EMP table

     If bExists
       Return 20001
     Else
       Return 0
   )
```

```
for each row;
```

The delete trigger will not allow this referential integrity violation.

```
DELETE FROM JOB where JOB_NO = 103;
```

See also

ALTER TRIGGER
DROP TRIGGER

CREATE VIEW
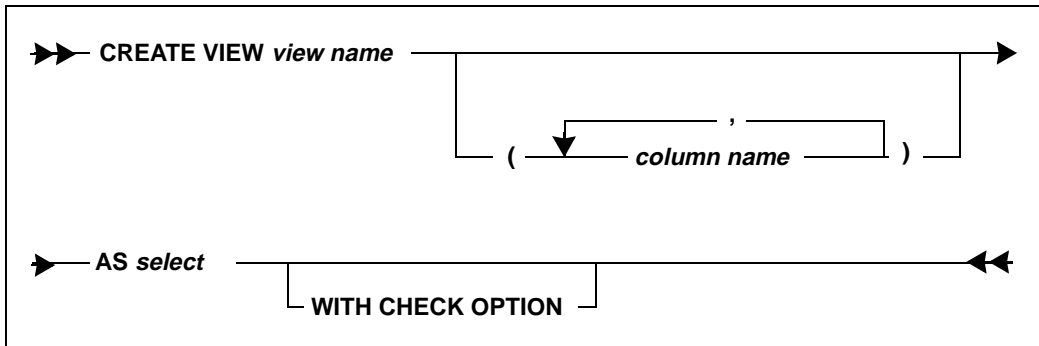
This command creates a view on one or more tables or views.

By granting certain privileges on a view instead of on base tables, you can selectively restrict access to the data in the base tables. See GRANT (Table Privileges) for more information.

You can modify tables through a view only if the view references a single table name in the FROM clause of the SELECT command, and the view columns are not derived from a function or arithmetic expression.

If you create the view from a table join, or it has derived columns, it is read-only and you cannot update the underlying tables through it.

To create a view, you must possess the corresponding SELECT privileges on the columns of the base tables that comprise the view.

If you have either SYSADM or DBA authority and create a table for another user, SQLBase assumes that unqualified names specified in your CREATE VIEW statement belong to the user, not you. For example, assume you execute the following command as SYSADM:

```
CREATE VIEW A_PAYAS
  SELECT FNAME, LNAME, SALARY
  FROM EMP, EMPSAL...
```

Since tables EMP and EMPSAL are unqualified, SQLBase assumes that the qualified table name is A.EMP and A.EMPSAL, not SYSADM.EMP and SYSADM.EMPSAL.

Like all DDL commands, this command locks system tables while executing.

Be aware that creating views can significantly increase the size of your database, since each view generally adds 20-40k to the database. Views that reference other views can be even larger.

Clauses

view name

The view name has the form:

```
authorization ID.view-name
```

The view name, including the authorization ID, must not be the name of an existing view in the database.

column name

Specify column names if you want to give different names to the columns in the view. If you do not specify column names, the columns of the view have the same names as those of the result table of the SELECT command.

If the results of the SELECT command have duplicate column names (as can occur with a join), or if a column is derived from a function or arithmetic expression, you must give names to all the columns in the view. The new column names have to appear in parenthesis after the view name.

SELECT

A SELECT command defines the view. The view has the rows that would result if the SELECT command were executed. See the description of SELECT for an explanation of this clause.

You cannot use the ORDER BY clause in a view definition.

A view is considered read-only and cannot be updated if its definition involves any of these:

- A FROM clause that names more than one table or view
- A DISTINCT keyword
- A GROUP BY clause
- A HAVING clause

- An aggregate function

WITH CHECK OPTION

This causes all inserts and updates through the view to be checked against the view definition and rejected if the inserted or updated row does not conform to the view definition. If the clause is omitted, then no checking occurs.

If a view is read-only, or if the SELECT command includes a subselect, the WITH CHECK OPTION must *not* be specified. If the view definition allows updates to some columns, the WITH CHECK OPTION applies *only* to the updates.

Examples

This view is the result of a two-table join, and is therefore read only. Since the column names of the view are not specified, they are the same as the column names in the underlying table.

```
CREATE VIEW PAY AS
    SELECT FNAME, LNAME, SALARY
    FROM EMP, EMPSAL
    WHERE EMP.EMPNO = EMPSAL.EMPNO;
```

The next view example has different column names than the underlying table.

The creator of the view does not have to be the creator of the underlying table, since the fully-qualified table name is used. Using the fully-qualified name is a good idea when creating views if the views will be used by a variety of users.

Since this view references only one table, you could update the EMP table through this view, given the proper privileges.

```
CREATE VIEW STARTDATES (FIRST, LAST, DOH) AS SELECT
    FNAME, LNAME, HIREDATE FROM EMP;
```

This next view contains a column (TOTSAL) which is derived from the application of an aggregate function. This makes it read only.

```
CREATE VIEW DEPT_SAL (DEPT, TOTSAL) AS
    SELECT DEPTNO, SUM(SALARY) FROM EMP, EMPSAL WHERE
    EMP.EMPNO = EMPSAL.EMPNO GROUP BY DEPTNO;
```

This view uses the WITH CHECK OPTION clause. Any update of the column ORDERDATE is checked to make sure the value is a date later than the July 5, 1994.

```
CREATE VIEW WEEK2 AS
    SELECT * FROM ORDERS
    WHERE ORDERDATE > = 05-JUL-94
    WITH CHECK OPTION;
```

The following view is created from two base tables, with one column from the second table appearing twice, but from different rows.

```
CREATE VIEW MYVIEW
      (POSITION, ARG1,ADESCRIPT, ARG2, BDESCRIPT) AS SELECT
      TABLE1.POSITION, TABLE1.ARG1,A.DESCRIPT AS ADESCRIPT,
      TABLE1.ARG2, B.DESCRIPT AS BDESCRIPT
      FROM TABLE1, TABLE2 A, TABLE2 B
      WHERE TABLE1.ARG1=A.CODE AND TABLE1.ARG2=B.CODE;
```

See also

CREATE TABLE
SELECT

DBATTRIBUTE
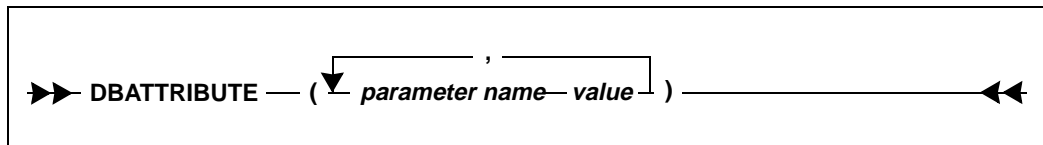
Note: This command is provided for informational purposes only. You should never need to use this command.

This command sets database specific parameters by initializing them to a specified value. Note that this command is intended for SQLBase internal use only and is required for the SQLBase Unload/Load utility when it performs database migration. Currently the supported parameters are SYSDBSequence and SYSDBTRANSID. For details on these parameters, read *Chapter 2, SQL Elements.*

Warning: If you are using this command at all, exercise extreme caution. Although you can compile and execute it like any other SQL command, you may experience serious complications and problems with data integrity.

Clauses

parameter name

The name of the database specific parameter. SQLBase currently supports SYSDBSequence and SYSTRANSID. Either one or both of these parameters can be specified. A value must be assigned to the parameter.

value

The value of the specified database parameter. Note that parameter name and value are separated by a space. Each parameter-value pair must be separated by a colon.

Example

```
DBATTRIBUTE (SYSDBSequence 1000, SYSDBTRANSID 2000);
```

DEINSTALL DATABASE

▶▶━━━━━ **DEINSTALL DATABASE *database name*** ━━━━━━━━━━━━━━◀◀

This command removes the database name from the network (removes the database name from the list of names for which the server is listening) and makes the database inactive. It updates the *dbname* keyword in *sql.ini* by deleting the database's name.

This command does not physically delete the database, but it makes the database unavailable to users.

You cannot DEINSTALL a database that is open (a database that has a user connected).

This command deinstalls the database on the server that you specified by the last SET SERVER command.

To bring the database back online, use INSTALL DATABASE.

Clauses

database name
The name of the database to deinstall.

Example

```
DEINSTALL DATABASE CUSTOMER;
```

See also

CREATE DATABASE
DROP DATABASE
INSTALL DATABASE
SET SERVER (SQLTalk command)

DELETE

```
>>── DELETE FROM ──┬── table name ──┬─────────────────────┬──>
                   └── view name ──┘  └── correlation name ─┘

>──┬──────────────────────────────────────────────┬──<<
   └── WHERE ──┬── search condition ──────────────┬──┘
              └── CURRENT OF cursor name ─────────┘
```
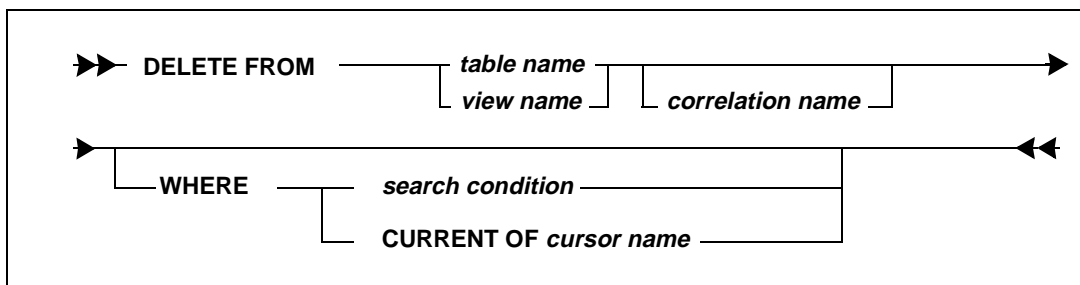
This command deletes one or more rows from a single table or view. All rows that satisfy the search condition are deleted from either the table, or the base table of the specified view.

You must possess the DELETE privilege on a table to execute this command.

Clauses

table name

Any table name can be specified for which the user has delete privileges. The name cannot identify a system table.

view name

Any view name can be specified for which the user has delete privileges. The name cannot identify a read-only view.

correlation name

A correlation name can be used within a search condition to designate the table or view.

WHERE search condition

The search condition qualifies a set of rows for deletion.

A DELETE command with this clause is called a "searched DELETE."

If you do not specify a search condition, all the rows in the specified table or view are deleted.

Read the section *Search conditions* on page *2-24* for more information.

WHERE CURRENT OF cursor name

A DELETE command with this clause is called a "positioned DELETE" or a "cursor-controlled DELETE."

This type of update requires two open cursors:

- Cursor 1 is associated with a SELECT command. The current row references the row of the most recent fetch.

- Cursor 2 is associated with the DELETE command.

A cursor-name must be associated with cursor 1 before this command can be executed.

You can only use a CURRENT OF clause if all of the following are true for the corresponding SELECT command:

- The cursor must be named or be in result set mode.

- The SELECT command cannot contain joins, GROUP BY, DISTINCT, SET functions, UNION, or ORDER BY.

- Any subselect in the SELECT command must satisfy the previous condition.

Examples

This command deletes employee 1234 from the EMP table.

```
DELETE FROM EMP WHERE EMPNO = 1234;
```

Delete employees in department 2500 from the EMPSAL table.

```
DELETE FROM EMPSAL
    WHERE EMPNO IN
    (SELECT EMPNO FROM EMP
    WHERE DEPTNO = 2500);
```

Delete all rows from the table.

```
DELETE FROM ORDERS;
```

Delete the row referenced by the current fetch, using the cursor named EMPCURSOR.

```
SET SCROLL ON;
SET CURSORNAME EMPCURSOR;
PREPARE SELECT * FROM EMPSAL;
PERFORM;
SET SCROLLROW 0;
FETCH 1;
CONNECT 2;

DELETE FROM EMPSAL WHERE CURRENT OF EMPCURSOR;
```

See also

CREATE TABLE
SET CURSORNAME (SQLTalk command)

DROP DATABASE

▶▶   **DROP DATABASE *database name*** ─────────────────────── ◀◀

This command physically deletes the entire database directory for a database *including* all associated transaction log files on the server specified by the last SET SERVER command. If the log is redirected, the log directory for the database is also completely removed.

If the database is active, DROP DATABASE also automatically DEINSTALLs a database; it deletes the database name from the network and also the *sql.in*i file.

You cannot drop a database that has any users connected to it.

Clauses

database name
The name of the database to be dropped.

Example

```
DROP DATABASE ACCTPAY;
```

See also

CREATE DATABASE
DEINSTALL DATABASE
INSTALL DATABASE
SET SERVER

DROP DBAREA

▶▶   **DROP DBAREA *dbarea name*** ─────────────────────── ◀◀

This command physically deletes the entire database area if none of its file space is currently allocated.

Clauses

dbarea name
The name of the database area to delete.

Example

```
DROP DBAREA ACCT1;
```

See also

ALTER DATABASE
CREATE DATABASE
CREATE DBAREA
SET DEFAULT STOGROUP

DROP EXTERNAL FUNCTION

```
►► ─── DROP EXTERNAL FUNCTION function name ───┬─────────────┬─── ◄◄
                                               ├─ RESTRICT ──┤
                                               ├─ CASCADE ───┤
                                               └─ FORCE ─────┘
```
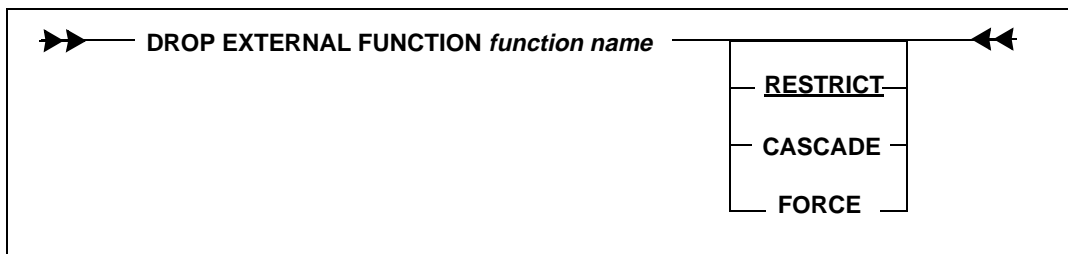
This command removes the specified external function from the database.

An external function can only be dropped by its creator or by a user with SYSADM or DBA authority.

A system catalog table, SYSDEPENDENCIES, maintain dependencies between dependent objects and determinant objects. If a stored procedure calls an external function, the stored procedure is the *dependent object* of the external function, since its existence depends on the external function. The external function is the *determinant object*, since it determines the existence of the stored procedure.

The SYSDEPENDENCIES table contains one row for each dependency between a stored procedure and an external function. SQLBase checks this table when enforcing rules for the DROP EXTERNAL FUNCTION clause options. For details on the SYSDPENDENCIES tables, refer to *Appendix A*, *System Catalog Tables*, of the *Database Administrator's Guide*.

Clauses

function name

Specify the name of the external function that you want to delete.

RESTRICT

This is the default option for dropping the external function. RESTRICT allows the DROP command to fail if the external function is a determinant object. For example,

if a procedure invokes the external function, the external function is the determinant object.

CASCADE

This option drops all dependent objects associated with the external function. For example, if a procedure invokes the external function, this option also causes the procedure to be dropped.

Note: When using the CASCADE options, be aware of the implications of dropping the external function and its dependent objects.

FORCE

This option drops the external function even if it is a determinant object, but does not drop any dependent objects. If the FORCE option is specified and there are dependent objects, the objects are marked invalid. You can check the SYSCOMMANDS system catalog table for invalid dependent objects.

For example, if a procedure invokes the external function, the external function which is a determinant object is dropped. The procedure remains, but is marked invalid.

Examples

```
DROP EXTERNAL FUNCTION FORCE;
```

See also

CREATE EXTERNAL FUNCTION
ALTER EXTERNAL FUNCTION

DROP INDEX

DROP INDEX *index name*

This command removes the specified index from the database.

Precompiled commands that reference the dropped index are *not* automatically dropped.

An index can only be dropped by its creator or by a user with SYSADM or DBA authority.

Like all DDL commands, this command locks system tables while executing.

If you drop a table's primary index, the table is incomplete and you cannot perform tasks such as inserting or deleting data.

Clauses

index name

This removes the index. Indexes on system tables cannot be dropped. The existence of views and tables are not affected.

Example

```
DROP INDEX EMP_IDX;
```

See also

CREATE INDEX

DROP STOGROUP

```
▶▶──── DROP STOGROUP *stogroup name* ─────────────────────◀◀
```

This command deletes the storage group if it is not being used by any database and it is not the default storage group. This command does not affect any existing space allocations for databases or logs.

Clauses

stogroup name
The name of the storage group to be deleted.

Example

```
DROP STOGROUP ACCTDEPT;
```

See also

ALTER DATABASE
CREATE DATABASE
CREATE STOGROUP
SET DEFAULT STOGROUP

DROP SYNONYM

```
▶▶──── DROP ──┬────────┬── SYNONYM *synonym name* ──────────▶
              └ PUBLIC ┘

 ▶──┬──────────────────────────────────┬─────────────────◀◀
    └ FOR ──┬──── TABLE ──────┬─
            ├── EXTERNAL FUNCTION ──┤
            ├──── COMMAND ────┤
            └──── PROCEDURE ──┘
```
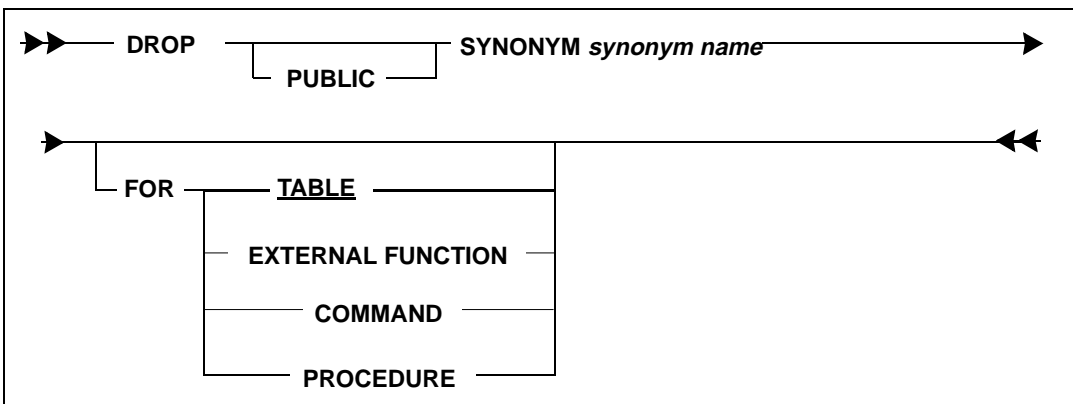
This command removes the specified synonym from the database.

Precompiled commands that reference the dropped synonym are *not* automatically dropped.

A synonym can only be dropped by its creator or by a user with SYSADM or DBA authority.

If a synonym for an external function is dropped explicitly, all procedures that refer to the synonym still remain, but are invalidated. If a synonym is dropped implicitly because the external function is dropped, all synonyms are dropped. For details, read *DROP EXTERNAL FUNCTION* on page *3-74*.

Clauses

PUBLIC
This removes the PUBLIC synonym. Views based on the synonym are also dropped.

synonym
This removes the synonym. Views based on the synonym are also dropped.

FOR TABLE, EXTERNAL FUNCTION, COMMAND, or PROCEDURE
This clause identifies the object type of the synonym. If omitted, the object type is a table by default. You must specify the keyword EXTERNAL FUNCTION in the FOR clause when dropping an external function synonym. You must specify the keyword COMMAND in the FOR clause when dropping a stored command synonym. You must specify the keyword PROCEDURE in the FOR clause when dropping a stored procedure synonym.

Examples

```
DROP SYNONYM SN1 FOR EXTERNAL FUNCTION MYFUNC;
```

Note that in the following examples, since no object type is included, the object type defaults to TABLE.

```
DROP SYNONYM ES;
DROP PUBLIC SYNONYM ES;
```

See also

CREATE SYNONYM

DROP TABLE

```
  ▶▶──────  DROP TABLE *table name* ──────────────────────────────  ◀◀
```

This command removes the specified table from the database.

Precompiled commands that reference the dropped tables are *not* automatically dropped.

A table can only be dropped by its creator or by a user with SYSADM or DBA authority.

In a database with referential constraints, dropping a table drops its primary key. This also drops any foreign keys in other tables that reference the parent table. When the parent table of the relationship is dropped, or when the primary key of the parent table is dropped, the referential constraint is also dropped.

DROP TABLE drops all constraints in which the table is a parent or dependent. Dropping a table is not the same as deleting all its rows. Instead, when you drop a table, you also drop all the relationships in which the table is involved, either as a parent or dependent. This can affect application programs that depend on the existence of a parent table, so use caution with the DROP TABLE command.

Like all DDL commands, this command locks system tables while executing.

When you drop a table, any triggers defined on that table are also dropped.

Clauses

table name

This clause drops the following:

- The specified table.
- All synonyms and indexes defined for the table.
- All privileges granted on the table.
- Any views whose definition depends either partially or wholly on the dropped table.
- Any triggers defined on that table.

System tables cannot be dropped.

Examples

```
DROP TABLE EMP;
```

See also

CREATE TABLE
CREATE VIEW

DROP TRIGGER

▶▶——— **DROP TRIGGER *trigger name*** ————————————————————— ◀◀

Use this command to remove the specified trigger from the database. Dropping a trigger disables it.

You must be the owner of a table, or a user with SYSADM or DBA authority, to drop a trigger from the table.

Example

```
DROP TRIGGER trg_insert;
```

See also

CREATE TRIGGER

DROP VIEW

▶▶——— **DROP VIEW *view name*** ————————————————————————— ◀◀

This command removes the specified view from the database.

Precompiled commands that reference the dropped view are *not* automatically dropped.

An view can only be dropped by its creator or by a user with SYSADM or DBA authority.

Like all DDL commands, this command locks system tables while executing.

Clauses

view name

This removes the view from the system catalog. Also, any views whose definition depends either partially or wholly on the dropped view are also dropped. All privileges on the views are also removed.

Example

```
DROP VIEW WEEK2;
```

See also

CREATE VIEW

GRANT (Database Authority)
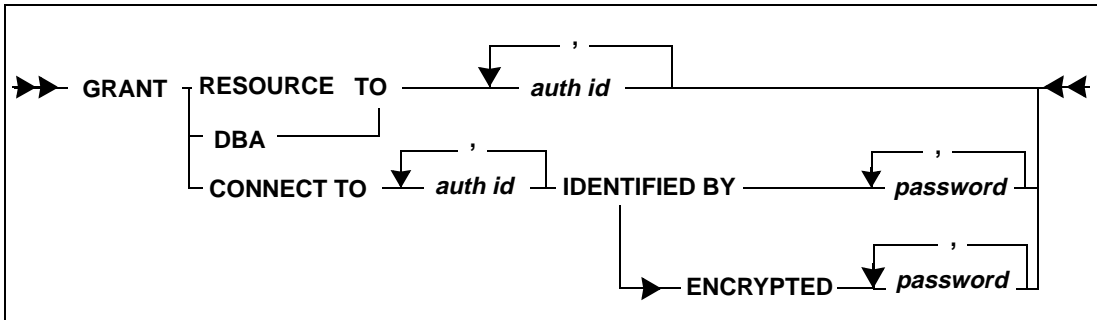
This form of the GRANT command assigns users of the database and assigns their authority level. Authority level means the types of operations a user can perform (such as logging on, creating tables, or creating users).

A different form of the GRANT command assigns privileges for individual tables.

This form of the GRANT command can only be given by SYSADM. SYSADM can create new users and change the authority levels and table privileges of existing users. This is the highest authority level and it is preassigned by SQLBase to SYSADM.

A user cannot be granted SYSADM authority. The username SYSADM cannot be changed and there can only be one SYSADM for a database. The only thing that can be changed for SYSADM is the password.

If you GRANT a user the RESOURCE or DBA authority, it does not take effect until the next time the user connects.

When a database is unloaded, the GRANT statements are unloaded. Passwords remain encrypted in the UNLOAD file and cannot be used other than in a GRANT CONNECT TO statement using the encrypted keyword.

Clauses

<authority levels>

The following authority levels can be granted by SYSADM:

CONNECT	This authority level must be granted before any other. It allows the user to log onto the database and exercise any of the privileges assigned for specific tables. The IDENTIFIED BY clause is required for granting CONNECT.
RESOURCE	This gives a user the right to create tables, to drop those tables, and to grant, modify or revoke privileges to those tables for valid users of the database. A user with RESOURCE authority automatically has all privileges on tables that he or she has created.
DBA	This level of authority automatically assigns all privileges on any table in the database to a user, including the right to grant, modify, or revoke the table privileges of any other user in the database. However, a DBA cannot create new users or change a password or authority level of an existing user. These privileges are restricted to SYSADM.

authorization id

The authorization-id is the username that gives a user authorization to connect to a database. The authorization-id SYSADM is preassigned by the system and reserved for the SQLBase "superuser."

IDENTIFIED BY password

This is required *only* when granting CONNECT authority to a user and is the phrase used to introduce the new user's encrypted password.

password

A GRANT CONNECT command must include a password. The password can be any valid SQL short identifier. To change the password of a user, grant that user CONNECT authority with the new password.

ENCRYPTED password

This is required *only* when granting CONNECT authority to a user and is the phrase used to introduce a user's password, as encrypted by an UNLOAD.

password

A GRANT CONNECT command must include a password. The password can be any valid SQL short identifier. To change the password of a user, grant that user CONNECT authority with the new password.

When a database is first created, the original creator of the database (SYSADM) is always identified by the password SYSADM. The owner of the database can change the password to a private password before granting authority to any other user.

The password is stored in the system catalog and can be read by a user with SYSADM or DBA authority.

If the GRANT CONNECT command is issued using the ENCRYPTED clause, the password is given as encrypted, for example, when unloaded in a GRANT statement. Note that passwords are encrypted when transmitted across a network.

Examples

Create two new users, JOE and JEAN. JOE is given the password SWAN and JEAN is given the password EAGLE.

```
GRANT CONNECT TO JOE, JEAN
    IDENTIFIED BY SWAN, EAGLE;
```

Give Jean the privilege to CREATE tables.

```
GRANT RESOURCE TO JEAN;
```

Give Joe DBA privilege, which includes RESOURCE privileges.

```
GRANT DBA TO JOE;
```

Change SYSADM's password.

```
GRANT CONNECT TO SYSADM
    IDENTIFIED BY CONDOR;
```

See also

GRANT (Table Privileges)
REVOKE

GRANT (Table Privileges)
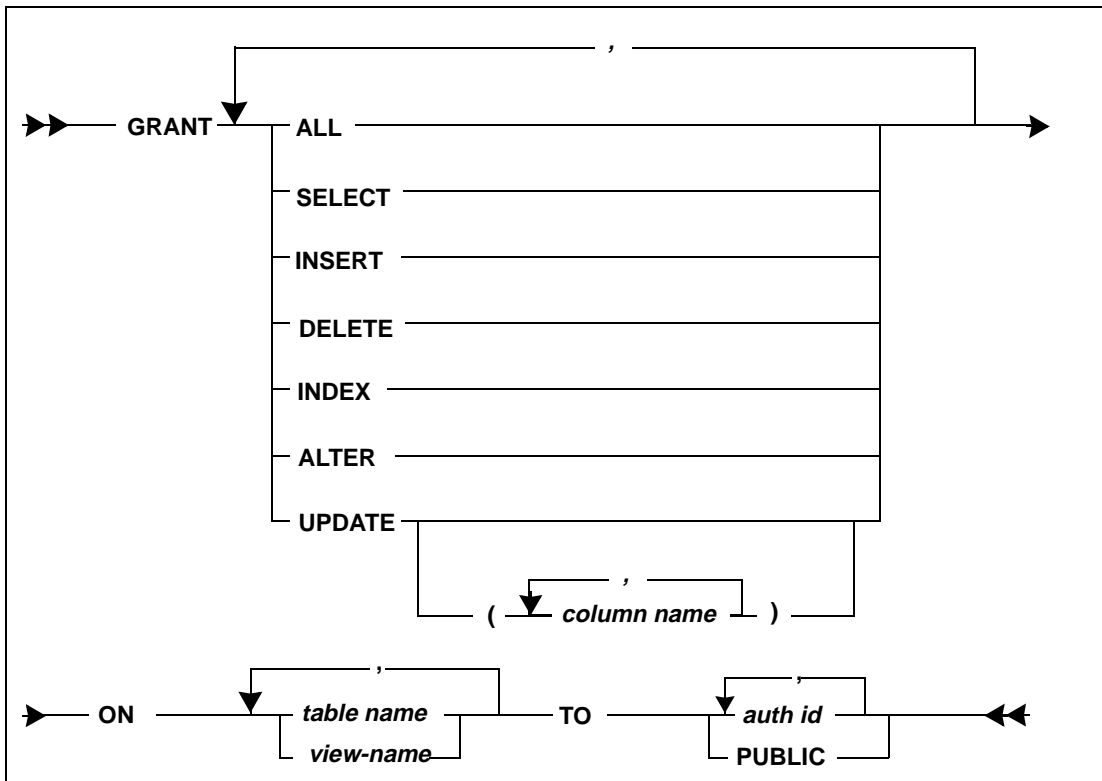
This form of the GRANT command gives a user one or more specified privileges for a table or view.

You cannot GRANT the INDEX and ALTER privileges on views.

Table privileges can be granted by any user who has the authority to do so.

- A user with DBA authority can grant privileges on any tables or views in the database.

- A user with RESOURCE authority (but without DBA authority) can grant privileges only on tables created by him or on views that are based completely on tables created by him.

- A user with only CONNECT authority cannot grant privileges. Nor does he have privileges to any tables or views unless he is explicitly granted such privileges with a GRANT command.

A different form of the GRANT command assigns privileges for a database.

The system catalog tables are owned by the creator of the database (SYSADM) so their name must be prefixed with the authorization-id SYSADM. For a description of the system catalog tables, read the *Database Administrator's Guide*.

Clauses

<privilege>

The following privileges can be assigned.

Privilege	Description
SELECT	Select data from a table or view.
INSERT	Insert rows into a table or view.
DELETE	Delete rows from a table or view.
UPDATE	Update a table and (optionally) update only the specified columns.
INDEX	Create or drop indexes for a table.
ALTER	Alter a table.
ALL	Exercise all the above for a table.

Note that you cannot GRANT the INDEX or ALTER privileges for a view. You should GRANT these privileges directly on the base tables.

table name

Table names (including an implicit qualifier) must identify a table that exists in the database.

view name

View names (including any implicit qualifier) must identify a view that exists in the database.

column name

This is a column in the tables or views specified in the ON clause. Each column name must be unqualified and each column name must be in *every* table or view identified in the ON clause.

authorization id

The authorization-id must refer to a user who has been granted at least CONNECT authority to the database.

PUBLIC

This means all users. By granting a privilege to PUBLIC, it means that all current and future users have the specified privilege on the table or view.

Examples

Give Jean privilege to read (SELECT from) the EMPSAL table and change (UPDATE) two columns, SALARY and REVIEW.

```
GRANT SELECT, UPDATE(SALARY,REVIEW)
    ON EMPSAL TO JEAN;
```

Give JOE global privileges on the tables EMP and EMPSAL.

```
GRANT ALL ON EMP, EMPSAL TO JOE;
```

Allow all users (PUBLIC) to read SYSADM.SYSTABLES.

```
GRANT SELECT ON SYSADM.SYSTABLES TO PUBLIC;
```

See also

GRANT (Database Authority)
REVOKE (Database Privileges)
REVOKE (Table Privileges)
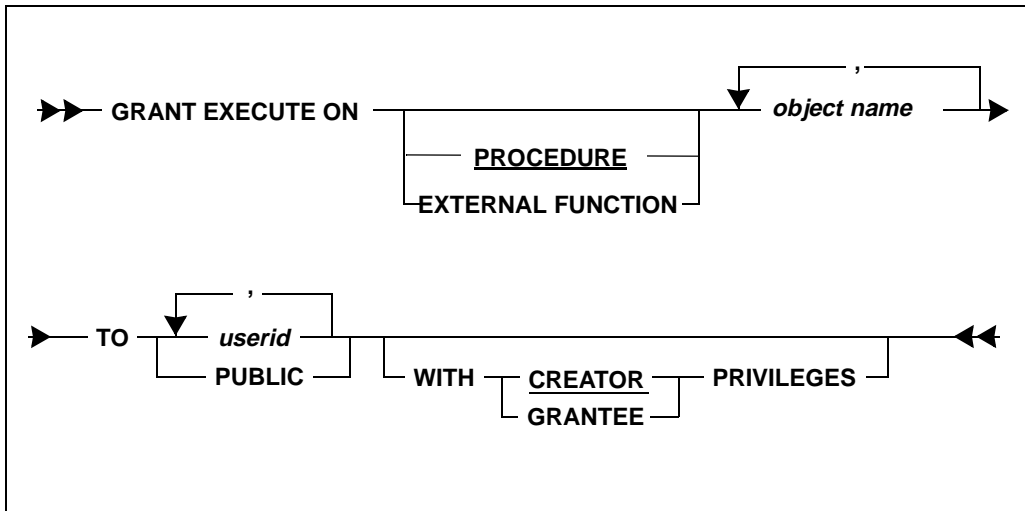
GRANT EXECUTE ON

Use this command to grant execute privilege on stored procedures or external functions to other users.

Privilege can only be granted by the owner of the stored procedure or by the DBA. The clause WITH CREATOR OR GRANTEE PRIVILEGES does not apply to external functions.

Privileges on an external function are checked at procedure compile and retrieval time.

Note: If a user has been granted EXECUTE with CREATOR privileges on a stored procedure, then the user does not need EXECUTE privileges on any external function invoked within the procedure. Only the creator of the procedure needs to have EXECUTE privileges on the external functions.

If a user has been granted EXECUTE with GRANTEE privileges on a stored procedure, the user must also have EXECUTE privileges on an external function invoked with the procedure.

Ownership of Runtime Results

If a stored procedure or external function creates a table at runtime, SQLBase determines the table's owner based on the privileges of the user executing the procedure or external function.

If the user was granted *creator's* execute privilege, then the creator of the procedure or external function is the table's owner.

If the user was granted *grantee's* execute privilege, then the user is the table's owner.

If the procedure or external function references the USER keyword, SQLBase interprets that to mean the user executing the procedure or external function, not the procedures's or external function creator (regardless of the privileges granted to the user).

Clauses

object name

The name of an existing stored procedure or external function.

PROCEDURE

If object name is omitted, the default object type is PROCEDURE.

EXTERNAL FUNCTION

If the object name is an external function, you must specify EXTERNAL FUNCTION as the object type.

TO userid or PUBLIC

The authorization ID of a user who has been granted at least CONNECT authority to the database.

Specifying PUBLIC grants *all* current and future users access to the stored procedure.

WITH CREATOR or GRANTEE PRIVILEGES

This clause applies only to procedures. Specify whether the user being granted privileges is to have the creator's (owner's) privileges or the grantee's (his own) privileges while the stored procedure function executes.

Example

This example grants all users execute privilege on the *pr_pres* stored procedure. Users assume the data access privileges of the creator, by default.

```
GRANT EXECUTE ON pr_pres TO PUBLIC;
```

The following example grants two users execute privilege on the *pr_pres* stored procedure. The users access the data using their own privileges.

```
GRANT EXECUTE ON pr_pres TO user1, user2 WITH GRANTEE
    PRIVILEGES;
```

INSERT
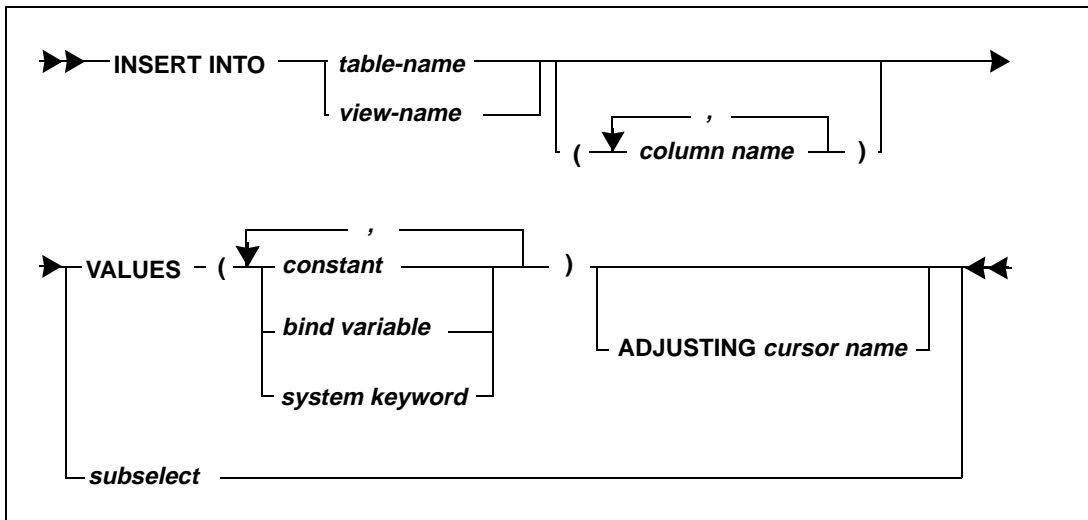
This command inserts rows of data into a table or view. For a view, the rows are inserted into the base table.

If inserting a row causes a unique index to become non-unique, or if the row does not satisfy the definition of a view that has the WITH CHECK OPTION, then the insert is not allowed.

You must possess INSERT privileges on the table to execute this command.

SQLBase itself does not restrict the number of records you can insert; this is limited only by the amount of available disk space.

If the database has referential constraints, use the following guidelines when inserting data into a parent table with a primary key:

- Do not enter non-unique values for the primary key.

- Insert only non-null values for any column of the primary key.

- Set LONG VARCHAR to a bind variable.

Use the following guidelines when inserting data into a dependent table with foreign keys:

- Each non-null value inserted into a foreign key column must be equal to a value in the primary key.

- The entire foreign key is regarded as null if any column in the foreign key is null. The INSERT statement does not perform any referential checks for a NULL foreign key, and will therefore successfully complete (as long as there are no unique index violations).

- An INSERT into either the parent table or dependent table will not work if the index enforcing the primary key of the parent table has been dropped (resulting in an incomplete table).

Read *Chapter 6, Referential Integrity* for more information.

All new data inserted into the table receives an exclusive lock.

Clauses

INTO
table name

Table names (including any qualifier) must reference a table that exists in the database. You cannot insert into system catalog tables.

column name

This is one or more column names in the specified table or view for which you provide insert values. You can name the columns in any order.

If you omit the column list, you are implicitly using a list of all the columns, in the order they were created in the table or view, and must therefore provide a value for each column.

You cannot omit a column name or insert NULL data into a column defined as NOT NULL.

view name

View names (including any qualifier) must reference a view that exists in the database but they *cannot* be any system catalog views.

VALUES

This clause contains one row of column values to be inserted. The values can be constants, bind variables, or system keywords.

Separate the column values with commas. Do *not* put a space before or after the comma.

To embed characters such as commas, surround the string with double quotes. To embed double quotes, enclose the string with additional single quotes. Refer to the next section for an example.

SQLBase will convert the values to the target data type wherever possible.

System keywords such as NULL, USER, SYSTIME, SYSDATE, SYSDATETIME cannot be used with inserts that use bind variables. However, you can enter them directly, as shown in the following example:

```
insert into T1 values (SYSDATETIME);
```

subselect

This clause inserts the rows of a result table produced by a SELECT command. The number of columns retrieved must match the number of columns being inserted. Similarly, the rows of the select must match the create definition with respect to data types and length of data. SQLBase attempts data type conversions where possible. You can use a self-referencing INSERT here; in other words, you can insert from the same table in this subselect clause.

You cannot use an ORDER BY clause in a subselect.

You cannot use a UNION clause in a subselect. However, you can create a view containing a UNION, and use the view in the subselect statement. This allows you to insert values from a SELECT statement that contains a UNION.

ADJUSTING cursor name

This clause is used for result set programming. This clause allows a user to INSERT a row without invalidating the current result set.

INSERTed rows are added to the end of the result set and the database.

You cannot perform a multi-row insert with an ADJUSTING clause and a subselect.

You cannot perform an insert with an ADJUSTING clause and a subselect with a join.

You cannot use the ADJUSTING clause on a join. The join uses a virtual table, and SQLBase cannot hold its place with a table held in memory.

Examples

This SQL command inserts one complete row into the EMP table.

```
INSERT INTO EMP VALUES (1001,'Carver','Dan',2500,01-
    APR-1994, 'Manager');
```

If all columns in the row are not being filled, you must specify the column names.

```
INSERT INTO EMP (EMPNO,LNAME,FNAME,HIREDATE)
    (1002,'Murphy','Bill',17-APR-1994);
```

The following example inserts double quotes in a string.

```
INSERT INTO EMP VALUES (1003,'Johnson','Bob "Bo"',
    2500,01-FEB-1994, 'Analyst');
```

This command uses bind variables to insert multiple rows of data.

```
INSERT INTO EMP VALUES (:1,:2,:3,:4,:5,:6)
\
1004,Drape,Jane,2600,01-FEB-1994,Programmer
1005,Foghorn,Ellen,2500,01-FEB-1994, Programmer
/
```

Use a subquery to derive rows for insertion.

```
CREATE TABLE RDEMP
  (RDNO INTEGER,
  RDLNAME CHAR(15),
  RDFNAME CHAR(10);

INSERT INTO RDEMP (RDNO, RDLNAME, RDFNAME) SELECT
    EMPNO, LNAME,FNAME FROM EMP
    WHERE DEPTNO = 2500';
```

The following example uses SQLTalk commands in an ADJUSTING clause and a result set.

```
SET CURSORNAME MYCUR;
SET SCROLL ON;
SELECT * FROM EMP;
SET SCROLLROW 1;
FETCH 2;
```

A different cursor is used to INSERT, preserving the result set.

```
CONNECT SAMPLE 2;
```

INSERT into result set.

```
INSERT INTO EMP (EMPNO,LNAME) VALUES (1006,' Bush')
    ADJUSTING MYCUR;
```

Return to the result set cursor.

```
USE 1;
```

Since the result set is unaffected, we can fetch without reissuing the SELECT.

```
FETCH 3;
```

See also

SELECT
SET CURSORNAME (SQLTalk command)

INSTALL DATABASE

> **INSTALL DATABASE** *database name*

This command assumes that the specified database exists and installs the database name on the network, adding a *dbname* keyword in *sql.ini*, and making the database accessible to users.

The database is installed on the server specified by the last SQLTalk SET SERVER command.

Clauses

database name
The name of the database to be installed.

Example

```
INSTALL DATABASE CUSTOMER;
```

See also

DEINSTALL DATABASE
DROP DATABASE
INSTALL DATABASE
SET SERVER

LABEL
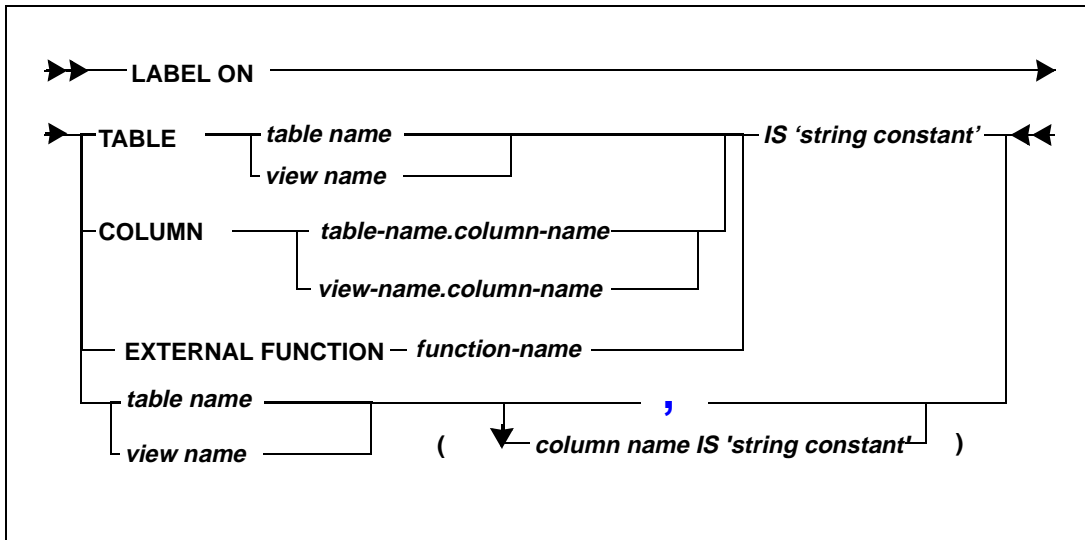
This command adds or replaces labels in the system catalog descriptions of tables, views, columns (or sets of columns), or external functions.

The system catalog can maintain a comment on every table, view, or column in the SYSTABLES or SYSCOLUMNS tables. The LABEL command places a comment in the LABEL column of the following tables: SYSTABLES, SYSCOLUMNS, or SYSEXTFUN tables.

The COMMENT ON command is like the LABEL ON command. The difference is that the REMARKS columns (maintained by COMMENT ON) is 254 characters long while the LABEL column (maintained by LABEL ON) is 30 characters long.

The LABEL column can be retrieved through an API call.

Adding labels for more than one column

Do not specify the keywords TABLE, or COLUMN. Give the table, view name and then, in parentheses, specify the label for each column. Separate each label definition with a comma.

Clauses

ON TABLE table name

You can use this to specify the name of a table that you want to add a LABEL column for.

ON TABLE view name

You can use this to specify the name of a view that you want to add a LABEL column for.

ON COLUMN table name.column name

You can use this to specify the name of a column in a table that you want to add a LABEL column for.

ON COLUMN view name.column name

You can use this to specify the name of a column in a view that you want to add a LABEL column for.

ON EXTERNAL FUNCTION function name

You can use this to specify the name of an external function that you want to add a LABEL column for.

IS 'string constant '

You can use this to specify the comment. It can be up to 30 characters.

Examples

```
LABEL ON TABLE EMP IS 'CONTAINS EMP. INFO.';
LABEL ON COLUMN EMP.DEPTNO IS 'CONTAINS DEPARTMENT NUMBER.';
LABEL ON EMP (DEPTNO IS 'CONTAINS DEPARTMENT NUMBER.',
  HIREDATE IS 'STARTING DATE');
```

The following example selects all labels from the SYSCOLUMNS system catalog table. Note that you must enclose the column name (LABEL) in double-quotes and it must be in upper-case:

```
SELECT NAME, TBNAME, "LABEL" FROM SYSCOLUMNS;
```

See also

COMMENT ON

LOAD

The external file can be split into segments, which allows you to load information to databases that might exceed single disk or system unit limits. It also lets you take advantage of available space that is spread out over several disks. Information about the file segments is contained in a load control file.
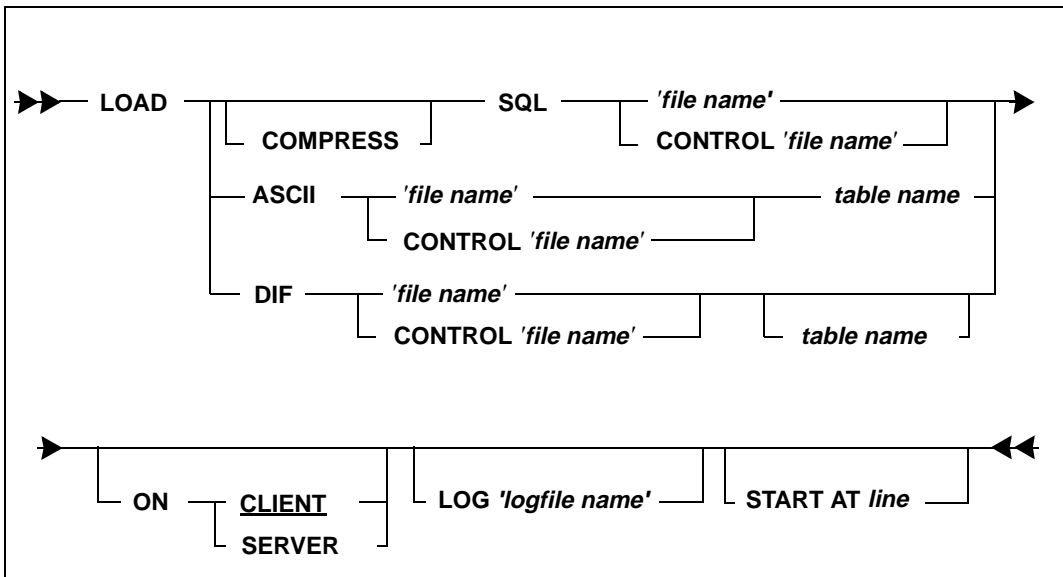
This command loads database information such as tables or data from an external file into the current database.

You can use the LOAD command to restore data from an unloaded backup file, or to enter data into the database from an external file. The external file can be in SQL, ASCII or DIF format. You can create the file either manually or with the UNLOAD command. ASCII files contain only data. DIF files can contain either data only, or both data and tables.

You can load (and unload) to the server but *not* to the client from within a stored command or procedure.

The external file can be split into segments, which allows you to load information to databases that might exceed single disk or system unit limits. It also lets you take advantage of available space that is spread out over several disks. Information about the file segments is contained in a load control file.

SQLBase does *not* issue a COMMIT operation to the database before executing the LOAD command. If AUTOCOMMIT is on, this command *does* not turn it to off before executing. Make sure that BULK is not set to ON if AUTOCOMMIT is set to ON.

A LOAD operation retains all AUTORECOMPILE settings.

If you have changed the SYSADM password, a subsequent UNLOAD and LOAD operation retains these new settings. To enhance security of the passwords in the external unload file, it is recommended that you do one of the following:

- Store the external unload file in an access-protected location on disk.
- Compress the unload file using the UNLOAD command's COMPRESS clause.

This command does not perform any referential integrity checks before executing. The checks are turned back on after the LOAD operation completes.

During the LOAD operation, if any ALTER TRIGGER commands exist in the load file, they are automatically processed.

Note if objects were dropped or altered that are referenced by triggers or procedures, SQLBase issues errors when it encounters the missing or changed object. To correct or prevent the error:

- Recreate any reference object that is dropped, or
- Restore any referenced object you changed back to its original state (known by the procedure or trigger)

When you specify the load file name, enclose it in single quotes ('). This ensures that the file name is processed correctly, even if the client and server are on different platforms.

Do not edit the load files manually. If you try to add commands such as COMMIT or ROLLBACK to the file, the load will fail.

To improve load performance, set an exclusive lock on the database first by running LOCK DATABASE. This prevents users from connecting to the database. When you are finished, run UNLOCK DATABASE.

Read the *Database Administrator's Guide* for more information on loading and unloading.

Clauses

SQL

This specifies that the load file is in SQL format and was probably created by an UNLOAD command.

A SQL format file contains the CREATE TABLE and CREATE INDEX data definition commands along with corresponding INSERT commands for each table. It does not contain the other two data manipulation commands, DELETE and UPDATE. The CREATE TABLE and CREATE INDEX commands are optional depending on whether you specified the DATA option in the UNLOAD command. If you specified the ALL option in the UNLOAD command, the SQL file does not contain any other database objects.

If the load file contains INSERT commands only, the tables into which loading occurs must exist in the database. The opposite is true for the data definition commands; if the load file includes data definition commands (such as CREATE), SQLBase creates the tables and associated indexes are created for you, so the tables *cannot* yet exist.

LOAD SQL *file-name* is equivalent to running the file as a script.

The data rows are inserted using bind variables.

When unloading, SQLBase converts binary data to ASCII characters. SQLBase marks the converted binary data with a tilde (~) character. If you want to LOAD a tilde character *as data*, mark it like this:

        ~HO~

Loading DB2 Tables

SQLTalk writes a line with $datatypes in UNLOADed tables in SQL format. The $datatypes keyword provides data type mapping for compatibility with DB2. A subsequent LOAD works for either DB2 or SQLBase.

SQLTalk allows "--" (two hyphens) in columns 1 and 2 of lines. This makes unload files produced by SPUFI (SQL Preprocessor Using File Input) on a mainframe compatible with SQLTalk. When SQLTalk sees "--" in columns 1 and 2, it ignores the line and assumes it is a comment.

Also, SQLBase interprets the "¬" operator (the *not* symbol for DB2) as a "!" (the *not* operator in SQLBase).

ASCII

This specifies a load file that contains input data organized in ASCII format. You must specify the name of the table into which the data is loaded.

Files produced with this format cannot create database objects.

You can only specify *one* load table.

ASCII format is similar to the data format in SQL except that the character fields are always delimited by double quotes ("). To enter a double quote character as data, precede it with a back slash: "He said, \"Hi.\""

DIF

The load file must contain input data organized in Data Interchange Format (DIF), a common format for spreadsheets and databases. Only one table can be loaded from a single DIF file.

The rules governing loading a file in DIF format depend on whether the file was UNLOADed with or without the DATA option.

DIF file unloaded with DATA option

The table from which the data was UNLOADed must exist. Data is then LOADed into this table. It does not matter if you specify the table name in the LOAD command or not.

DIF file unloaded without DATA option

The table *must not* currently exist. The DIF file tries to create the table named on the UNLOAD and rolls back if the table exists. However, the table name specified for the LOAD DIF command does not have to be the same as the table name specified during the UNLOAD command.

file name

The name of an existing file from which loading occurs. If you are using the file name with the ON SERVER clause, be sure to provide the volume name if it applies to your SQLBase Server environment. The following example specifies the volume name on a Netware Server:

```
db:\demo\acct1
```

table name

The name of the table into which you loaded data from an ASCII or DIF (data only) file.

COMPRESS

Use this option to load information from a compressed external file. SQLBase decompresses the information when it loads it.

This option is not valid for a DIF or ASCII file.

CONTROL

Use this clause with the load control file name to load data from a file split into multiple segments.

If you specify CONTROL, SQLBase automatically creates the load control file during the UNLOAD command execution, and puts this load control file in the same file directory as the unload control file. The load control file uses the same file prefix as the unload file segments, and is appended with a *.lcf* suffix.

If you do not supply a path, SQLBase assumes that the load control file resides in the default directory (for example, \Centura).

The load control file follows this syntax:

```
FILEPREFIX  <filename prefix>
DIR    <destination dir>
DIR    <destination dir>
```

This file provides the following information:

Parameter	Description
FILEPREFIX	The prefix of the file segment names used for the load.
DIR	The destination directory where the load file segments reside.

The following example shows a load control file for the Windows NT environment:

Example:

```
FILEPREFIX  dbs
DIR  c:\unldir\
DIR  d:\unldir\
DIR  e:\unldir\
```

In this example, there are three file segments with the following characteristics:

- a segment called **c:\unldir\dbs.1**
- a second segment called **d:\unldir\dbs.2**
- a third segment called **e:\unldir\dbs.3**

There is no SIZE parameter in this load control file. The name of this control file itself is *dbs.lcf.* SQLBase loads the information from these three segments according to their listed order.

Note: For a NetWare Server, be sure to specify the fully qualified volume name for the file segments. For example: db:\demo\dbs.1

You should use the SQLBase-generated load control file whenever possible, without making changes. However, if you need to create new file or edit the existing one (for example, if you unloaded the information from a non-SQLBase database, or have since moved the file segments to new directories since the UNLOAD), use an online editor. The file must be in ASCII format, and strictly follow the syntax shown in the example.

ON CLIENT
ON SERVER

This clause specifies whether the source file for the load is on the client or on the server. The default is ON CLIENT.

If you are loading information from multiple file segments, they must either reside on or be accessible from the same machine as the load control file.

You cannot use the ON CLIENT clause with either a SQLBase procedure or
SQLWindows program; with these two applications, you must use ON SERVER.

LOG

Use this option to automatically create a message log file. This message log file
documents activities occurring during the load, and also any errors. The log files
contain a timestamp for each action, summary information on the number of database
objects loaded, and a statement confirming at the load completed successfully.

Errors are logged along with the line number where the error occurred. After you fix
the error, use this line number with the START AT option to restart the load at that
point in the source file.

The default is no message log file.

If you do not specify a path for the message log file, SQLBase creates it in the
Centura home directory (for example, \Centura). If the client and server are on
different machines, SQLBase creates the message log file on the server machine.

You must specify this option to review any messages SQLBase generates during the
load operation. SQLBase does not send these messages to the screen.

START AT

For SQL or ASCII (not DIF) format files, use this option to start the load operation
from a specific line in the load input file. To find the line number where the error
occurred, use the message log generated with the LOG clause.

The line number for the START AT clause for a DDL statement must be the first line
of the DDL command. For an INSERT, the line number for the START AT clause
must be either the first line of the INSERT command or one of the line numbers that
corresponds to a row of data you are inserting.

Note that with segmented loads, the line numbers are cumulative in the load file
segments. To restart a segmented load from a specific line number, you must
determine yourself in which file segment the specified line number is located.

Examples

Load the SQL formatted external file located at the client:

```
LOAD SQL emp.sql;
```

Load the ASCII formatted external file located at the client, and load it into a table
called EMP:

```
LOAD ASCII emp.asc EMP;
```

Load the DIF formatted external file located at the client, and load it into a table
called EMP:

```
LOAD DIF emp.dif EMP;
```

Load the DIF formatted external file located at the client:

```
LOAD DIF emp.dif;
```

Load the SQL formatted external file located at the server, and log messages to a message log file located at the server:

```
LOAD SQL db.unl ON SERVER LOG db.log;
```

Load the SQL formatted external file located at the server using a control file and log messages to a message log file located at the server:

```
LOAD SQL CONTROL dbs.lcf ON SERVER LOG db.log;
```

Restart the load at the line of failure (101) using a control file and log messages to a message log file:

```
LOAD SQL CONTROL dbs.lcf ON SERVER LOG db.log START AT 101;
```

Load the DIF formatted external file located at the server, and load it into a table called T1:

```
LOAD DIF t1.unl t1 ON SERVER;
```

LOCK DATABASE

```
►►———— LOCK DATABASE ——————————————————————◄◄
```

This command exclusively locks the database you are currently connected to, preventing access by other users. This command requires at least DBA privileges.

When you issue this command, SQLBase prevents any new connections to the database by other users, and waits the default time-out amount (300 seconds) for other user connections to terminate. You can change this timeout value either by changing the value of the *locktimeout* configuration keyword or by running the SET TIMEOUT command. If the timeout limit is reached before all the concurrent user sessions terminate, you receive an error.

When your session becomes the only session active after you issue LOCK DATABASE, you have an exclusive lock on the entire database. You can acquire additional connections to the database yourself, but no other users can connect. Note that the lock is an exclusive lock; this command does not give you any other type of lock on the database.

The user associated with the current session, not the transaction, receives the exclusive lock. This means that when SQLBase performs a commit, the database lock

is not automatically released. You must either run UNLOCK DATABASE or disconnect your session to release the exclusive lock.

Issuing LOCK DATABASE before and UNLOCK DATABASE after a load operation can noticeably improve performance. You can also use database locking to accelerate other database operations which require a long time to complete, or for which a high degree of concurrency control is not necessary, such as index maintenance and referential integrity updates.

Example

The following example shows how you can improve a LOAD command's performance by issuing LOCK DATABASE and UNLOCK DATABASE.

```
CONNECT ACCTSDB1 SYSADM/SYSADM;
LOCK DATABASE;
LOAD SQL accts.unl;
UNLOCK DATABASE;
```

PROCEDURE:
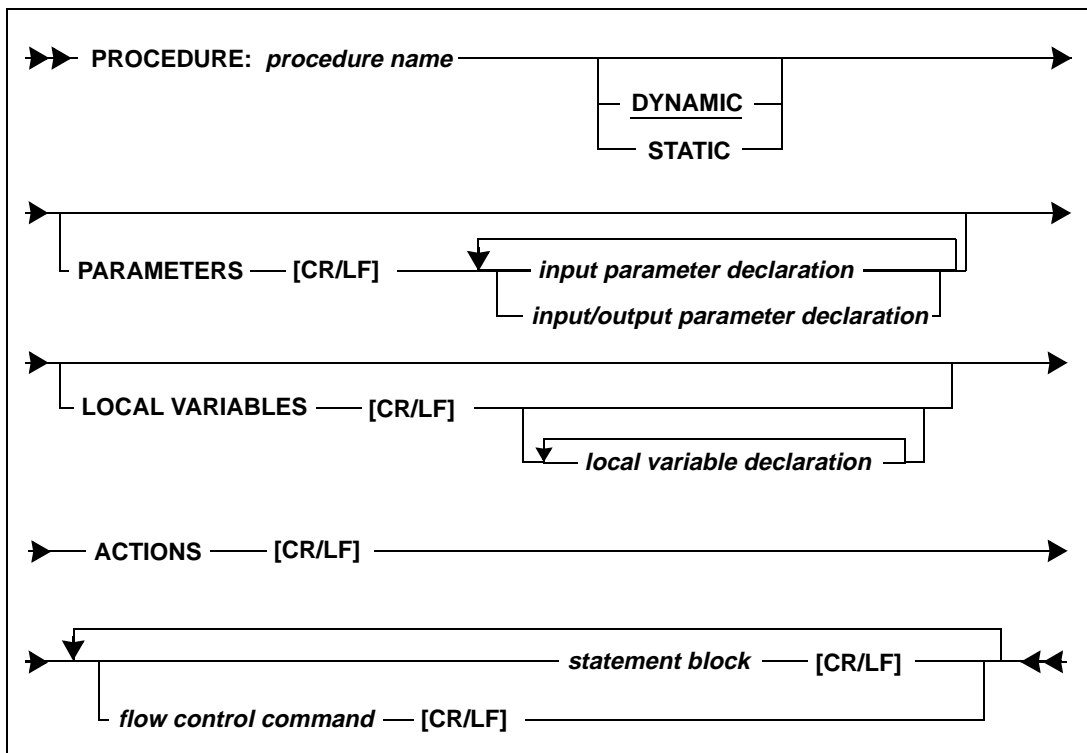
Use this command to create a procedure. The procedure can only access table and views accessible to the creator.

If you have either SYSADM or DBA authority and create an object for another user to be used in a procedure, SQLBase assumes that unqualified names specified in your PROCEDURE statement belong to the user.

For example, if your procedure references a table called EMP created for user A, SQLBase assumes that the qualified table name is A.EMP, not SYSADM.EMP or DBA.EMP.

Procedures cannot perform SQL commands that require a SET SERVER command. These SQL commands are:

CREATE DATABASE
DROP DATABASE
CREATE STOGROUP
DELETE
INSTALL DATABASE
DEINSTALL DATA

Read *Chapter 7, Procedures and Triggers* for more information on procedures.

Clauses

procedure name
The name of the procedure, which can contain up to 18 characters.

"PROCEDURE" is valid as a procedure name.

STATIC
DYNAMIC
A procedure is either dynamic or static. Dynamic is the default. Read the section *Static versus dynamic procedures* on page *7-32* for detailed information on both these clauses.

PARAMETERS
Specify this clause if you want to define input or input/output parameters in the procedure. Parameters provide you with a way to pass data to and from a procedure.

CR/LF
A carriage return or line feed character.

input parameter declaration

Specify the data type and name of each input parameter in this form. A colon after the data type is optional.

```
        ┌─┬─ BOOLEAN ──────┬─ variable name ─
        │  ├─ DATE/TIME ───┤
        │  ├─ NUMBER ──────┤
        │  ├─ STRING ──────┤
        │  ├─ LONG STRING──┤
        └  └─ WINDOW HANDLE ┘
```

input/output parameter declaration (Receive)

Specify the data type and name of each input/output parameter in this format. A colon after the data type is optional.

```
 ─ RECEIVE ─┬─ BOOLEAN ──────┬─ variable name ─
            ├─ DATE/TIME ────┤
            ├─ NUMBER ───────┤
            ├─ STRING ───────┤
            ├─ LONG STRING───┤
            └─ WINDOW HANDLE ┘
```

LOCAL VARIABLES

Specify this clause if you want to define local variables in the procedure. Local variables provide temporary storage locations.

local variable declaration

Specify the data type and name of each local variable. A colon after the data type is optional.

```
 ┌─ BOOLEAN ──────┬─ variable name ─
 ├─ DATE/TIME ────┤
 ├─ NUMBER ───────┤
 ├─ STRING ───────┤
 ├─ LONG STRING ──┤
 ├─ FILE HANDLE ──┤
 ├─ SQL HANDLE ───┤
 └─ WINDOW HANDLE ┘
```

ACTIONS

This clause introduces the section in which you include statements to be executed.

flow control command

Specify one of the following Scalable Application Language (SAL) statements. You can also specify a comment with an exclamation point (!) at the beginning of the line. Read *Chapter 7, Procedures and Triggers* for a detailed description of each statement.

```
 ┌─ BREAK ─────────┬─
 ├─ CALL ──────────┤
 ├─ IF [ELSE] ─────┤
 ├─ LOOP ──────────┤
 ├─ ON ────────────┤
 ├─ RETURN ────────┤
 ├─ SET ───────────┤
 ├─ TRACE ─────────┤
 ├─ WHEN SQLERROR ─┤
 └─ WHILE ─────────┘
```

statement block

Specify the statements to execute. For a discussion of statement blocks, read the section *Actions* on page *7-7.*

Example

```
CREATE TABLE T1 (c1 integer, c2 integer);
INSERT into T1 values (1000, 2000);

PROCEDURE: P1
Parameters
  Receive Number: nOutput1
  Receive Number: nOutput2
Local Variables
  Sql Handle: hSqlCur1
  Sql Handle: hSqlCur2
  Number: nInd
Actions
  On Procedure Startup
    Call SqlConnect(hSqlCur1)
    Call SqlConnect(hSqlCur2)
  On Procedure Execute
    Call SqlPrepare(hSqlCur1, 'Insert into T1 values (7,8)')
    Call SqlPrepare(hSqlCur2, 'Select c1, c2 \
       from T1 into :nOutput1, :nOutput2')
    Call SqlExecute(hSqlCur1)
    Call SqlExecute(hSqlCur2)
  On Procedure Fetch
    If NOT SqlFetchNext(hSqlCur2, nInd)
      Return 1
    Else
      Return 0
  On Procedure Close
    Call SqlDisconnect(hSqlCur1)
    Call SqlDisconnect(hSqlCur2)
\
''
/
```

REVOKE (Database Authority)
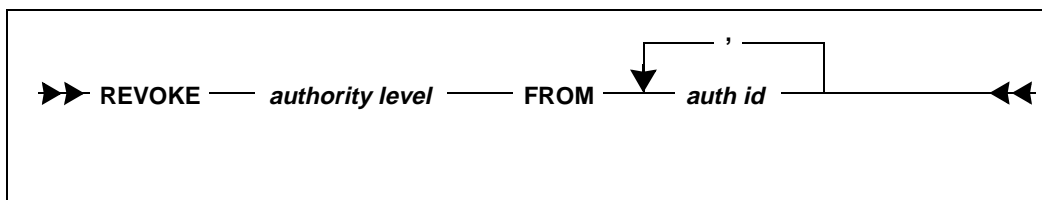
This form of the REVOKE command removes the authority level of a user who has previously been granted authority for a database.

Only a user with SYSADM authority can revoke the DBA authority of another user.

If you REVOKE a user's RESOURCE or DBA authority, it does not take effect until the next time the user connects.

Clauses

<authority level>

The authority levels DBA, RESOURCE and CONNECT can be revoked by SYSADM.

Privilege	Description
SYSADM	This authority level cannot be removed. It is assigned by the system when the database is created.
DBA	Revoking this authority means the user can no longer create or drop tables, or grant or revoke privileges from users. However, the user retains CONNECT privilege; this privilege cannot be revoked. All tables and views previously created by this user remain.
RESOURCE	Revoking this authority means the user no longer has the right to create or drop tables. However, the user retains CONNECT authority. Previously-created tables and views remain. You can revoke CONNECT privilege from a user with RESOURCE authority.
CONNECT	Revoking this authority means that the user is no longer authorized to access the database. All privileges on tables and views must be revoked from a user before revoking CONNECT authority. CONNECT authority cannot be revoked while a user owns tables.

Examples

```
REVOKE CONNECT FROM JOE, JEAN;
REVOKE RESOURCE FROM JEAN;
```

See also

GRANT (Database Authority)
GRANT (Table Privileges)
REVOKE (Table Privileges)

REVOKE (Table Privileges)
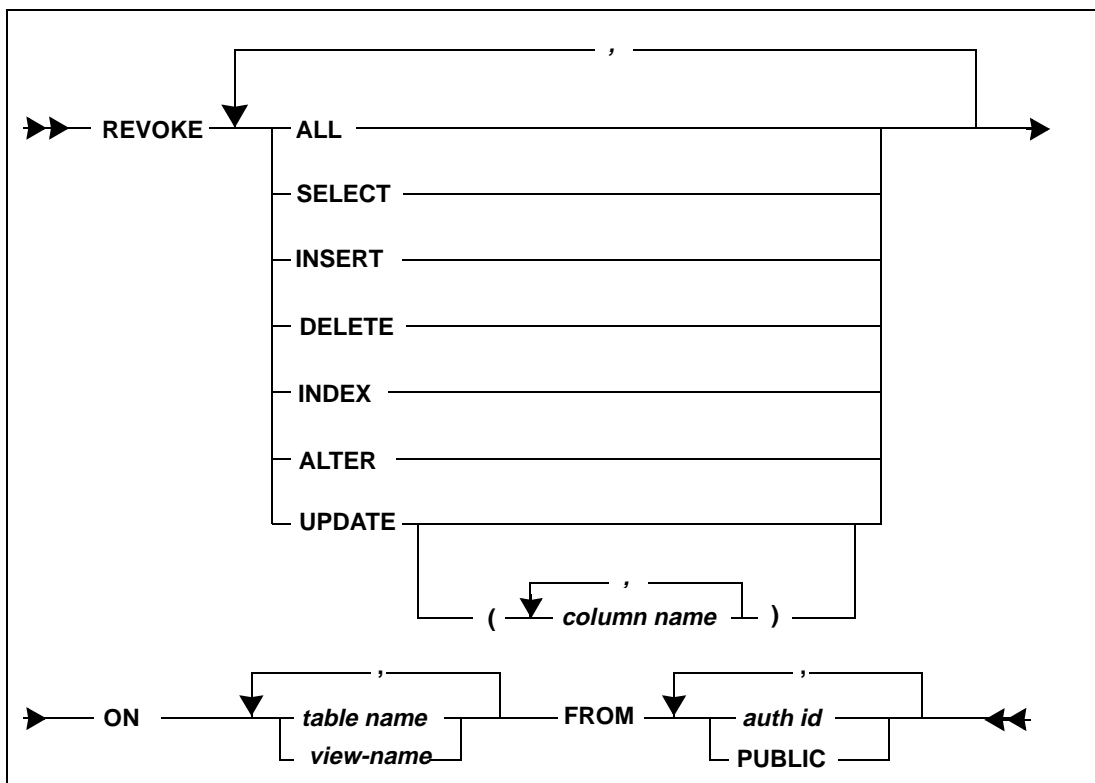
This form of the REVOKE command revokes privileges previously granted to users for a table or view.

Any user with the appropriate GRANT (Table Privileges) authority for a table can revoke the privileges for the corresponding tables or views. The creator of a table can revoke privileges on it.

Clauses

<privilege>

The following privileges can be revoked.

Privilege	Description
SELECT	Select data from a table or view.
INSERT	Insert rows into a table or view.

Privilege	Description
DELETE	Delete rows from a table or view.
UPDATE	Update a table and (optionally) update only the specified columns.
INDEX	Create or drop indexes for a table.
ALTER	Alter a table.
ALL	All of the above for a table.

ON table name

Table names (including any implicit qualifier) must identify a table that exists in the database.

ON view name

View names (including any implicit qualifier) must identify a view that exists in the database.

column name

If you specify more than one table or view, and UPDATE privileges are revoked for selected columns, then each column named must be in the specified tables or views.

FROM authorization id

The authorization id must refer to a valid user who currently has the privileges that are being revoked.

FROM PUBLIC

This keyword signifies all users. By revoking a privilege from PUBLIC, it means that all current users have the specified privilege revoked.

Examples

Prevent Jean from reading the EMPSAL table or updating the columns SALARY and REVIEW.

```
REVOKE SELECT, UPDATE(SALARY,REVIEW) ON EMPSAL FROM
    JEAN;
```

Revoke all privileges on EMP and EMPSAL from JOE.

```
REVOKE ALL ON EMP, EMPSAL FROM JOE;
```

Prevent users from reading the system catalog table SYSTABLES.

```
REVOKE SELECT ON SYSADM.SYSTABLES
    FROM PUBLIC;
```

See also

GRANT (Database Authority)
GRANT (Table Privileges)
REVOKE (Database Authority)

REVOKE EXECUTE ON
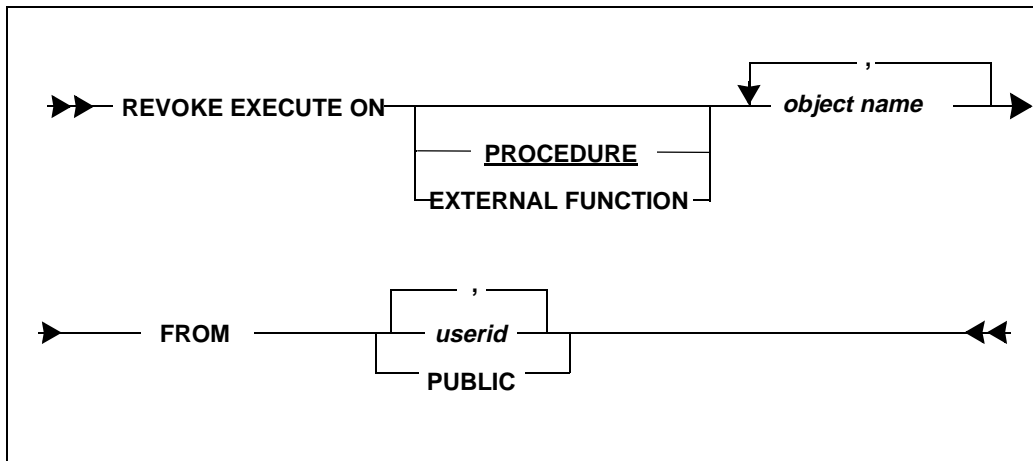
This command revokes a user's execute privilege on a stored procedure or external function.

Privilege can only be revoked by the owner of the stored procedure/external function or by the DBA.

Clauses

object name

The name of the stored procedure or external function.

PROCEDURE

If object name is omitted, the default for the object type is PROCEDURE.

EXTERNAL FUNCTION

If you are specifying an external function name, you must specify EXTERNAL FUNCTION as the object type.

FROM userid or PUBLIC

The authorization ID of one or more users who had been granted execute privilege on the stored procedure or external function.

Specify PUBLIC to revoke access privileges to the stored procedure or external function and underlying tables and views from *all* current and future users.

Example

This example revokes execute privileges on the PR_PRES stored procedure from all users. Note since no object type is included, the object type defaults to PROCEDURE.

```
REVOKE EXECUTE ON PR_PRES FROM PUBLIC;
```

This example revokes execute privilege on the PR_PRES stored procedure from two users.

```
REVOKE EXECUTE ON PR_PRES FROM USER1, USER2;
```

ROLLBACK

▶▶─── **ROLLBACK** ──────────────────────────────────◀◀
　　　　　　　　　　savepoint identifier
　　　　　　　　　　TRANSACTION <id> FORCE
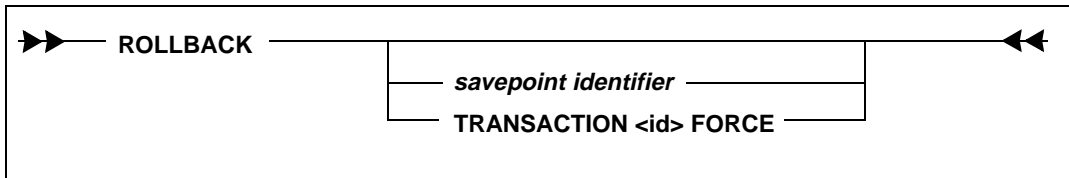
This command ends the current transaction (logical unit of work). A transaction contains one or more SQL commands that must either all be committed or none at all.

When you issue a ROLLBACK command, SQLBase aborts the current transaction. This restores the database either to the state it was in at the last COMMIT or ROLLBACK, or if none has been previously given, since the user connected to the database. The rollback applies to the work done for all cursors that the SQLTalk session or the application has connected to the database.

If you set PRESERVECONTEXT to ON for the current cursor, SQLBase preserves the cursor context after a user-initiated ROLLBACK if *both* of the following are true:

- The application is in Release Locks (RL) isolation level
- No data definition language (DDL) operation was performed

Note SQLBase does not preserve the cursor context after a system-initiated ROLLBACK, such as a deadlock, timeout, etc.

A ROLLBACK applies to all SQL commands including data definition (CREATE, DROP, ALTER) and data manipulation commands (GRANT, REVOKE, UPDATE, INSERT).

If you have CONNECT authority, you can execute the ROLLBACK command.

A ROLLBACK destroys a compiled command unless you set cursor context preservation on.

Clauses

savepoint identifier

If you specify the savepoint identifier, the transaction is rolled back to that savepoint. A savepoint is marked within a transaction by the SAVEPOINT command.

If the specified savepoint does not exist, the entire transaction is rolled back and an error is returned.

If you use the same savepoint identifier again, a ROLLBACK to that savepoint identifier will cause a rollback to the later savepoint.

Rolling back to a savepoint does *not* release locks. Rolling back without specifying a savepoint *does* release locks.

TRANSACTION <ID> FORCE

This clause forces a manual ROLLBACK of an in-doubt distributed transaction. Generally, the automatic recovery feature of the commit server daemon will resolve all transactions; you should only force a ROLLBACK as a last resort. The <ID> value is the transaction's global ID in the SYSADM.SYSPARTTRANS table.

Example

In the following example, the COMMIT statement signals the end of one transaction and the start of another. The ROLLBACK command undoes the three previous SQL commands.

```
COMMIT;
<SQL Command>
<SQL Command>
<SQL Command>
ROLLBACK ;
```

See also

COMMIT
SAVEPOINT

ROWCOUNT

▶▶   **ROWCOUNT *tablename*** ◀◀

This command returns the number of rows in a table.

The difference between this command and the SQLTalk SHOW ROWCOUNT command is that SHOW ROWCOUNT displays the number of rows in a result set, not a table.

Clauses

tablename
The name of the table.

Example

Show how many rows are in the EMP table:

```
ROWCOUNT EMP;
```
5 ROWS IN TABLE

SAVEPOINT

▶▶   **SAVEPOINT *savepoint identifier*** ◀◀

This command assigns a savepoint within the current transaction.

The ROLLBACK command can optionally specify a savepoint identifier. If an identifier is specified, the transaction is rolled back to that savepoint. If the specified savepoint does not exist, the entire transaction is rolled back and an error is returned.

The diagram on the next page illustrates the use of the SAVEPOINT command.

Rolling back to a savepoint does *not* release locks. Rolling back without specifying a savepoint *does* release locks. A SAVEPOINT for one transaction does not affect a SAVEPOINT on another transaction.

If you are using distributed actions, the SAVEPOINT applies to all the databases which participate in that transaction. For example, if a user is connected to both database A and database B and sets a SAVEPOINT on database B, a ROLLBACK to that SAVEPOINT will rollback actions on both databases.

Clauses

savepoint identifier

The savepoint is identified by a long identifier that be can be up to 18 characters in length.

If the same savepoint-identifier is specified twice in SAVEPOINT commands within the same transaction, the transaction will rollback to the location of the most-recent savepoint when the ROLLBACK command is given (the first savepoint is forgotten).
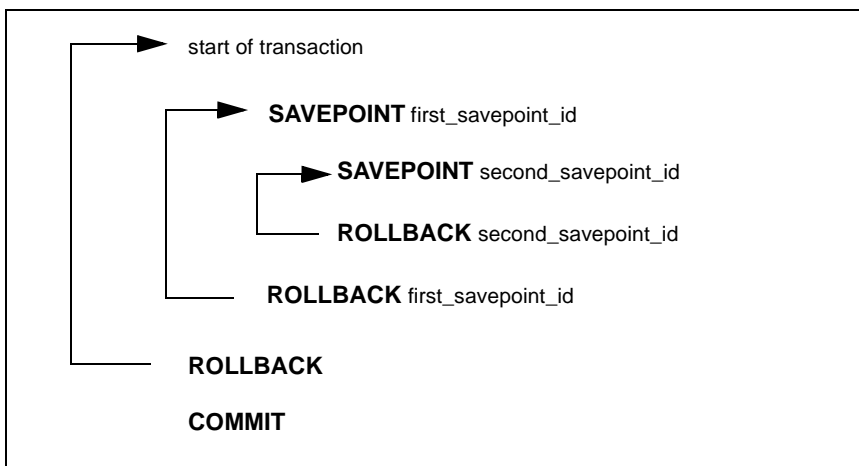
Example

This example shows the COMMIT, ROLLBACK, and SAVEPOINT commands used in a C program.

```
/* Example of savepoint use */

/* Start of application is an implicit begin
/* transaction */
for (;;)
{
/* Process 1st screen */
/* If non-fatal error encountered processing 1st screen,
   rollback work done so far and reprocess 1st screen. */
if non_fatal_error
  {
   sqlcex(cur, "ROLLBACK", 0);
   continue;
  }
}
```

```
/* 1st screen successfully processed. Set SAVEPOINT so we
    don't have to reprocess 1st screen if subsequent
    errors are encountered. */
sqlcex(cur, "SAVEPOINT screen1", 0);
for (;;)
{
/* Process 2nd screen */

/* If non-fatal error encountered processing 2nd */
/* screen,rollback work done so far for 2nd screen */
/* and reprocess 2nd */screen. */
if non_fatal_error
{
  sqlcex(cur, "ROLLBACK screen1", 0);
  continue;
}
}
sqlcex(cur, "COMMIT");
```

See also

COMMIT
ROLLBACK

SELECT

▶▶ SELECT ─┬─────────────┬─ ... expression ... ─▶
           ├─ **ALL** ───┤   name =
           └─ **DISTINCT**─┘
                          expression
                                    AS name

▶ **FROM** ─┬─ *table name* ─┬─ , ─ *correlation name* ─▶
            └─ *view name* ──┘

▶ ─┬────────────────────────────────┬─ ▶
   └─ **WHERE** *search-condition* ──┘

▶ ─┬──┬─ ▶
   └─ **GROUP BY** ─┬─ **integer constant** ─┬─ , ─┘
                    └─ **column name** ──────┘

▶ ─┬────────────────────────────────────┬─ ▶
   └─ **HAVING** *search condition* ─────┘

▶ ─┬───┬─ ▶
   └─ **ORDER BY** ─┬─ **integer constant** ─┬─ , ─┬─────────┬─┘
                    └─ **column name** ──────┘   ├─ **ASC** ─┤
                                                   └─ **DESC**─┘

▶ ─┬──┬─ ◀◀
   └─ **FOR UPDATE OF** ─┬─ *column name* ─┬─ , ───┘
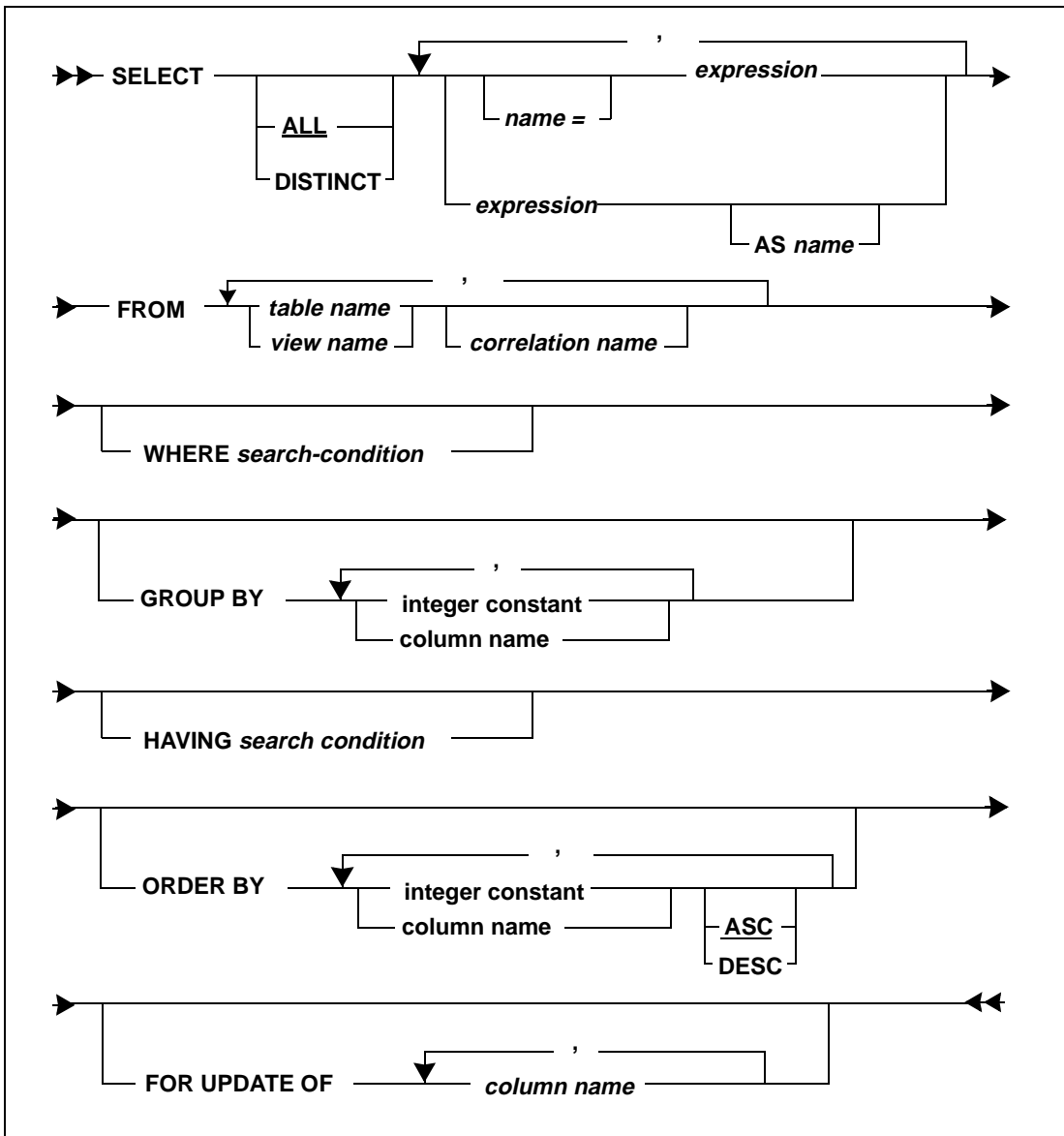
This command finds, retrieves, and displays data. It specifies the following information:

- The tables or views in the database which are searched to find the data.
- The conditions for the search.
- The sequence in which the data is output.

SELECT commands are recursive; they can be nested within the main SELECT clause. A nested SELECT command is called a subquery. You can select from other tables in the subquery.

The result of a SELECT is a set of rows called a result table which meets the conditions specified in the SELECT command.

You must have SELECT privileges on the tables and views to execute this command.

Clauses

ALL

The default for a SELECT is to retrieve ALL rows.

DISTINCT

This suppresses duplicate rows.

You cannot use the DISTINCT keyword to SELECT LONG VARCHAR data types.

You cannot use a DISTINCT keyword while in restriction mode.

You cannot perform an operation with the CURRENT OF clause on a result set that you formed with the DISTINCT keyword.

expression

This is a select list that contains expressions that are separated by commas. An expression can be:

- A column name
- A constant
- A bind variable
- The result of a function
- A system keyword

A maximum of 255 expressions are allowed in the list. Read the section *Expressions* on page *2-22* for more information.

If you are using the concatenate operator (||) to concatenate two or more strings, the result of the concatenation cannot be greater than 254 characters. SQLBase issues an error message if the resulting string size is greater than 254 characters.

A select list is usually a list of columns from one or more tables.

An asterisk (*) is a wildcard search operator that represents the entire set of columns in the tables or views specified in the FROM clause. You can also specify all the columns in a single table if '*' is qualified with the desired table name. For example, the command

```
SELECT TAB1.*, COL1 FROM TAB1, TAB2;
```

Returns all of the columns in table TAB1 and the single column COL1 from table TAB2.

Each column name in the select list must unambiguously identify a column in one of the tables or views named in the FROM clause. If a result set is derived from a select list of columns from more than one table or view, any column name in the list which is the same in two tables must be qualified by the table name to make it a unique name.

```
SELECT CUSTOMER.CUSTNO, ORDERNO FROM CUSTOMER, ORDERS WHERE
CUSTOMER.CUSTNO = ORDERS.CUSTNO;
```

In this example, the name CUSTNO appears in the CUSTOMER table and the ORDERS table. It therefore must be qualified to make it unambiguous within the SQL command.

The select list can only contain aggregate functions when the GROUP BY clause is used, or when the select list consists entirely of aggregate functions.

name = expression
expression AS name

Both of these formats assign a *name* that is used as a column heading in the output. For example:

```
SELECT CUSTOMER_NUMBER=CUSTNO FROM CUSTOMER;
```

FROM

The FROM clause contains the names of the tables or views from which the set of resulting rows are formed. Each name must identify a table or view that exists in the database.

A *correlation name* can be assigned for the table or view immediately preceding the name. Each correlation name in a FROM clause must be unique.

Correlation names are required when a search condition is executed more than once for the same table or view in a single SQL command (as in joining a table to itself or in correlated subqueries, described below). They provide a shorthand way to qualify column names.

The above SQL command can be written using the correlation name C to designate CUSTOMER and O to designate ORDERS:

```
SELECT C.CUSTNO, ORDERNO FROM CUSTOMER C, ORDERS O
    WHERE C.CUSTNO = O.CUSTNO;
```

WHERE search condition

The WHERE clause specifies a *search condition* for the base tables or views.

The search condition of the WHERE clause cannot contain any aggregate functions (unless part of a subselect). Read the section *Search conditions* on page *2-24* for more information.

You cannot use a LONG VARCHAR column in a subselect search condition.

GROUP BY

The GROUP BY clause groups the result rows of the query in sets according to the columns named in the clause.

If the column by which a grouping occurs is an expression (but not an aggregate function), you must specify a number that indicates its relative position in the select list only if the expression contains more than one column. If only one column is used in the expression, it can be used in the GROUP BY.

Aggregate functions, since they yield one value, cannot be grouping columns.

The result of a grouping is the set of rows for which all values of the grouping column are equal. NULL values in a grouping column are treated as a separate group.

If a GROUP BY clause is specified, each column in the select list must be listed in the GROUP BY clause or each column in the select list must be used in an aggregate set function that yields a single value.

The following example finds the total salary for each department, the average salary, the number of people in each department. It illustrates a GROUP BY and an equijoin (for getting the department name).

```
SELECT DEPTNO, SUM(SALARY), AVG(SALARY), COUNT(SALARY)
    FROM EMP, EMPSAL
    WHERE EMP.EMPNO = EMPSAL.EMPNO
    GROUP BY DEPTNO;
```

You cannot use GROUP BY while in restriction mode.

You cannot perform an operation with the CURRENT OF clause on a result set that you formed with a GROUP BY clause.

HAVING search condition

The HAVING clause allows a search condition for a group of rows resulting from a GROUP BY or grouping columns. If a grouping column is an expression that is *not* an aggregate function (such as SAL*10), it cannot be used in the HAVING clause.

Using the example for the GROUP BY clause, we are only interested in the departments where the average salary is greater than 30000.

```
SELECT DEPTNO, SUM(SALARY), AVG(SALARY), COUNT(SALARY)
     FROM EMP, EMPSAL
     WHERE EMP.EMPNO = EMPSAL.EMPNO
     GROUP BY DEPTNO HAVING AVG(SALARY) > 30000;
```

You cannot use a HAVING clause while in restriction mode.

The HAVING clause is useful to retrieve data that is grouped by one column, but only returns one row for each group based on the maximum of another column.

ORDER BY

This specifies the ordering, or sorting, of rows in a result table. Rows can be sorted on more than one column. The major sort is on the first column specified in the ORDER BY clause and the minor sorts are on the columns specified after that.

If the sort is on a column derived from a function or arithmetic expression, the column must be specified by an integer that signifies its relative number in the select list of the command.

Each column name (or number) can be optionally followed by ASC or DESC for ascending or descending sort sequence. ASC is the default order.

You cannot use the ORDER BY clause in a SELECT command that is a component of a UNION of SELECT commands. You cannot use the ORDER BY clause in a view definition.

You cannot perform an operation with the CURRENT OF clause on a result set that you formed with an ORDER BY clause, since ORDER BY creates virtual row tables that do not include rowids.

You cannot use an ORDER BY clause while in restriction mode.

You cannot use an ORDER BY clause in a subselect.

You cannot use string functions in an ORDER BY clause. Instead, specify the string function in the select list and then use the select list column number in the ORDER BY clause.

FOR UPDATE OF

If you are using a named cursor, this locks parts of a table so that a subsequent UPDATE or DELETE will not cause a deadlock between concurrent users. This clause is compatible with DB2.

You can UPDATE columns in the column-name list. Those columns must be a part of the table or view named in the FROM clause of the SELECT command.

When you use the FOR UPDATE OF clause, SQLBase uses update locks. An update lock reduces the possibility of deadlocks. Update locks are compatible with shared locks, but not with other update locks or exclusive locks. An update lock is released if the transaction does not immediately follow the SELECT...FOR UPDATE command with an UPDATE or DELETE. This is in contrast to exclusive locks which are held until a COMMIT or ROLLBACK.

For the read repeatability (RR) and cursor stability (CS) isolation levels, the FOR UPDATE OF clause uses update locks. The FOR UPDATE OF clause has no effect on read only (RO) or release lock (RL) isolation levels.

You can use the CURRENT OF clause in an UPDATE or DELETE command on a result set formed with the FOR UPDATE OF clause.

You cannot use the FOR UPDATE OF clause with the UNION or ORDER BY clauses or with multi-table selects.

Examples

Select all rows from the CUSTOMER table.

```
SELECT * FROM CUSTOMER;
```

Make a list of the job titles.

```
SELECT DISTINCT JOB FROM EMP;
```

Display the employee number and monthly salary of people whose annual salary is greater than $40000.

```
SELECT EMPNO,SALARY/12 FROM EMPSAL
    WHERE SALARY > 40000;
```

Find the minimum and average salary for each department.

```
SELECT DEPTNO, MIN(SALARY), AVG(SALARY)
    FROM EMP, EMPSAL
    WHERE EMP.EMPNO=EMPSAL.EMPNO
    GROUP BY DEPTNO;
```

Find the total employees hired for each quarter. This command illustrates the use of an integer when using a function in a GROUP BY clause.

```
SELECT @QUARTERBEG(HIREDATE), COUNT(EMPNO) FROM EMP
    GROUP BY 1;
```

Get the employee information for people with the same job as Drape.

```
SELECT * FROM EMP
    WHERE JOB IN
    (SELECT JOB FROM EMP WHERE LNAME = 'Drape');
```

Find the orders where the price paid was equal to the list price.

```
SELECT * FROM ORDERS X
    WHERE PRICE = (SELECT LISTPRICE FROM PARTS WHERE
    PARTS.PNUM = X.PNUM);
```

Find an order so that you can update it.

```
SELECT * FROM ORDERS
    WHERE CUSTNO=2 ORDER BY ORDERDATE;
```

Update the EMP database to show employees in department 2500.

```
SELECT LNAME FROM EMP WHERE DEPTNO = 2500 FOR UPDATE OF
    JOB;
```

```
UPDATE EMP SET JOB = '?'
    WHERE CURRENT OF EMPCURSOR;
```

```
FETCH 1;
```

```
LNAME
=========
Carver
```

UNION Clause
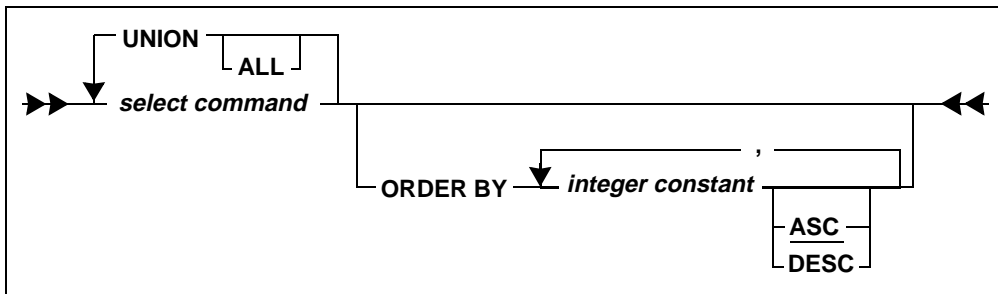
This clause merges the result of two or more SELECT commands. Any duplicate rows are eliminated.

Each result table must have the same number of columns. None of the columns can be LONG VARCHAR columns. Except for column names, the description of the corresponding column in each table must be identical.

You cannot use UNIONS in restriction mode.

You cannot perform an operation with the CURRENT OF clause on a result set that you formed with a UNION clause.

ALL

If this is specified, duplicate rows will *not* be eliminated. The result contains all the rows selected. If ALL is used, it must be repeated for every SELECT command:

```
select-cmd-1 UNION ALL select-cmd2, ....
    UNION ALL select-cmd-n.
```

ORDER BY

An ORDER BY clause sorts the final result set of rows from the UNION of two (or more) tables. When an ORDER BY clause is used with a UNION, you must use an integer specifying the sequence number of the column in the select list.

Example

This command finds the employees from department 2500 and those whose salary is more than 50000.

```
SELECT EMPNO FROM EMP WHERE DEPTNO = 2500 UNION SELECT
    EMPNO FROM EMPSAL WHERE SALARY> 50000;
```

SET DEFAULT STOGROUP
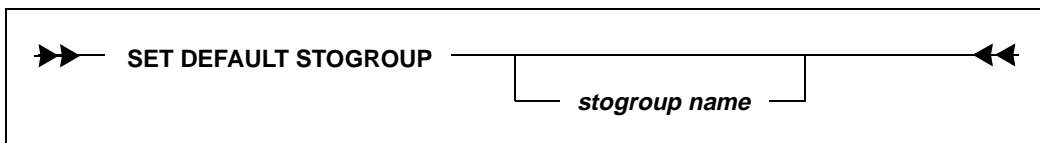
This command sets the default storage group. After a default name is given to a storage group, all subsequent CREATE DATABASE commands will cause databases to be partitioned.

Clauses

stogroup name

The name of the specified storage group. The storage group name is optional. If you omit the storage group name, the storage group is null. This allows databases to be created in the normal file system (non-partitioned).

Example

```
SET DEFAULT STOGROUP ACCTDEPT;
```

See also

ALTER STOGROUP
CREATE STOGROUP
DROP STOGROUP

START AUDIT
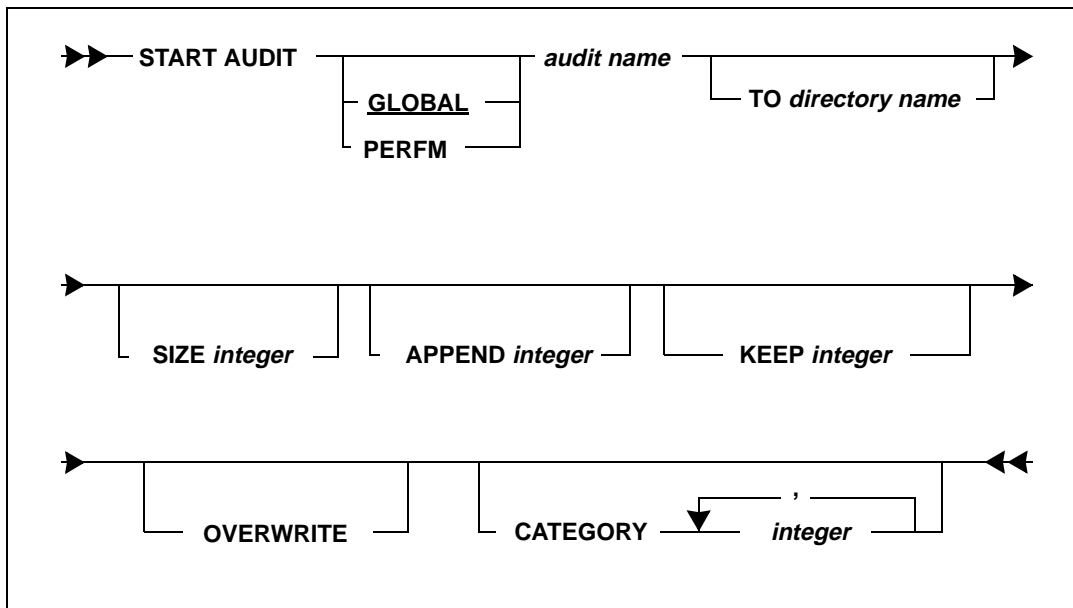
This command starts an *audit*, and creates an entry in the appropriate section of the configuration file *(sql.ini)*. An audit collects various system and performance information, and writes it to an audit file. You can also write a message to an audit file. For more details on the type of information collected, see the list of audit types and applicable categories in the *Clauses* section.

An audit remains active while the server is running. It stops when you shut down the SQLBase server, but restarts when you bring the server back up. To completely stop an audit operation, use STOP AUDIT, or delete the audit entry from the configuration file *(sql.ini)*.

You can have up to 32 active audit operations running concurrently. However, it generally is not necessary to have more than one or two, since you can record

different types of information within each audit. Also, be aware that each additional audit operation can affect performance.

This command requires a server connection.

Clauses

GLOBAL

PERFM

Enter either GLOBAL or PERFM here. These are the audit types, and determine which category options you access.

Enter GLOBAL for a global audit. A global audit includes information about the entire SQLBase server, such as rejected logons, SQL security violations, and recovery operations.

Enter PERFM for a performance audit. This type of audit tracks how long a certain operation takes, such as the length of time needed to compile and execute a SQL statement.

You can start a global audit in conjunction with an active performance audit, although they must have different names.

The default is GLOBAL.

audit name

This is an identifier that names the current audit operation. Use this audit name when you stop the audit with STOP AUDIT, or write a message to the audit file with the AUDIT MESSAGE command.

The audit name is a short identifier, and therefore can be up to eight characters long.

All concurrent audit names must be unique.

Audit files have the same name as the audit operation, with an extension .x, where *x* is an ascending value. For example, starting an audit operation with the name *myaudit* generates an audit file called *myaudit.1*.

Since the sequence of audit files is controlled by the file extension, you can theoretically have up to 1000 audit files for each individual audit (though this may not be practical in real-time applications). The audit file extension sequence ranges from *auditname. 1* to *auditname.999*. After *auditname.999*, the next file is called *auditname.0*, and then wraps around to begin the sequence again with *auditname.1*.

TO directory name

Use this clause to specify a directory that contains audit files. If you do not specify a directory, SQLBase creates the audit files in the home directory specified by the *dbdir* configuration keyword (for example, \Centura). The directory name can be enclosed in single quotes.

SIZE integer

Use this clause to specify the maximum size of each audit file, in kilobytes. When the file reaches the maximum size you specify, SQLBase automatically generates a new file. This means that if you specify a maximum size of 100 kilobytes for an audit called *myaudit*, SQLBase automatically creates and starts writing output to *myaudit.2* when *myaudit.1* reaches 100 kilobytes.

The default is 1000 kilobytes (1 megabyte).

Specify zero (0) to turn off all file size checking. This causes SQLBase to produce only one trace file of unlimited size.

APPEND integer

If you specify this clause, stop the audit with STOP AUDIT, and specify the same audit name again with START AUDIT, SQLBase continues to append audit information to the audit file having the extension you specify.

For example, assume you run STOP AUDIT *myaudit* and see that SQLBase has created four audit files, the last being *myaudit.4*. If you use the same name to start a new audit with the following command:

```
START AUDIT myaudit APPEND 4;
```

SQLBase appends the records of the new *myaudit* operation to *myaudit.4*, even though they are two separate audits.

KEEP integer

This clause tells SQLBase the number of old audit files to keep besides the current file. For example, if you specify a KEEP value of 2 for an audit called *myaudit*, SQLBase automatically deletes *myaudit.1* when it creates *myaudit.4*, since it can only keep two old files *(myaudit.2* and *myaudit.3)*.

The default is one file.

OVERWRITE

If you specify this clause, SQLBase automatically overwrites an existing audit file if it encounters one.

Unless you specify this clause, you cannot start an audit if there are existing audit files with the same name. For example, if you try to run START AUDIT *myaudit* without the OVERWRITE clause, and there is already an audit file called *myaudit.1*, SQLBase returns an error message.

If SQLBase encounters an existing file later in the audit process, (for example, *myaudit.4*), it automatically stops the audit and writes a message to the last audit file (*myaudit.3*).

CATEGORY integer

This clause identifies what types of information the audit operation records. The category options depend on what type of audit you chose: GLOBAL or PERFM.

You can record information for multiple categories by separating them by commas. Each category is independent of the others. This means that choosing category 3 does not also include information for 1 and 2.

This clause is optional. However, if you do not use it, SQLBase automatically records information for *all* categories of the audit type you chose.

GLOBAL type categories. The following table lists the valid categories for a GLOBAL audit:

Category	Data collected	Description
1	Rejected logons	Records unsuccessful login attempts. Useful to see if a user tried to access a restricted database.
2	Security violations	Tells you if a user tried to access data without proper privileges.
3	Valid logins/logoffs	Records all valid logons, telling you when a user first connected, and whether he/she disconnected. Use it to find out what users were logged on.
4	Valid connects/disconnects	Records CONNECT and DISCONNECT statements.
5	Database creates, drops, installs, and deinstalls	Records CREATE DATABASE, DROP DATABASE, INSTALL DATABASE, and DEINSTALL DATABASE commands.
6	Recovery operations	Records all ROLLBACK commands.
7	Backup and restore operations	Records all BACKUP and RESTORE commands.
8	Database Lock Manager deadlocks and timeouts	Records information about deadlocks and timeouts.
9	Table access information (queries)	Tells you which users accessed which tables.
10	Table update information (inserts, updates, and deletes)	Records database manipulation language commands (DML), and which users issued the commands.

PERFM type categories. The following table lists the valid categories for a
PERFM type audit:

Category	Data collected	Description
1	Connects and Disconnects	Tells you how long it takes users to connect to and disconnect from a database
2	SQL command compilation, execution, storage, and retrieval.	Tells you how much time SQLBase takes to compile, store, retrieve, and execute a particular SQL statement.
3	End of transaction.	Tells you when a transaction ended, and how long the transaction took. Useful for locating long-running transactions.

Examples

Start a global audit called *auditall* to record all information categories:

```
START AUDIT AUDITALL;
```

Start a performance audit to record long-running transactions, and write the output to
the \Centura directory:

```
START AUDIT PERFM LONGTRAN TO C:\CENTURA
    CATEGORY 3;
```

Start a new global security audit to track rejected logons and attempts to access data
without the proper authority. Overwrite any existing files for that audit, and keep 10
old audit files:

```
START AUDIT SECURITY KEEP 10 OVERWRITE CATEGORY 1,2;
```

Start and stop a performance trace called *testing*. Start a new performance audit with
the same name, and append the output to the last *testing* audit file (*testing.3*):

```
START AUDIT PERFM TESTING;
STOP AUDIT TESTING;
START AUDIT PERFM TESTING APPEND 3 OVERWRITE;
```

STOP AUDIT

```
►► ──── STOP AUDIT ───────────────────────────── ◄◄
                        └─ audit name ─┘
```
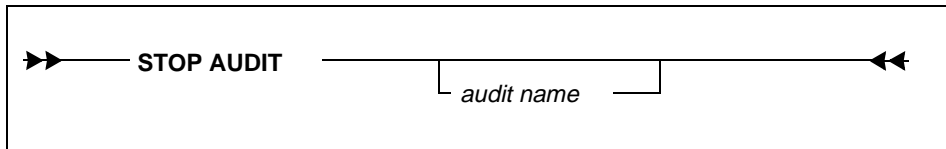
This command stops either all or the specified audit operations. This command requires a server connection.

This command removes the AUDIT entry in the server's configuration file (*sql.ini)*.

Clauses

audit name

This is the audit name. This is the name of the audit operation created with the START AUDIT *audit name* clause.

This clause is optional. If you do not designate a specific audit operation, all active audit operations are stopped.

Examples

Stop all audit operations.

```
STOP AUDIT;
```

Stop only one audit operation called *myaudit*.

```
STOP AUDIT MYAUDIT;
```

UNLOAD

```
>>─ UNLOAD ─┬─────────────┬─┬──────┬─ SQL ─┬─ 'file name' ──────────────┬─────────────┬──┬── source table ──┬─>
            └─ COMPRESS ──┘ └─ DATA ┘       └─ CONTROL 'file name' ──────┘ └─ OVERWRITE ─┘  └─ ALL ─┘

                          ┌──────┐     ASCII ─┬─ 'file name' ──────────────┬──────────────── source table ──
                          └─ DATA ┘            └─ CONTROL 'file name' ──────┘ └─ OVERWRITE ─┘

                          ┌──────┐     DIF ─┬─ 'file name' ──────────────┬──────────── source table ──
                          └─ DATA ┘          └─ CONTROL 'file name' ──────┘ └─ OVERWRITE ─┘

            ┌─────────────┐     DATABASE ─┬─ 'file name' ──────────────┬─────────────┐
            └─ COMPRESS ──┘     SCHEMA    └─ CONTROL 'file name' ──────┘ └─ OVERWRITE ─┘
                                ALL

>─┬──────────────────────────────┬─┬─────────────────────────┬─<<
  └─ ON ─┬─ CLIENT ─┬────────────┘ └─ LOG 'logfile name' ─────┘
         └─ SERVER ─┘
```
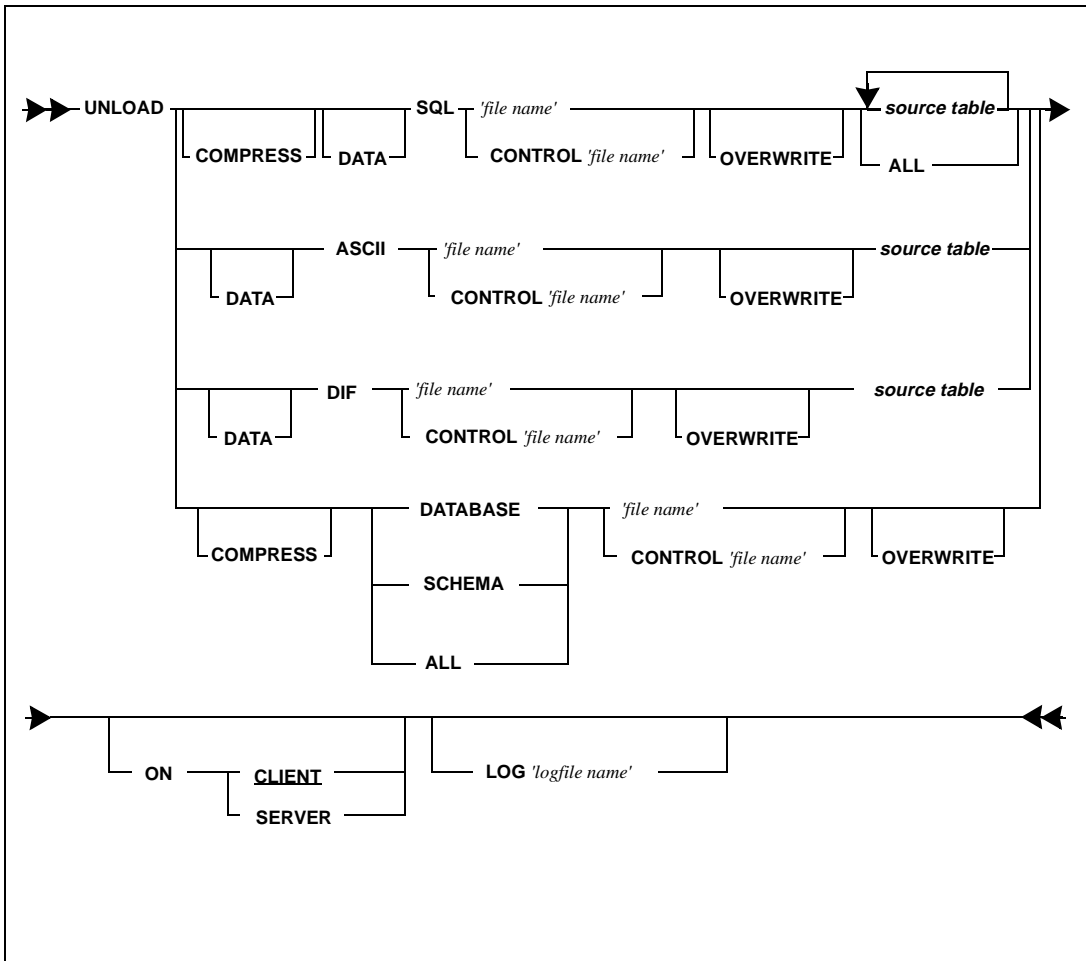
This command dumps some or all of a database to an external file.

If you are unloading data only, you can unload to a SQL, ASCII, or DIF file. Otherwise, SQLBase unloads information to a SQL formatted file.

You can unload (and load) to the server but *not* to the client from within a stored command or procedure.

With the UNLOAD command, you can back up a database or transfer data from a database to another program through interchange formats.

You can split the unload file into segments that reside on multiple disks. This allows you to unload information from a database that might exceed single disk or system unit limits, and also split an unload file into multiple files if you have smaller pockets of available disk space spread across different disks.

To restore database information from an external file, use the LOAD command.

If you are running UNLOAD with a control file for file segmentation, SQLBase automatically creates a corresponding load control file in the same directory. Use this control file to reload the information with the LOAD command.

The control file you specify must contain sufficient valid segment specifications to accommodate the total database size.

Note: SQLBase does not verify the existence of disk space availability for any of the files you create. Be sure that there is sufficient space in the directories you designate for the LOAD command and for the control file.

You should run the UNLOAD command in the Read-Only isolation level so that you do not lock out other users.

You cannot UNLOAD while in restriction mode.

LONG data type columns can only be unloaded in SQL format.

If you have changed the SYSADM password, a subsequent UNLOAD and LOAD operation retains these new settings. To enhance security of the passwords in the external unload file, it is recommended that you do one of the following:

- Store the external unload file in an access-protected location on disk.
- Compress the unload file using the UNLOAD command's COMPRESS clause.

The UNLOAD operation does not unload invalid stored commands or procedures

When triggers are encountered during the UNLOAD operation, the status of the trigger is checked. If the trigger is disabled, the UNLOAD operation generates an ALTER TRIGGER *triggername* DISABLE statement for that trigger, which immediately follows the trigger's CREATE TRIGGER statement. If the trigger is enabled, the UNLOAD operation takes no action.

When you specify the unload file name, enclose it in single quotes ('). This ensures that the file name is valid on both the client and server platforms.

Read the *Database Administrator's Guide* for more information on loading and unloading.

Clauses

DATA

In the context of the SQL format, this means that only data is written to the external file and no CREATE TABLE or CREATE INDEX statements are written. Depending on whether the file format is SQL, ASCII or DIF, this can have varying implications for the contents of the file.

SQL

This option causes the external file to be created with a series of SQL commands.

If you specify the DATA option in the command, the file contains only INSERT commands followed by data rows.

If you do not specify the DATA option, the file also includes CREATE TABLE and CREATE INDEX data definition commands along with corresponding INSERT commands for each table. It does not contain any other database object if you also specify the ALL clause.

The data rows associated with the INSERT command use bind variables.

When you use this option, multiple tables can be UNLOADed. You can specify ALL to unload all the tables of the logged-in user.

SQLTalk writes a line with $datatypes in UNLOADed tables in SQL format. The $datatypes keyword provides data type mapping for compatibility with DB2. A subsequent LOAD works for either DB2 or SQLBase tables.

When unloading, SQLBase converts binary data to ASCII characters. SQLBase marks the converted binary data with a tilde (~) character. If you want to LOAD a tilde character *as data*, you must mark it as follows:

```
~HO~
```

A continuation character "\" (backslash) can be used in unload files while entering data or commands. The continuation character works anywhere except in column 1.

The following example shows lines from a sample SQL UNLOAD file:

```
INSERT INTO SYSADM.ELECTION VALUES(
:1,
:2,
:3,
:4)
\
$datatypes NUMERIC,CHARACTER,NUMERIC,CHARACTER
1796,"Pinckney T",59,"L",
...
1796,"Washington G",2,"L",
/
```

ASCII

If you specify this, the external file contains only data, organized in ASCII format. This is true even if you do not specify the DATA option.

You can only specify *one* source table.

You cannot unload LONG data type columns in ASCII.

The following example shows lines from a sample ASCII UNLOAD file:

```
1796,"Pinckney T",59,"L"
1796,"Burr A",30,"L"
1796,"Adams S",15,"L"
1796,"Ellsworth O",11,"L"
1796,"Clinton G",7,"L"
1796,"Jay J",5,"L"
1796,"Iredell J",3,"L"
1796,"Henry J",2,"L"
1796,"Johnson S",2,"L"
1796,"Washington G",2,"L"
```

DIF

If you specify this, the external file contains data organized in Data Interchange Format (DIF) which is a common format for spreadsheets and databases.

Only one table can be unloaded in a single DIF file.

If you specify the DATA option, only the data in the table is written to the file. If it is not specified, the names of the table and columns are also written.

If the file is subsequently loaded into a database with the LOAD command, SQLTalk automatically creates the table and columns into which the data is loaded.

You cannot unload LONG data type columns in DIF.

file name

This is the name of the file into which unloading occurs. If the file already exists, you must specify the OVERWRITE clause. If you are using the file name with the ON SERVER clause, be sure to provide the volume name if it applies to our SQLBase Server environment. The following example specifies the volume name on a NetWare Server:

```
db:\demo\acct1
```

source table

This is the name of the table from which data is unloaded. The table must exist in the current database. The current database is the database to which you are connected at sign-on or with the most recent CONNECT or USE command.

If you specify a source table list, you must separate the table names with blanks. This is not applicable for ASCII and DIF formats.

The source table can also be a view or a synonym.

ALL

This option unloads all tables and indexes belonging to the connected user and is only applicable for the SQL format. It does not unload any other database objects.

DATABASE

This unloads the entire database to which the user is connected. You must be logged on as SYSADM to give this command.

You can only use UNLOAD DATABASE for SQLBase databases.

SCHEMA

This is similar to the DATABASE parameter, but it unloads only DDL (Data Definition Language) commands.

ALL

Use this command to unload all tables and indexes belonging to you. This command does not unload views and synonyms.

COMPRESS

Use this option to compress the data when you unload it.

This option is not valid for DIF or ASCII data files.

CONTROL filename

Use this clause with an unload control file name if you are unloading information into multiple file segments. SQLBase also generates a corresponding load control file. The unload control file name cannot be the same name as the load control file name, since they both reside in the same directory.

If you do not specify a path, SQLBase assumes that the control file resides in the default directory (for example, \Centura).

You create the unload control file with an online editor using the following syntax:

```
FILEPREFIX   <filename prefix>
DIR    <destination dir> SIZE   <maximum size of the unload
                         segment file in megabytes>
DIR    <destination dir>SIZE   <maximum size of the unload
                         segment file in megabytes>
```

This file provides the following information:

Parameter	Description
FILEPREFIX	The prefix of the file segment names used for the unload.
DIR	The destination directory where the unload file segments will reside.
SIZE	Maximum file segment size in megabytes. You can specify a null or integer value of 1 through 2048 megabytes. The control file must indicate a minimum aggregate size to account for all the unload data. Use the following to calculate the maximum number of bytes you can allocate: bytes = (size * 1048576) which is the maximum file size common across most systems. The last unload file segments may not use the entire size that you allocated.

The following example shows an unload control file for the Windows NT environment:

Example:

```
FILEPREFIX  dbs
DIR  c:\unldir\ SIZE  100
DIR  d:\unldir\ SIZE  50
DIR  e:\unldir\ SIZE  200
```

Note: For NetWare, you specify the fully qualified volume name for the file segments. For example: db:\demo\dbs.1

During an UNLOAD operation, this control file tells SQLBase to unload database information in the following order:

1.   100 megabytes of information to a file called **c:\unldir\dbs.1**

2.   50 megabytes of information to a file called **d:\unldir\dbs.2**

3.   200 megabytes to a file called **e:\unldir\dbs.3**

It also creates a load control file called *dbs.lcf.*

OVERWRITE

This option allows you to overwrite an existing unload file. The default is NO OVERWRITE.

ON CLIENT
ON SERVER

Use this clause to tell SQLBase where to create the destination file for the unload operation - on the client or the server machine. The default is ON CLIENT.

If you intend to use a control file, use this clause to tell SQLBase where the unload control file is. You can create the unload control file on either the client or server machines, but it must be consistent with the ON SERVER or ON CLIENT designations. As SQLBase unloads the database information, it generates the unload file segments on the same machine as this control file.

Note: Even though SQLBase creates the unload file segments on the same *machine* as the unload control file (unless you are using connected network drives), the file segments and the control file can reside in different *directories* (including network drives) on that machine.

You cannot use the ON CLIENT clause with either a SQLBase procedure or SQLWindows program; with these two applications, you must use ON SERVER.

LOG

Use this option to automatically create a message log file. If you do not designate a path for the log file, SQLBase creates it in the Centura home directory (for example, \Centura. The log files contain a timestamp for each action, summary information on the number of database objects unloaded, any errors that occurred, and a statement confirming the load completed successfully.

The default is no log file.

If the client and server are on different machines, SQLBase creates the message log file on the server machine.

Examples

Unload the EMP and DEPT (data only) tables in SQL format.

```
UNLOAD DATA SQL personnel.sql EMP DEPT;
```

Unload the EMP table (data only) in ASCII format.

```
UNLOAD ASCII emp.asc EMP;
```

Unload the EMP table in DIF format. Table and data are unloaded:

```
UNLOAD DIF table.unl EMP;
```

Unload all tables and indexes belonging to the user who issues the command:

```
UNLOAD ALL mytables.uld;
```

Unload the entire EMPLOYEE database:

```
UNLOAD DATABASE emp.uld;
UNLOAD COMPRESS DATA SQL db.unl ALL ON SERVER  LOG db.log;
UNLOAD SQL table.unl OVERWRITE t1 t2;
UNLOAD COMPRESS DATABASE db.unl ON SERVER;
UNLOAD DIF table.unl;
```

Unload the entire database at the server using a control file located also at the server:

```
UNLOAD DATABASE CONTROL contrl1.fil ON SERVER;
```

UNLOCK DATABASE

▶▶ ──── **UNLOCK DATABASE** ────────────────────────────── ◀◀

This command releases the exclusive lock acquired on the current database with the LOCK DATABASE command. After you run this command, SQLBase allows additional connections by other users again.

Issuing LOCK DATABASE before and UNLOCK DATABASE after a load operation can noticeably improve performance. You can also use database locking to accelerate other database operations which require a long time to complete, or for which a high degree of concurrency control is not necessary, such as index maintenance and referential integrity updates.

If you hold an exclusive lock on the database and disconnect your last database connection before running UNLOCK DATABASE, SQLBase automatically releases your exclusive database lock.

Example

The following example shows how you can improve a LOAD command's performance by issuing LOCK DATABASE and UNLOCK DATABASE.

```
CONNECT ACCTSDB1 SYSADM/SYSADM;
LOCK DATABASE;
LOAD SQL accts.unl;
UNLOCK DATABASE;
```

UPDATE
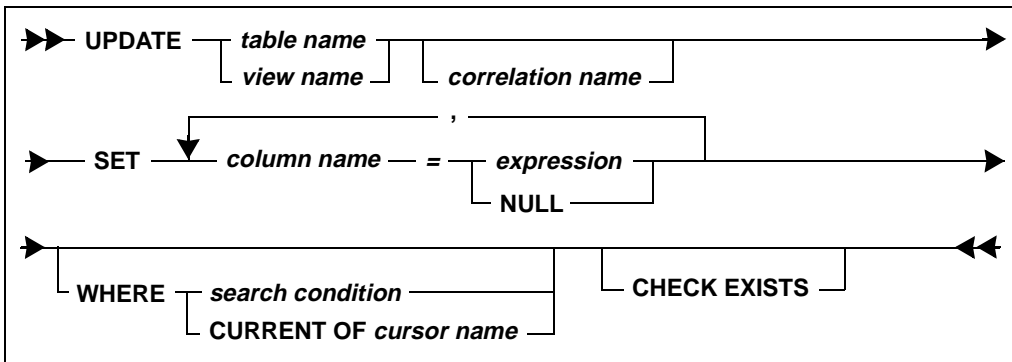
This command updates the value of one or more columns of a table or view based on the specified search conditions. You must possess the UPDATE privilege on the columns of the table or view.

The UPDATE command (for referential integrity) updates tables with primary or foreign keys. Any non-null foreign key values that you enter must match the primary key for each relationship in which the table is a dependent.

If you are updating a parent table, you cannot modify a primary key for which dependent rows exist. This would violate referential constraints for dependent tables and would leave a row without a parent. In addition, you cannot give a primary key a null value.

In a database with referential integrity, the only UPDATE rule that can be applied to a parent table is RESTRICT. This means that any attempt to update the primary key of the parent table is restricted to cases where there are no matching values in the dependent table.

If an UPDATE against a table with a referential constraint fails, an error message is returned.

When a record is updated, the fields being updated are removed from the record and new fields are added. If the new value's data size is the same as the old size, SQLBase overwrites the old field.

For more information on referential integrity, read *Chapter 6, Referential Integrity*.

Clauses

table name

This identifies an existing table.

System catalog tables can be named, but only-user defined columns can be updated.

view name

This identifies an existing view.

You cannot UPDATE a view based on more than one table.

correlation name

The correlation name must be specified if the search condition involves a correlated subquery.

column name

This identifies the columns to be updated in the table or view.

Columns derived from an arithmetic expression or a function cannot be updated.

If a view was specified with WITH CHECK OPTION, the updated row must conform to the view definition.

SET

If the update value is specified as NULL, the column must have been defined to accept null values.

If a unique index is specified on a column, the update column value must be unique or an error results. Note that for a multi-column index, it is the *aggregate* value of the index that must be unique.

If the update value is a string expression in which two or more strings are concatenated, the resulting string size cannot exceed 254 characters.

WHERE search condition

The WHERE clause specifies the rows to be updated based on a search condition.

When this clause is used, it is called a "searched UPDATE."

WHERE CURRENT OF cursor name

This clause causes the row at which a cursor is currently positioned to be updated according to the specification of the SET clause.

When this clause is used, it is called a "positioned UPDATE" or a "cursor-controlled UPDATE."

This type of update requires two open cursors:

- Cursor 1 is associated with a SELECT command. The current row references the row of the most recent fetch.
- Cursor 2 is associated with the UPDATE command.

A cursor-name must be associated with cursor 1 before this command can be executed.

You can only use CURRENT OF if all of the following are true for the corresponding SELECT command:

- The cursor must be named or be in result set mode.
- The SELECT command cannot contain joins, GROUP BY, DISTINCT, SET functions, or UNION.
- If the SELECT command contains an ORDER BY clause, the isolation level must be RE (release lock).
- Any subselect in the SELECT command must satisfy the previous condition.

CHECK EXISTS

This clause specifies to return an error if at least one row is *not* updated. This clause can be used in any context, including in chained commands.

Examples

Change employee 1004's salary.

```
UPDATE EMPSAL SET SALARY = 45000 WHERE EMPNO= 1004;
```

Give all employees in department 2500 a 10% raise.

```
UPDATE EMPSAL SET SALARY = SALARY*1.10
    WHERE EMPNO IN (SELECT EMPNO FROM EMP WHERE DEPTNO =
    2500);
```

Prefix all job titles with the letter P. Update every row in the table.

```
UPDATE EMP SET JOB= 'P'||JOB;
```

Update the row reference by the current fetch of cursor named FINDBUG.

```
UPDATE EMPSAL SET SALARY= 40000
    WHERE CURRENT OF EMPCURSOR;
```

See also

CREATE TABLE
SELECT
SET CURSORNAME (SQLTalk command)

UPDATE STATISTICS
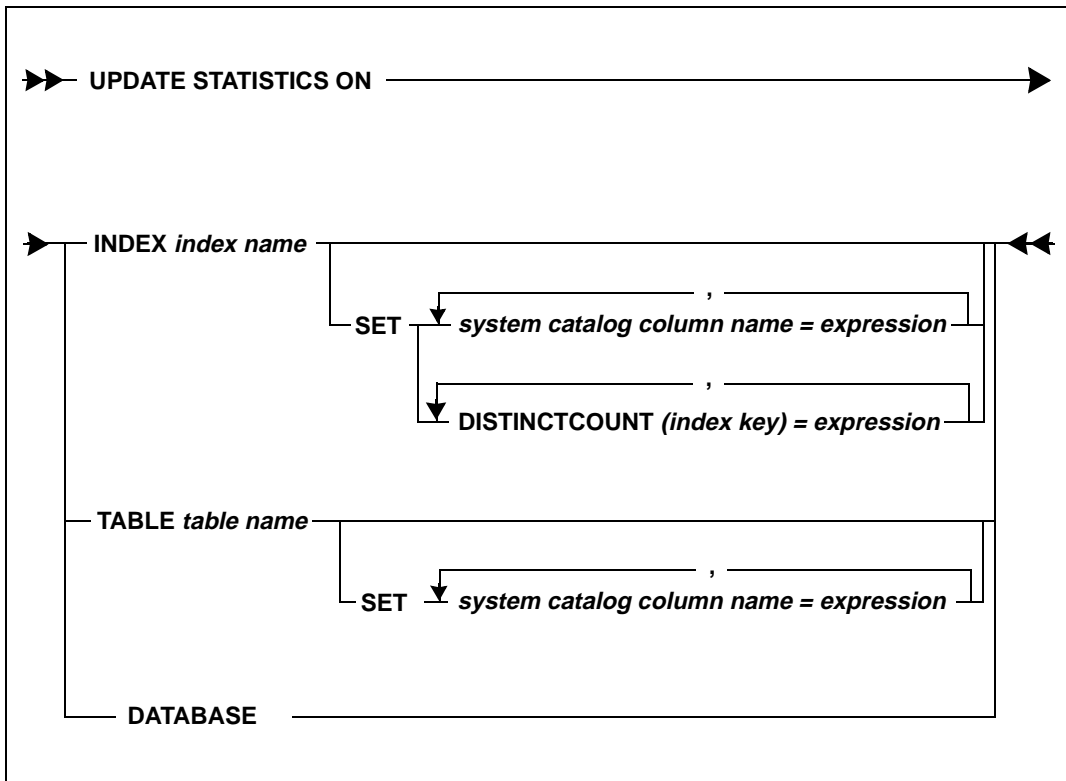
This command updates the statistics for an index, table, or database.

Generally, you should execute this command when more than 10% of your data has been modified or recently added, such as when the number of distinct key values have changed for an index.

To have your stored commands take advantage of statistics yielding better performance, you can restore your commands and then run the SQLBase-supplied RECOMPILE procedure. For more information on RECOMPILE, read Appendix B of the *Database Administrator's Guide*.

You can run SET TIME ON to check the performance before and after the statistics have been updated. This command displays the time required to obtain the results of SQL commands. The command does not show times for each function call, although it may calculate it that way.

By using the SET clause, you can enter test values for index and table statistics to simulate a production environment, without using real data. This command updates the data dictionaries with your test values as if you were using real data. The SQLBase query optimizer then uses these new statistics to find the most optimal access strategy. To restore table and index statistics to actual values, run this command again without using the SET clause.

Whether you are updating real or test values for statistics, this command updates both the internal dictionary (database control pages) as well as the external dictionary (SYSADM catalogs). No database statistics are preserved when you unload, then reload the database.

UPDATE STATISTICS has the following security rules:

- If you create an index or table, you have privileges to update its statistics. You cannot update statistics for an object you do not own unless you have DBA or SYSADM privileges.
- Only a user with SYSADM or DBA authority can update statistics for a database.
- A user with DBA or SYSADM authority can update statistics on any table, index, or database.

Clauses

INDEX index name

Updates the statistics for the specified index.

TABLE table name

Updates both the table statistics for the specified table, and also index statistics for all indexes in that table.

DATABASE

Updates the statistics for all indexes and tables in the database. If you use this option, you cannot enter test statistics with the SET option.

SET

To enter user-modified statistics, use this clause in conjunction with either one of the following system catalog columns or the DISTINCTCOUNT *index key* clause. SQLBase then updates the specified value in the system catalog, as if the values were real data.

system catalog column name=expression

Set a test value here for the system catalog column describing the index or table specified in the ON clause. The value must evaluate to a constant. You enter a column from either the SYSADM.SYSTABLES or SYSADM.SYSINDEXES table,

depending on whether you are entering statistics for a table or index. SQLBase updates the specified system catalog table with this value.

If you are updating statistics for a table, set a value for one of the following SYSADM.SYSTABLES columns:

- ROWCOUNT
- PAGECOUNT
- ROWPAGECOUNT
- LONGPAGECOUNT

If you are updating statistics for an index, set a value for one of the following SYSADM.SYSINDEXES columns:

- HEIGHT
- LEAFCOUNT
- CLUSTERCOUNT
- PRIMPAGECOUNT
- OVFLPAGECOUNT
- INDEXPAGECOUNT

Note: The PRIMPAGECOUNT and OVFLPAGECOUNT columns are applicable only to clustered hashed indexes.

Read Appendix A in the *Database Administrator's Guide* for complete descriptions of these columns.

DISTINCTCOUNT (index key)

Use this clause to enter test values for the number of distinct index key values. The value must be a constant.

The *index key* parameter is a valid prefix key of the index you specified with the INDEX *index name* clause. The syntax for *index key* is:

```
column-name [ , column-name ] ...
```

See the following section for an example.

Example

The following example updates statistics on the CUSTOMER_ID index:

```
UPDATE STATISTICS ON INDEX CUSTOMER_ID;
```

The following example sets values for the ROWCOUNT, ROWPAGECOUNT, and LONGPAGECOUNT columns in SYSADM.SYSTABLES for the EMPLOYEE table. Other statistics are unaffected.

```
UPDATE STATISTICS ON TABLE employee
    SET rowcount = 5000, rowpageount=200,
    longpagecount = 0;
```

The following command sets some of the index statistics for the index *emp_name_indx* on the table EMPLOYEE. This is a BTree index.

```
UPDATE STATISTICS ON INDEX emp_name_indx
    SET height = 2, leafcount=100, clustercount = 200;
```

The following example updates statistics for an index key. Assuming that the index EMP_NAME_IDX is a two column key, with 5000 distinct values for the key (LNAME, FNAME) and 4000 distinct values for (LNAME*)* only, the following alternatives can be used to define the distinct value.

```
UPDATE STATISTICS ON INDEX EMP_NAME_IDX SET
    DISTINCTCOUNT (LNAME, FNAME)  =  5000;
```

```
UPDATE STATISTICS ON INDEX EMP_NAME_IDX SET
    DISTINCTCOUNT(LNAME) =  4000;
```

or

```
UPDATE STATISTICS ON INDEX EMP_NAME_IDX SET
    DISTINCTCOUNT(LNAME, FNAME)  =  5000,
    DISTINCTCOUNT(LNAME) =  4000;
```

See also

CREATE INDEX
SET TIME (SQLTalk command)

Chapter 4

SQL Function Reference

SQLBase has a set of functions for manipulating strings, dates and numbers. Each function is described in this chapter.

A function returns a value that is derived by applying the function to its arguments.

Functions are classified as:

- Aggregate functions
- String functions
- Date and time functions
- Logical functions
- Special functions
- Math functions
- Finance functions

SQLBase provides both DB2-compatible and other functions. Functions which are extensions of DB2 and are *not* compatible with DB2 are prefixed with an "at sign" (@).

Data type conversions in functions

In most cases, functions accept any data type as an argument if the value conforms to the operation that function performs. SQLBase will automatically convert the value to the required data type.

For example, in functions that perform arithmetic operations, arguments can be character data types if the value forms a valid numeric value (only digits and standard numeric editing characters).

For date/time functions, an argument can be a character or numeric data type if the value forms a valid date/time value.

Aggregate functions

An aggregate function computes one summary value from a group of values.

Aggregate functions can be applied to the data values of an entire table or to a subset of the rows in a table.

They may be nested up to two levels deep.

The data type of the argument may be numeric, date/time, or character. If an argument is a character data type, the value must form a valid numeric or date/time value (only digits and standard editing characters). SQLBase automatically converts the value to the required data type.

You cannot use aggregate functions while in restriction mode.

SQLBase supports the following aggregate functions:

> AVG
> COUNT
> MAX
> @MEDIAN
> MIN
> SUM
> @SDV

String functions

String functions return information about character data types.

The output of a string function is always a string or a number. Some functions yield TRUE or FALSE. TRUE is expressed as the number 1 and FALSE is expressed as 0.

You can nest string functions within one another, so that the output of the inner function is used as an argument to the outer function.

SQLBase supports the following string functions:

> @CHAR
> @CODE
> @DECODE
> @EXACT
> @FIND
> @LEFT
> @LENGTH
> @LOWER
> @MID
> @NULLVALUE
> @PROPER
> @REPEAT
> @REPLACE
> @RIGHT
> @SCAN
> @STRING
> @SUBSTRING
> @TRIM
> @UPPER
> @VALUE

String functions cannot be used in an ORDER BY clause of the SELECT command. Instead, specify the string function in the select list and then use the select list column number in the ORDER BY clause.

Date/Time functions

These functions return information about date/time data values or return a date/time result. SQLBase supports the following date/time functions:

> @DATE
> @DATETOCHAR
> @DATEVALUE
> @DAY
> @HOUR
> @MICROSECOND
> @MINUTE
> @MONTH
> @MONTHBEG
> @NOW
> @QUARTER
> @QUARTERBEG

@SECOND
@TIME
@TIMEVALUE
@WEEKBEG
@WEEKDAY
@YEAR
@YEARBEG
@YEARNO

For date/time functions, an argument can be a character or numeric data type if the value forms a valid date/time value.

When a portion of the input date/time string is missing, SQLBase supplies the default of 0, which converts to December 30, 1899 12:00:00 AM. Functions behaving this way are @DATE, @DATEVALUE, @NOW, @TIME and @TIMEVALUE.

Math functions

These functions take single numeric values as arguments and return numeric results.

The mathematical functions are similar to Microsoft C Library math functions. Trigonometric functions are based on radians instead of degrees.

Arguments can be character data types if the value forms a valid numeric value (only digits and standard numeric editing characters). SQLBase will automatically convert the value to the required data type.

SQLBase supports the following math functions:

@ABS
@ACOS
@ASIN
@ATAN
@ATAN2
@COS
@EXP
@FACTORIAL
@INT
@LN
@LOG
@MOD
@PI
@ROUND
@SIN
@SQRT
@TAN

Finance functions

The finance functions are similar to Microsoft C Library math functions.

Arguments can be character data types if the value forms a valid numeric value (only digits and standard numeric editing characters). SQLBase automatically converts the value to the required data type.

SQLBase supports the following finance functions:

> @CTERM
> @FV
> @PMT
> @PV
> @RATE
> @SLN
> @SYD
> @TERM

Logical functions

Logical functions return a value based on a condition. The result of these functions is always 1 or 0 (TRUE = 1, FALSE = 0).

SQLBase supports the following logical functions:

> @IF
> @ISNA

Special functions

These functions provide special capabilities.

> @CHOOSE
> @DECIMAL
> @DECODE
> @HEX
> @LICS

SQLBase function summary

Function Name	Description
AVG	Average of items.
COUNT	Count of items.
MAX	Maximum of items.
MIN	Minimum of items.
SUM	Sum of items.
@ABS	Absolute value.
@ACOS	Arc-cosine.
@ASIN	Arc-sine.
@ATAN	Two-quadrant arc-tangent.
@ATAN2	Four-quadrant arc-tangent.
@CHAR	ASCII character for a decimal code.
@CHOOSE	Select a value from a list based on a correlation.
@CODE	ASCII decimal code of the first character in a string.
@COS	Cosine.
@CTERM	Compounding periods to earn a future value.
@DATE	Convert to a date.
@DATETOCHAR	Edit a date value.
@DATEVALUE	Edit a date value.
@DAY	Day of the month.
@DECIMAL	Decimal value of a hexadecimal string.
@DECODE	Returns a string, given an expression.
@EXACT	Compare two strings.
@EXP	Natural logarithmic base (e) raised to the x power.

Function Name	Description
@FACTORIAL	Factorial.
@FIND	Position within string1 that occurs in string2.
@FV	Future value of a series of equal payments.
@HEX	Hexadecimal string of a decimal number.
@HOUR	Hour of the day.
@IF	Test number and return 1 if TRUE or 2 if FALSE.
@INT	Integer portion.
@ISNA	Return TRUE if NULL.
@LEFT	Left-most substring.
@LENGTH	Length of a string.
@LICS	Sort using international character set.
@LN	Natural logarithm (base e) of (positive) x.
@LOG	Positive base-10 logarithm of x.
@LOWER	Upper-case to lower-case.
@MEDIAN	Middle value in a set of items.
@MICROSECOND	Microsecond value.
@MID	Return a string, starting with the character at start-position.
@MINUTE	Minute of the hour.
@MOD	Modulo (remainder) of x/y.
@MONTH	Month of the year.
@MONTHBEG	First day of the month.
@NOW	Current date and time.
@NULLVALUE	Return a string or number specified by y if x is NULL.
@PI	Value Pi ($\pi = 3.14159265$).
@PMT	Periodic payments needed to pay off loan principal.

Function Name	Description
@PROPER	Convert first character of each word in a string to uppercase and make other characters lowercase.
@PV	Present value of a series of equal payments.
@QUARTER	Number that represents the quarter.
@QUARTERBEG	First day of the quarter.
@RATE	Interest rate for an investment to grow to a future value.
@REPEAT	Concatenates a string with itself for the specified number of times.
@REPLACE	Replace characters in a string.
@RIGHT	Rightmost substring.
@ROUND	Round a number.
@SCAN	Search a string for a pattern.
@SDV	Standard deviation.
@SECOND	Second of the minute.
@SIN	Sine.
@SLN	Straight-line depreciation.
@SQRT	Square root.
@STRING	Convert a number to a string.
@SUBSTRING	Return a portion of a string.
@SYD	Sum-of-the-Years'-Digits depreciation.
@TAN	Tangent.
@TERM	Number of payment periods for an investment.
@TIME	Return a date/time value given the hour, minute, and second.
@TIMEVALUE	Return a date/time value, given HH:MM:SS [AM or PM].
@TRIM	Strip leading and trailing blanks; compress multiple spaces.

Function Name	Description
@UPPER	Lower-case to upper-case.
@VALUE	Convert a character string with digits to a number.
@WEEKBEG	Monday of the week.
@WEEKDAY	Day of the week.
@YEAR	The year relative to 1900.
@YEARBEG	First day of the year.
@YEARNO	Calendar year.

AVG

```
 ►► AVG ─── ( ┌─ ALL ──────────┐  expression ─ ) ─── ◄◄
             └─ DISTINCT ──────┘
```

This function returns the average of the values in the argument.

The data type of the argument may be numeric, date/time, or character. If an argument is a character data type, the value must form a valid numeric or date/time value (only digits and standard editing characters). SQLBase will automatically convert the value to the required data type.

The data type of the result is numeric.

The keyword DISTINCT eliminates duplicates. If DISTINCT is not specified, duplicates are not eliminated.

Null values are ignored.

Example

```
SELECT AVG (SALARY) FROM EMPSAL;
```
This example finds the total salary for each department, the average salary, the number of people in each department.

```
SELECT DEPTNO, SUM(SALARY), AVG(SALARY), COUNT(SALARY)
    FROM EMP.EMPSAL WHERE EMP.EMPNO = EMPSAL.EMPNO GROUP BY
    DEPTNO;
```

COUNT
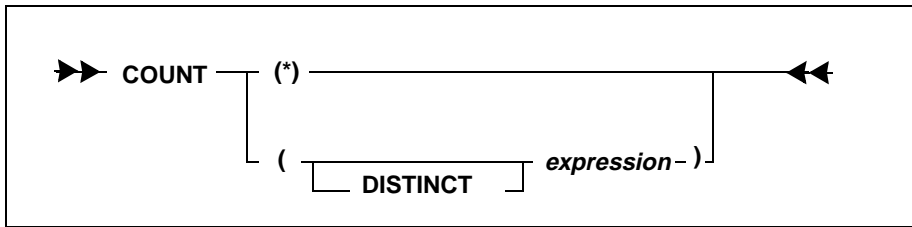
This function returns a count of items.

COUNT(*) always returns the number of rows in the table. Rows that contain null values are included in the count.

COUNT(column-name) returns the number of column values.

COUNT(DISTINCT column-name) filters out duplicate column values.

LONG VARCHARs can be counted.

The keyword DISTINCT eliminates duplicates. If DISTINCT is not specified, duplicates are not eliminated.

Example

How many rows are in each department of the EMP table?

```
SELECT DEPTNO, COUNT(*) FROM EMP
   GROUP BY DEPTNO;
```

MAX
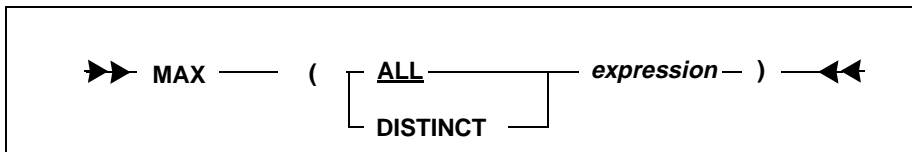
This function returns the maximum value in the argument, which is a set of column values.

The data type of the argument may be numeric, date/time, or character. If an argument is a character data type, the value must form a valid numeric or date/time

value (only digits and standard editing characters). SQLBase will automatically convert the value to the required data type.

The data type of the result is the same as the input argument.

The keyword DISTINCT eliminates duplicates. If DISTINCT is not specified, then duplicates are not eliminated.

Null values are ignored.

Example

This example finds the highest and the lowest salary.

```
SELECT MAX(SALARY), MIN(SALARY) FROM EMPSAL;
```

MIN

```
 ▶▶ MIN ───── (    ┌─ ALL ──────────┐  expression ─ ) ──◀◀
                   └─ DISTINCT ──────┘
```

This function returns the minimum value in the argument, which is a set of column values.

The data type of the argument may be numeric, date/time, or character. If an argument is a character data type, the value must form a valid numeric or date/time value (only digits and standard editing characters). SQLBase will automatically convert the value to the required data type.

The data type of the result is the same as the input argument.

The keyword DISTINCT eliminates duplicates. If DISTINCT is not specified, then duplicates are not eliminated.

Null values are ignored.

Example

This example finds the highest and the lowest salary.

```
SELECT MAX(SALARY), MIN(SALARY) FROM EMPSAL;
```

SUM

```
>>── SUM ── ( ─┬── ALL ──┬─ expression ── ) ──────<<
              └─ DISTINCT ─┘
```

This function returns the sum of the values in the argument, which is a set of column values.

The data type of the argument may be numeric, date/time, or character. If an argument is a character data type, the value must form a valid numeric or date/time value (only digits and standard editing characters). SQLBase will automatically convert the value to the required data type.

The data type of the result is the same as the input argument.

The keyword DISTINCT eliminates duplicates. If DISTINCT is not specified, then duplicates are not eliminated.

Null values are ignored.

Example

Calculate the total salary.

```
SELECT SUM (SALARY) FROM EMPSAL;
```

This example totals the salary by department.

```
SELECT DEPTNO, SUM (SALARY), AVG (SALARY), COUNT
    (SALARY) FROM EMP, EMPSAL
    WHERE EMP.EMPNO = EMPSAL.EMPNO GROUP BY DEPTNO;
```

@ABS

@ABS(x)

This function returns the absolute value of x.

Example

The following expression returns 1.1:

```
@ABS(-1.1)
```

The following expression returns all column entries as positive (absolute) values:

```
SELECT @ABS(IVAL) FROM GEOM;
```

@ACOS

@ACOS (X)

This function returns the arc-cosine of *x* in radians. *x* must be in the range [-1, 1].

Example

The following expression returns 1.47062891:

```
@ACOS(.1)
```

The following expression returns the arc-cosine of all PVAL column entries in the
GEOM table:

```
SELECT @ACOS(PVAL) FROM GEOM;
```

@ASIN

@ASIN(*x*)

This function returns the arc-sine of *x* in radians. *x* must be in the range [-1, 1].

Example

The following expression returns .100167421:

```
@ASIN(.1)
```

The following expression returns the arc-sine of all PVAL column entries in the
GEOM table:

```
SELECT @ASIN(PVAL) FROM GEOM;
```

@ATAN

@ATAN (*x*)

This function returns the arc-tangent of *x* in radians.

Example

The following expression returns .099668652:

```
@ATAN(.1)
```

The following expression returns the arc-tangent of all PVAL column entries in the GEOM table:

```
SELECT @ATAN(PVAL) FROM GEOM;
```

@ATAN2

@ATAN2(*x, y*)

This function returns the arc-tangent of *y*/*x*.

The order of arguments is the opposite of C [atan2(y,x)]. The value of X may not be zero.

Example

The following expression returns .785398163:

```
@ATAN2(.1,.1)
```

The following expression returns the arc-tangent of all QVAL and PVAL column entries in the GEOM table:

```
SELECT @ATAN2(PVAL,QVAL) FROM GEOM;
```

@CHAR

@CHAR (*number*)

This function returns the ASCII character for a decimal code. If the argument is outside the ASCII character set range, results depend on the display character set.

Example

The following expression returns the letter 'A':

```
@CHAR(65)
```

@CHOOSE

@CHOOSE (*selector number, value 0, value 1, ..., value n*)

This function selects a value from a list based on a correlation between the selector-number and the sequence number of a value in the list.

You must specify a selector-number and at least one value. A negative selector-number maps to the first value in the list (value 0). If the selector number exceeds the number of values in the list, the result is the last value in the list. Every value in the list is cast to the data type of the first value (value 0).

Example

```
@CHOOSE(SEL_NUM, 'A', 'B', 'C', 'D', 'E', 'F', 'G')
```

Selector number	Values
0	A
-1	A
2	C
12	G

This example finds the day of the week on which each employee was hired.

```
SELECT @CHOOSE(@WEEKDAY(HIREDATE), 'Sat','Sun', 'Mon',
    'Tue', 'Wed', 'Thu', 'Fri'), @YEAR(HIREDATE)
    FROM EMP WHERE @YEARNO(HIREDATE) > 1990;
```

@CODE

@**CODE(***string***)**

This function returns the ASCII decimal code of the first character in a string.

Example

The following expression returns the number 65, which is the code for 'A':

```
@CODE('ABC')
```

@COS

@**COS(***x***)**

This function returns the cosine of x, where *x* is in radians.

Example

The following expression returns .995004165:

```
@COS(.1)
```

The following SQL statement returns the cosine of all PVAL column entries in the GEOM table:

```
SELECT @COS(PVAL) FROM GEOM;
```

@CTERM

@**CTERM(***int, fv, pv***)**

This function returns the number of compounding periods to an investment of present value *pv* to grow to a future value *fv*, earning a fixed periodic interest rate *int*.

@CTERM uses this formula to compute the term:

ln (fv/pv)	fv = future value
ln (1+int)	pv = present value
	int = periodic interest rate
	ln = natural logarithm

Example

The following expression:

```
@CTERM(.10/12,20000,10000)
```

returns 83.5237559, which is the number of months it will take to double a $10,000 investment that earns a 10% annual interest rate compounded monthly.

@DATE

@DATE(*year number, month number, day number*)

This function converts the arguments to a date.

The data type of the result is date.

Example

The following expression returns 31-JAN-1996:

```
@DATE(1996,1,31)
```

@DATETOCHAR

@DATETOCHAR(*date, picture*)

This function accepts a DATE, TIME, or DATETIME data type value (specified in *date*), applies the editing specified by *picture* and returns the edited value. For an explanation of *picture*, see the SQLTalk COLUMN command.

The data type of the result is character.

Example

This SQL statement returns a string in the form 05-07-96:

```
SELECT @DATETOCHAR(SYSDATETIME, 'dd-mm-yy') FROM ...
```

@DATEVALUE

@DATEVALUE(*date string*)

This function converts the argument to a date.

@DATEVALUE is like @DATE, except its argument is a date string, or a portion of a date string. It converts the date string in any standard date string form (dd-mon-yyyy hh:mm:ss) to the date portion of the string.

The data type of the result is date.

Example

If a DATE column called APPT contains '18-JAN-1996 10:14:27 AM', then the following expression returns 18-JAN-1996:

```
@DATEVALUE(APPT)
```

@DAY

@DAY(*date*)

This function returns a number between 1 and 31 that represents the day of the month.

Example

If BIRTHDATE contains '12/28/46', then the following expression returns 28:

```
@DAY(BIRTHDATE)
```

@DECIMAL

@DECIMAL(*string*)

This function returns the decimal equivalent for the given hexadecimal number.

Example

The following expression returns 10:

```
@DECIMAL('A')
```

@DECODE

@DECODE(*expr, search1, return1, search2, return2, ..., [default]*)

If *expr* equals any *search*, this function returns the search's corresponding *return*; if not, it returns *default*. If default is omitted and there is no match, NULL is returned. The *expr* may be any data type; *search* must be the same type. The value returned is forced to the same datatype as the first *return*.

Example

This returns employees' names and their department name and number. If no match is found, "Other" is returned:

```
SELECT LNAME, @DECODE (DEPTNO, 2500, 'R&D', 2600,
    'SALES', 'OTHER'), DEPTNO FROM EMP;

LNAME      @DECODE(DEPTNO...)DEPTNO
Carver     R&D                2500
Murphy     OTHER         2400
Johnson    R&D                2500
Drape      SALES              2600
Foghorn    R&D                2500
```

@EXACT

@EXACT(*string1, string2*)

This function compares two strings or numbers.

If the strings are identical, the function returns 1; otherwise the function returns 0.

This function is case sensitive.

Example

The following expression returns 0:

```
@EXACT('TRUDY', 'NOAH')
```

If the NAME column contains the value 'TRUDY', then the following expression returns 1:

```
@EXACT('TRUDY', NAME)
```

The following expressions return 1:

```
@EXACT(2.3, 2.3)
@EXACT(3+4, 7)
```

@EXP

@EXP(*x*)

This function returns the natural logarithmic base (e) raised to the x power.

Example

The following expression returns 22026.4658:

```
@EXP(10)
```

The following SQL statement returns the natural logarithmic base of all PVAL column entries in the GEOM table:

```
SELECT @EXP(PVAL) FROM GEOM;
```

The following example raises 2 to the 10th power (2^10):

```
@EXP (10 * @LN(2))
```

@FACTORIAL

@**FACTORIAL(*x*)**

This function computes the factorial of the argument. The argument must be an INTEGER (no decimal portion) and non-negative (>= 0). The upper limit is 69.

Example

The following expression returns 3628800:

```
@FACTORIAL(10)
```

The following SQL statement returns the factorial of all WVAL column entries in the GEOM table:

```
SELECT @FACTORIAL(WVAL) FROM GEOM;
```

@FIND

@**FIND(*string1, string2, start position*)**

This function returns the position (offset) within *string1* that occurs in *string2*. The search begins with the character at *start-pos* in *string2*. If the pattern is not found, the function returns -1.

The starting position represents an offset within a string argument. The first character in a string is at position 0. For example, in the string 'RELATION', the character 'R' is at position 0, the final character 'N' is at position 7, and the string is 8 characters long. In other words, the last position in *string1* is calculated by subtracting one from the length of *string1*.

Example

The following expression returns 5:

```
@FIND( 'TRIPLETT', 'NOAH TRIPLETT', 0)
```

@FV

@FV(*pmt, int, n*)

This function returns the future value of a series of equal payments (*pmt*) earning periodic interest rate (*int*) over the number of periods (*n*).

@FV uses this formula to compute the future value of an ordinary annuity:

$$pmt * \frac{(1 + int)^n - 1}{int}$$
       pmt = periodic payment
       int = periodic interest rate
       n = number of periods

Ordinary Annuity Example

The expression:

```
@FV(2000,.10,20)
```

returns $114,549.999, which is the value of an account after 20 years of depositing $2,000 at the end of each year, at an annually compounded interest rate of 10%. Interest payments and deposits are transacted on the *last day of each year.*

Annuity Due Example

The following expression:

```
@FV(2000,.10,20) * (1+.10)
```

returns $126,004.999, which is the value of an annuity amount due annually. Note that this is 10% over the ordinary annuity calculated in the above example.

@HEX

@HEX(*number*)

This function returns the hexadecimal equivalent for the given decimal number.

Example

The following expression returns 'A':

```
@HEX(10)
```

@HOUR

@HOUR(*date*)

This function returns a number between 0 and 23 that represents the hour of the day.

Example

The following expression returns 15:

```
@HOUR(12/28/46 03:52:00 PM)
```

@IF

@IF(*number, value1, value2*)

This function tests *number* and returns *value1* if it is TRUE (non-zero) or *value2* if it is FALSE (zero).

A non-zero argument evaluates to TRUE, and an argument of zero evaluates to FALSE. A null value evaluates to FALSE. Each value in the list is cast to the data type of the first value.

Example

The following expression returns 'M' if TEST1 is non-zero, and 'F' if TEST1 is 0:

```
@IF(TEST1,'M','F')
```

@INT

@INT(*x*)

This function returns the integer portion of *x*. If *x* is negative, the decimal portion is truncated.

Example

The following expression returns 10:

```
@INT(10.2)
```

The following expression returns -3:

```
@INT(-3.7)
```

The following SQL statement returns the integer portion of all PVAL column entries in the GEOM table:

```
SELECT @INT(PVAL) FROM GEOM;
```

@ISNA

@ISNA(*argument*)

This function returns 1 (TRUE) if the argument is NULL. Any other value returns 0 (FALSE). The argument can be any value, including a column value.

Example

The following expression returns 1:

```
@ISNA (NULL)
```

The following expression returns 0:

```
@ISNA('hello')
```

@LEFT

@LEFT(*string, length*)

This function returns a string for the specified length, starting with the first (leftmost) character in the string.

Example

The following expression returns 'P8':

```
@LEFT('P8-196', 2)
```

The following example shows how to use the @LEFT function in a SELECT statement:

```
SELECT * FROMSYSCOLUMNS
    WHERE @LEFT (TBNAME, 3) ! = 'SYS';
```

@LENGTH

@LENGTH(*string*)

This function returns the length of a string. The length is the number of characters in the string.

You cannot use this function to find the length of a LONG VARCHAR column.

Example

If the value in the column EMPNAME is 'JOYCE', the following expression returns the number 5:

```
@LENGTH(EMPNAME)
```

The following example finds the entries in the EMP table where the length of the LNAME column exceeds 10 characters.

```
SELECT EMPNO, @SUBSTRING (LNAME, 0, 10)
    FROM EMP WHERE @LENGTH (LNAME) > 10
```

@LICS

@**LICS(***string***)**

This function uses an international character set for sorting its argument, instead of the ASCII character set. This is useful for sorting characters not in the English language. The translation table for this character set is shown below.

Example

The following expression returns 'NXTRI\' @LICS ('Murphy')

```
SELECT @LICS(LNAME) FROM EMP ORDER BY 1;
```

Code	Character	Description
0	0	Ctrl @
1	1	Ctrl A
2	2	Ctrl B
3	3	Ctrl C
4	4	Ctrl D
5	5	Ctrl E
6	6	Ctrl F
7	7	Ctrl G
8	8	Ctrl H
9	9	Ctrl I
10	10	Ctrl J line feed
11	11	Ctrl K
12	12	Ctrl L form feed
13	13	Ctrl M return
14	14	Ctrl N
15	15	Ctrl O

Code	Character	Description
16	16	Ctrl P
17	17	Ctrl Q
18	18	Ctrl R
19	19	Ctrl S
20	20	Ctrl T
21	21	Ctrl U
22	22	Ctrl V
23	23	Ctrl W
24	24	Ctrl X
25	25	Ctrl Y
26	26	Ctrl Z
27	27	[Esc]
28	28	FS
29	29	GS
30	30	RS
31	31	US
32	32	Space
33	33	!
34	34	"
35	35	#
36	36	$
37	37	%
38	38	&
39	39	Apostrophe
40	40	(
41	41	)

Code	Character	Description
42	42	*
43	43	+
44	44	
45	45	-
46	46	.
47	47	/
48	48	0
49	49	1
50	50	2
51	51	3
52	52	4
53	53	5
54	54	6
55	55	7
56	56	8
57	57	9
58	58	:
59	59	;
60	60	<
61	61	=
62	62	>
63	63	?
64	64	@
65	65	A
66	66	B
67	67	C

Code	Character	Description
68	68	D
70	69	E
71	70	F
72	71	G
73	72	H
74	73	I
75	74	J
76	75	K
77	76	L
78	7 7	M
79	78	N
81	79	O
82	80	P
83	81	Q
84	82	R
85	83	S
87	84	T
88	85	U
89	86	V
90	87	W
91	88	X
92	89	Y
93	90	Z
99	91	[
100	92	\
101	93	]

Code	Character	Description
102	94	^
103	95	_
104	96	`
65	97	a
66	98	b
67	99	c
68	100	d
70	101	e
71	102	f
72	103	g
73	104	h
74	105	i
75	106	j
76	107	k
77	108	l
78	109	m
79	110	n
81	111	o
82	112	p
83	113	q
84	114	r
85	115	s
87	116	t
88	117	u
89	118	v
90	119	w

Code	Character	Description
91	120	x
92	121	y
93	122	z
105	123	{
106	124	|
107	125	}
108	126	~ (tilde)
109	127	DEL
110	128	Uppercase grave
111	129	Uppercase acute
112	130	Uppercase circumflex
113	131	Uppercase umlaut
114	132	Uppercase tilde
115	133	
116	134	
117	135	
118	136	
119	137	
120	138	
121	139	
122	140	
123	141	
124	142	
125	143	
126	144	Lowercase grave
127	145	Lowercase acute

Code	Character	Description
128	146	Lowercase circumflex
129	147	Lowercase umlaut
130	14 8	Lowercase tilde
131	149	Lowercase i without dot
132	150	Ordinal indicator
133	151	Begin attribute (display)
134	152	End attribute (display only)
135	153	Unknown character (display)
136	154	Hard space (display only)
137	155	Merge character (display)
138	156	
139	157	
140	158	
141	159	
142	160	Dutch Guilder
143	161	Inverted exclamation mark
144	162	Cent sign
145	163	Pound sign
146	164	Low opening double quotes
147	165	Yen sign
148	166	Pesetas sign
149	167	Section sign
150	168	General currency sign
151	169	Copyright sign
152	170	Feminine ordinal
153	171	Angle quotation mark left

Code	Character	Description
154	1 72	Delta
155	173	Pi
156	174	Greater-than-or-equals
157	175	Divide sign
158	176	Degree sign
159	177	Plus/minus sign
160	178	Superscript 2
161	179	Superscript 3
162	180	Low closing double quotes
163	181	Micro sign
164	182	Paragraph sign
165	183	Middle dot
166	184	Trademark sign
167	185	Superscript 1
168	186	Masculine ordinal
16 9	187	Angle quotation mark right
170	188	Fraction one quarter
171	189	Fraction one-half
172	190	Less-than-or -equals
173	191	Inverted question mark
65	192	Uppercase A with grave
65	193	Uppercase A with acute
65	194	Uppercase A with circumflex
65	195	Uppercase A with tilde
65	196	Uppercase A with umlaut
65	197	Uppercase A with ring

Code	Character	Description
97	197	Uppercase A with ring
94	198	Uppercase AE with ligature
67	199	Uppercase C with cedilla
70	200	Uppercase E with grave
70	201	Uppercase E with acute
70	202	Uppercase E with circumflex
70	203	Uppercase E with umlaut
74	204	Uppercase I with grave
74	205	Uppercase I with acute
74	206	Uppercase I with circumflex
74	207	Uppercase I with umlaut
69	208	Uppercase eth (Icelandic)
80	209	Uppercase N with tilde
81	210	Uppercase O with grave
81	211	Uppercase O with acute
81	212	Uppercase O with circumflex
81	213	Uppercase O with tilde
81	214	Uppercase O with umlaut
80	215	Uppercase OE with diphthong
96	216	Uppercase O with slash
88	217	Uppercase U with grave
88	218	Uppercase U with acute
88	219	Uppercase U with circumflex
88	220	Uppercase u with umlaut
92	221	Uppercase Y with umlaut
98	222	Uppercase thorn (Icelandic)

Code	Character	Description
86	223	Lowercase German sharp s
65	224	Lowercase a with grave
65	225	Lowercase a with acute
65	226	Lowercase a with circumflex
65	227	Lowercase a with tilde
65	228	Lowercase a with umlaut
65	229	Lowercase a with ring
95	230	Lowercase ae with ligature
6 7	231	Lowercase c with cedilla
70	232	Lowercase e with grave
70	233	Lowercase e with acute
70	234	Lowercase e with circumflex
70	235	Lowercase e with umlaut
74	236	Lowercase i with grave
74	237	Lowercase i with acute
74	238	Lowercase i with circumflex
74	239	Lowercase i with umlaut
69	240	Lowercase eth (Icelandic)
80	241	Lowercase n with tilde
81	242	Lowercase o with grave
81	243	Lowercase o with acute
81	244	Lowercase o with circumflex
81	245	Lowercase o with tilde
81	246	Lowercase o with umlaut
80	247	Lowercase oe with diphthong
81	248	Lowercase o with slash

Code	Character	Description
88	249	Lowercase u with grave
88	250	Lowercase u with acute
88	251	Lowercase u with circumflex
88	252	L owercase u with umlaut
92	253	Lowercase y with umlaut
174	254	Lowercase thorn (Icelandic)

@LN

@LN(*x*)

This function returns the natural logarithm (base e) of (positive) *x*. The log of a zero or negative argument is handled as an overflow error.

Example

The following expression returns -2.3025851:

```
@LN(.1)
```

The following SQL statement returns the natural logarithm of all PVAL column entries in the GEOM table:

```
SELECT @LN(PVAL) FROM GEOM;
```

@LOG

@LOG(*x*)

This function returns the (positive) base-10 logarithm of *x*. The log of a zero or negative argument is handled as an overflow error.

Example

The following expression returns -1:

```
@LOG(.1)
```

The following SQL statement returns the natural logarithm of all PVAL column entries in the GEOM table:

```
SELECT @LOG(PVAL) FROM GEOM;
```

@LOWER

@LOWER(*string*)

This function converts upper-case alphabetic characters to lower-case. Other characters are not affected.

Example

The following expression returns the string 'joyce':

```
@LOWER('JOYCE')
```

@MEDIAN

This function returns the middle value in a set of values. An equal number of values lie above and below the middle value.

The data type of the argument may be numeric, date/time, or character. If an argument is a character data type, the value must form a valid numeric or date/time value (only digits and standard editing characters). SQLBase automatically converts the value to the required data type. The data type of the result is the same as the input argument.

@MEDIAN finds the middle value with this formula:

```
( n + 1 ) / 2
```

For example, if there are 5 items, then the middle item is the third:

```
( 5 + 1 ) / 2 = 6 / 2 = 3
```

For example, if there are 6 items, then the middle item is between the third and the fourth:

```
( 6 + 1) / 2 = 7 / 2 = 3.5
```

The median is the arithmetic average of the third and fourth values.

The keyword DISTINCT eliminates duplicates. If DISTINCT is not specified, then duplicates are not eliminated. Be cautious when using DISTINCT because the result may loose its statistical meaning.

Null values are ignored.

Example

This example finds the middle salary for department 2500.

```
SELECT @MEDIAN(SALARY) FROM EMP, EMPSAL
   WHERE EMP.EMPNO = EMPSAL.EMPNO AND DEPTNO=2500;
```

@MICROSECOND

@MICROSECOND(*date*)

This function returns the microsecond value in a DATETIME or TIME value. If a microsecond quantity was not specified on input, zero is returned.

Example

The following expression returns 500000:

```
@MICROSECOND(12:44:01:500000)
```

@MID

@MID(*string, start-pos, length*)

This function returns a string of specified length from a string, starting with the character at *start-pos*. This function is similar to @SUBSTRING, except that it requires the third argument.

Example

The following expression returns the second character from a string, a '9':

```
@MID('P9-186', 1, 1)
```

@MINUTE

@MINUTE(*date*)

This function returns a number between 0 and 59 that represents the minute of the hour.

Example

The following expression returns 52:

```
@MINUTE(12/28/46 03:52:00 PM)
```

@MOD

@MOD(*x, y*)

This function returns the modulo (remainder) of *x*/*y*. Division by zero is an overflow error.

Example

The following expression returns 5:

```
@MOD(5,10)
```

The following SQL statement returns the remainder of all PVAL/WVAL column entries in the GEOM table:

```
SELECT @MOD(PVAL,WVAL) FROM GEOM;
```

@MONTH

@MONTH(*date*)

This function returns a number between 1 and 12 that represents the month of the year.

Example

The following expression returns 10 which represents October:

```
@MONTH(25-OCT-96)
```

@MONTHBEG

@MONTHBEG(*date*)

This function returns the first day of the month represented by the date.

Example

If the value in BIRTHDATE is '16-FEB-1947', then the following expression returns 01-FEB-1947:

```
@MONTHBEG(BIRTHDATE)
```

@NOW

@NOW

This function returns the current date and time. It returns the same value as the system keyword SYSDATETIME.

For example, if the date and time is January 12, 1996, 3:15 PM, this function would return 12-JAN-1996 03:15:00 PM.

@NULLVALUE

@NULLVALUE(*x, y*)

This function returns one of the following values specified by *y* if *x* is null:

- string
- number

- date (if the date is a constant. If you try to specify a date by a bind variable such as *1:*, the bind variable is read literally, since it is treated as a CHAR value.)

The data type of the returned value is the same as the data type of the *x* argument.

SQLBase converts the second parameter (*y* argument) to the first parameter's data type (*x* argument). An error results if SQLBase cannot convert this correctly.

Example

The following example returns "N/A" when the column is null:

```
@NULLVALUE(FNAME, 'N/A')
```

The following SQL statement:

```
SELECT @NULLVALUE(DEPTNO,'NOT ASSIGNED') FROM EMP;
```

returns the string 'NOT ASSIGNED' if the DEPTNO column value is null, and DEPTNO is a character column. If the column is numeric, the replacement value must be a number. For example, the following SQL statement:

```
SELECT @NULLVALUE(DEPTNO,9999) FROM EMP;
```

returns 9999 if a null exists in the DEPTNO column. DEPTNO is a numeric data type.

@PI

@PI

This function returns the value Pi (3.14159265). This function has no arguments but could be used as a numeric constant in a nested set of math functions.

Example

The following expression returns 31.4159265:

```
10 * @PI
```

The following SQL statement returns all PVAL column entries multiplied by the value Pi in the GEOM table:

```
SELECT (PVAL) * @PI FROM GEOM;
```

@PMT

@PMT(*principal, interest, periods*)

This function returns the amount of each periodic payment needed to pay off a loan principal (*prin*) at a periodic interest rate (*int*) over a number of periods (*n*).

@PMT uses this formula:

print * int	prin = principal
$(1 - (1 + int)^{-n})$	int = periodic interest rate
	n = number of periods; term

Example

The following expression:

```
@PMT(50000,.125/12,30 * 12)
```

returns $533.628881 which is the value of a monthly mortgage payment for a $50,000, 30-year mortgage at an annual interest rate of 12.5%.

@PROPER

@PROPER(*string*)

This function converts the first character of each word in a string to uppercase and other characters to lower case.

The argument must be a CHAR or VARCHAR data type.

Example

The following expressions both return 'Johann Sebastian Bach':

```
@PROPER('JOHANN SEBASTIAN BACH')

@PROPER('johann sebastian bach')
```

@PV

@PV(*pmt, int, n*)

This function returns the present value of a series of equal payments (*pmt*) discounted at periodic interest rate (*int*) over the number of periods (*n*).

This function is useful when trying to decide the best way to receive a payment option, over time or immediately.

@PV uses this formula:

$$pmt * \frac{(1 - (1 + int)^{-n})}{int}$$

pmt = periodic payment

int = periodic interest rate

n = number of periods; term

Ordinary Annuity Example

The following expression:

```
@PV(50000,.12,20)
```

returns $373,472.181 which is what $1,000,000 paid equally ($50,000 at the end of each year) over 20 years at 12% is worth today.

Annuity Due Example

The following expression:

```
@PV(50000,.12,20) * (1+.12)
```

returns $418,288.843, which is what $1,000,000 paid equally ($50,000 at the beginning of each year) over 20 years at 12% is worth today.

@QUARTER

@QUARTER(*date*)

This function returns a number between 1 and 4 that represents the quarter. For example, the first quarter of the year is January through March.

Example

The following expression returns 1, which represents the first quarter:

```
@QUARTER(12-MAR-96)
```

@QUARTERBEG

@QUARTERBEG(*date*)

This function returns the first day of the quarter represented by the date.

Example

The following expression returns 01-JUL-1776:

```
@QUARTERBEG(04-JUL-1776)
```

The following SQL statement displays the first day of the quarter in which each employee was hired:

```
SELECT @QUARTERBEG (HIREDATE) FROM EMP;
```

@RATE

@RATE(*fv, pv, n*)

This function returns the interest rate for an investment of present value (*pv*) to grow to a future value (*fv*) over the number of compounding periods (*n*).

@RATE uses this formula:

$$((fv/pv)^{(1/n)}) - 1$$

fv = future value

pv = present value

n = number of periods; term

Example

The following expression:

```
@RATE(18000,10000,5 * 12)
```

returns .009844587 which is the periodic (monthly) interest rate calculated for a $10,000 investment for 60 months (5 years) with a maturity value of $18,000 (compounded monthly).

@REPEAT

@REPEAT(*string, number*)

This function concatenates a string with itself for the specified number of times. This creates a string of pattern repetitions.

This function returns nulls if specified in a select list. However, it can be used in a WHERE clause and in other contexts.

Example

The following expression returns the value '$$$$$':

```
@REPEAT('$',5)
```

@REPLACE

@REPLACE(*string1, start-pos, length, string2*)

This function returns a string in which characters from *string1* have been replaced with characters from *string2*. The replacement *string2* begins at *start-pos,* the position at which characters of the specified *length* have been removed.

The first position in the string is 0.

Example

The following expression returns the value 'RALPH':

```
@REPLACE('RALF', 3, 1, 'PH')
```

@RIGHT

@RIGHT(*string, length*)

This function returns a specified number of characters starting from the end, or rightmost part, of a string.

Example

The following expression returns '186':

```
@RIGHT('P4-186', 3)
```

@ROUND

@ROUND(*x, n*)

This function rounds the number *x* with *n* decimal places. The rounding can occur to either side of the decimal point.

Example

The following expression returns 31.42:

```
@ROUND(@PI * 10,2)
```

The following expression returns 1200:

```
@ROUND(1234.1234,-2)
```

The following SQL statement returns the value of all PVAL column entries in the GEOM table, rounded to 2 decimal places to the RIGHT of the decimal point.t:

```
SELECT @ROUND(QVAL,2) FROM GEOM;
```

The following SQL statement returns the value of all PVAL column entries in the GEOM table, rounded to 2 decimal places to the LEFT of the decimal point:

```
SELECT @ROUND(QVAL,-2) FROM GEOM;
```

@SCAN

@SCAN(*string, pattern*)

This function searches a given string for a specified pattern and returns a number indicating the numeric position of the first instance of the pattern.

This function returns null if the column being scanned is null.

The first position in the string is position 0. The match is performed without regard to case.

If the result is -1, it indicates no match was found.

The @SCAN function can perform a case-insensitive match on columns of type CHAR, VARCHAR, and LONG VARCHAR.

Example

The following expression returns 1 as the start position of the character '-':

```
@SCAN('P-186', '-')
```

@SDV

```
 ▶▶── @SDV ─ ( ─┬─── ALL ────┬──── expression ── ) ──── ◀◀
                └── DISTINCT ──┘
```

This function computes the standard deviation for the set of values specified by the argument.

The data type of the argument may be numeric, date/time, or character. If an argument is a character data type, the value must form a valid numeric or date/time value (only digits and standard editing characters). SQLBase automatically converts the value to the required data type.

The keyword DISTINCT eliminates duplicates. If DISTINCT is not specified, then duplicates are not eliminated.

Note that this function produces double precision, which is not the same as an integer value.

Example

The following SQL statement returns the standard deviation of the SALARY column in the table EMPSAL.

```
SELECT @SDV(SALARY) FROM EMPSAL;
```

@SECOND

@SECOND(*date*)

This function returns a number between 0 and 59 that represents the second of the minute.

Example

The following expression returns 58:

```
@SECOND(12/28/46 03:52:58)
```

@SIN

@SIN(*x*)

This function returns the sine of *x*, where *x* is in *radians*.

Example

The following expression returns .841470985:

```
@SIN(1)
```

The following SQL statement returns the value of all PVAL column entries in the GEOM table:

```
SELECT @SIN(PVAL) FROM GEOM;
```

@SLN

@SLN(*cost, salvage, life*)

This function returns the straight-line depreciation allowance of an asset for each period, given the base cost, predicted salvage value, and expected life of the asset.

@SLN uses this formula to compute depreciation:

$$\frac{(c - s)}{n}$$

c = cost of the asset

s = salvage value of the asset

n = useful life of the asset

Example

The following expression:

```
@SLN(10000,1200,8)
```

returns $1100, which is the yearly depreciation allowance for a machine purchased for $10,000, with a useful life of 8 years, and a salvage value of $1200 after the 8 years.

@SQRT

@SQRT(*x*)

This function returns the square root of x (which must be zero or positive). The square root of a negative argument is handled as an overflow error.

Example

The following expression returns 3.16227766:

```
@SQRT(10)
```

The following SQL statement returns the square root of all PVAL column entries in the GEOM table:

```
SELECT @SQRT(PVAL) FROM GEOM;
```

@STRING

@STRING(*number, scale*)

This function converts a number into a string with the number of decimal places specified by *scale*. Numbers are rounded where appropriate.

Example

The following expression returns the character string '123.46':

```
@STRING(123.456, 2)
```

@SUBSTRING

@SUBSTRING(*string, start-pos, length*)

This function returns a desired portion of a string from a given argument string. The substring starts at the specified start position and is of the specified length. If the start position and length define a substring that exceeds the actual length of the string, the result is truncated to the actual length of the string. If the start position is beyond length of the string, a null string ('') is returned. The first character in a string is at start-pos 0.

The length parameter is optional.

Example

The following expression returns 'SMITH':

```
@SUBSTRING('DR. SMITH', 4, 20)
```

The following example returns the first 10 characters of the LNAME column in the EMP table where the length of the LNAME column exceeds 10 characters.

```
SELECT EMPNO, @SUBSTRING(LNAME, 0, 10) FROM EMP WHERE
  @LENGTH(LNAME) > 10;
```

The function is nearly the same as @MID$ except that if the third argument is left off, the function returns a string beginning with the start position.

The following expression returns 'R. SMITH':

```
@SUBSTRING ('DR. SMITH', 1)
```

@SYD

@SYD(*cost, salvage, life, period*)

This function returns the Sum-of-the-Years'-Digits depreciation allowance of an asset for a given period, given the base cost, predicted salvage value, expected life of the asset and specific period.

@SYD uses this formula to compute depreciation:

$$(c - s) * (n - p + 1)$$ c = cost of the asset

$$(n * (n + 1)/2)$$ s = salvage value of the asset

p = period for which depreciation is being computed

n = useful life of the asset

Example

The following expression:

```
@SYD(10000,1200,8,5)
```

returns $977.777778, which is the depreciation allowance for the fifth year for a $10,000 machine with a useful life of 8 years, and a salvage value of $1200 after the 8 years.

@TAN

@TAN(*x*)

This function returns the tangent of *x*, where *x* is in radians.

Example

The following expression returns .648360827:

```
@TAN(10)
```

The following SQL statement returns the tangent of all PVAL column entries in the GEOM table:

```
SELECT @TAN(PVAL) FROM GEOM;
```

@TERM

@TERM(*pmt, int, fv*)

This function returns the number of payment periods for an investment, given the amount of each payment *pmt*, the periodic interest rate *int*, and the future value *fv* of the investment.

@TERM uses this formula to compute the term:

$$\frac{\ln (1 + (fv * int/pmt))}{\ln (1 + int)}$$

pmt = periodic payment

fv = future value

int = periodic interest rate

ln = natural logarithm

Example

The following expression:

```
@TERM(2000,.10,100000)
```

returns 18.7992455, which is the number of years it will take for an investment to mature to an amount of $100,000. This is based on a yearly deposit of $2,000 at the end of each year to an account that earns 10% compounded annually.

@TIME

@TIME(*hour, minute, second*)

This function returns a time value given the *hour, minute,* and *second.* An hour is a number from 0 to 23; a minute is a number from 0 to 59; a second is a number from 0 to 59.

Example

The following expression returns 13:00:00:

```
@TIME(13,0,0)
```

@TIMEVALUE

@TIMEVALUE(*time*)

The function returns a time value, given a string in the form HH:MM:SS [AM or PM]. If the AM or PM parameter is omitted, military time is used.

Example

If the CHAR column APPT contains '18-JAN-1994 10:14:27 AM', then the following expression returns 10:14:27:

```
@TIMEVALUE(APPT)
```

@TRIM

@TRIM(*string*)

This function strips leading and trailing blanks from a string and compresses multiple spaces within the string into single spaces.

Example

The following expression returns 'JOHN DEWEY':

```
@TRIM('   JOHN    DEWEY ')
```

@UPPER

@UPPER(*string*)

This function converts lower-case letters in a string to upper-case. Other characters are not affected.

Example

The following expression returns 'E.E. CUMMINGS':

```
@UPPER('e.e. cummings')
```

@VALUE

@VALUE(*string*)

This function converts a character string that has the digits (0-9) and an optional decimal point or negative sign into the number represented by that string.

Example

The following expression returns the number 123456 which will be interpreted strictly as a numeric data type by any function to which it is passed:

```
@VALUE('123456')
```

@WEEKBEG

@WEEKBEG(*date*)

This function returns the date of the Monday of the week containing the date. This is the previous Monday if the date is not a Monday, and the date value itself if it is a Monday.

Example

If the value in DATECOL is 01/FEB/94, then the following expression returns 31-JAN-1994:

```
@WEEKBEG(DATECOL)
```

@WEEKDAY

@WEEKDAY(*date*)

This function returns a number between 0 and 6 (Saturday = 0 and Friday = 6) that represents the day of the week.

Example

The following expression returns 1 which represents SUNDAY:

```
@WEEKDAY(12/28/86)
```

The following SQL statement finds the day of the week on which each employee was hired.

```
SELECT @CHOOSE (@WEEKDAY(HIREDATE), 'Sat','Sun','Mon',
    'Tue', 'Wed', 'Thu', 'Fri'), @YEAR(HIREDATE) FROM EMP
    WHERE @YEARNO (HIREDATE) > 1990;
```

@YEAR

@YEAR(*date*)

This function returns a number between -1900 and +200 that represents the year relative to 1900. The year 1900 is 0, 1986 is 86, and 2000 is 100. Years before 1900 are negative numbers and 1899 is -1.

Example

The following expression returns 23.

```
@YEAR(12/28/1923)
```

The following SQL statement finds the day of the week on which each employee was hired.

```
SELECT @CHOOSE (@WEEKDAY(HIREDATE), 'Sat','Sun','Mon',
    'Tue', 'Wed', 'Thu', 'Fri'), @YEAR(HIREDATE) FROM EMP
    WHERE @YEARNO (HIREDATE) > 1990;
```

@YEARBEG

@YEARBEG(*date*)

This function returns the first day of the year represented by the date.

Example

If the value in HIREDATE is '16-FEB-1996', then the following expression returns 01-JAN-1996:

```
@YEARBEG(HIREDATE)
```

@YEARNO

@YEARNO(*date*)

This function returns a 4-digit number that represents a calendar year.

Example

If the column HISTORIC_DATE contains the value 04/JUL/1776, then the expression returns 1776:

```
@YEARNO(HISTORIC_DATE)
```

The following SQL statement finds the day of the week on which each employee was hired.

```
SELECT @CHOOSE (@WEEKDAY(HIREDATE), 'Sat','Sun','Mon',
    'Tue', 'Wed', 'Thu', 'Fri'), @YEAR(HIREDATE) FROM EMP
    WHERE @YEARNO (HIREDATE) > 1990;
```

@LEFT

@LEFT(*string, length*)

This function returns a string for the specified length, starting with the first (leftmost) character in the string.

Example

The following expression returns 'P8':

```
@LEFT('P8-196', 2)
```

The following example shows how to use the @LEFT function in a SELECT statement:

```
SELECT * FROMSYSCOLUMNS
    WHERE @LEFT (TBNAME, 3) ! = 'SYS';
```

@LENGTH

@**LENGTH(***string***)**

This function returns the length of a string. The length is the number of characters in the string.

You cannot use this function to find the length of a LONG VARCHAR column.

Example

If the value in the column EMPNAME is 'JOYCE', the following expression returns the number 5:

```
@LENGTH( EMPNAME )
```

The following example finds the entries in the EMP table where the length of the LNAME column exceeds 10 characters.

```
SELECT EMPNO, @SUBSTRING (LNAME, 0, 10)
    FROM EMP WHERE @LENGTH (LNAME) > 10
```

@LICS

@**LICS(***string***)**

This function uses an international character set for sorting its argument, instead of the ASCII character set. This is useful for sorting characters not in the English language. The translation table for this character set is shown below.

Example

The following expression returns 'NXTRI\' @LICS ('Murphy')

```
SELECT @LICS(LNAME) FROM EMP ORDER BY 1;
```

Code	Character	Description
0	0	Ctrl @
1	1	Ctrl A

Code	Character	Description
2	2	Ctrl B
3	3	Ctrl C
4	4	Ctrl D
5	5	Ctrl E
6	6	Ctrl F
7	7	Ctrl G
8	8	Ctrl H
9	9	Ctrl I
10	10	Ctrl J line feed
11	11	Ctrl K
12	12	Ctrl L form feed
13	13	Ctrl M return
14	14	Ctrl N
15	15	Ctrl O
16	16	Ctrl P
17	17	Ctrl Q
18	18	Ctrl R
19	19	Ctrl S
20	20	Ctrl T
21	21	Ctrl U
22	22	Ctrl V
23	23	Ctrl W
24	24	Ctrl X
25	25	Ctrl Y
26	26	Ctrl Z
27	27	[Esc]

Code	Character	Description
28	28	FS
29	29	GS
30	30	RS
31	31	US
32	32	Space
33	33	!
34	34	"
35	35	#
36	36	$
37	37	%
38	38	&
39	39	Apostrophe
40	40	(
41	41	)
42	42	*
43	43	+
44	44	
45	45	-
46	46	.
47	47	/
48	48	0
49	49	1
50	50	2
51	51	3
52	52	4
53	53	5

Code	Character	Description
54	54	6
55	55	7
56	56	8
57	57	9
58	58	:
59	59	;
60	60	<
61	61	=
62	62	>
63	63	?
64	64	@
65	65	A
66	66	B
67	67	C
68	68	D
70	69	E
71	70	F
72	71	G
73	72	H
74	73	I
75	74	J
76	75	K
77	76	L
78	7 7	M
79	78	N
81	79	O

Code	Character	Description
82	80	P
83	81	Q
84	82	R
85	83	S
87	84	T
88	85	U
89	86	V
90	87	W
91	88	X
92	89	Y
93	90	Z
99	91	[
100	92	\
101	93	]
102	94	^
103	95	_
104	96	`
65	97	a
66	98	b
67	99	c
68	100	d
70	101	e
71	102	f
72	103	g
73	104	h
74	105	i

Code	Character	Description
75	106	j
76	107	k
77	108	l
78	109	m
79	110	n
81	111	o
82	112	p
83	113	q
84	114	r
85	115	s
87	116	t
88	117	u
89	118	v
90	119	w
91	120	x
92	121	y
93	122	z
105	123	{
106	124	|
107	125	}
108	126	~ (tilde)
109	127	DEL
110	128	Uppercase grave
111	129	Uppercase acute
112	130	Uppercase circumflex
113	131	Uppercase umlaut

Code	Character	Description
114	132	Uppercase tilde
115	133	
116	134	
117	135	
118	136	
119	137	
120	138	
121	139	
122	140	
123	141	
124	142	
125	143	
126	144	Lowercase grave
127	145	Lowercase acute
128	146	Lowercase circumflex
129	147	Lowercase umlaut
130	14 8	Lowercase tilde
131	149	Lowercase i without dot
132	150	Ordinal indicator
133	151	Begin attribute (display)
134	152	End attribute (display only)
135	153	Unknown character (display)
136	154	Hard space (display only)
137	155	Merge character (display)
138	156	
139	157	

Code	Character	Description
140	158	
141	159	
142	160	Dutch Guilder
143	161	Inverted exclamation mark
144	162	Cent sign
145	163	Pound sign
146	164	Low opening double quotes
147	165	Yen sign
148	166	Pesetas sign
149	167	Section sign
150	168	General currency sign
151	169	Copyright sign
152	170	Feminine ordinal
153	171	Angle quotation mark left
154	1 72	Delta
155	173	Pi
156	174	Greater-than-or-equals
157	175	Divide sign
158	176	Degree sign
159	177	Plus/minus sign
160	178	Superscript 2
161	179	Superscript 3
162	180	Low closing double quotes
163	181	Micro sign
164	182	Paragraph sign
165	183	Middle dot

Code	Character	Description
166	184	Trademark sign
167	185	Superscript 1
168	186	Masculine ordinal
16 9	187	Angle quotation mark right
170	188	Fraction one quarter
171	189	Fraction one-half
172	190	Less-than-or -equals
173	191	Inverted question mark
65	192	Uppercase A with grave
65	193	Uppercase A with acute
65	194	Uppercase A with circumflex
65	195	Uppercase A with tilde
65	196	Uppercase A with umlaut
65	197	Uppercase A with ring
97	197	Uppercase A with ring
94	198	Uppercase AE with ligature
67	199	Uppercase C with cedilla
70	200	Uppercase E with grave
70	201	Uppercase E with acute
70	202	Uppercase E with circumflex
70	203	Uppercase E with umlaut
74	204	Uppercase I with grave
74	205	Uppercase I with acute
74	206	Uppercase I with circumflex
74	207	Uppercase I with umlaut
69	208	Uppercase eth (Icelandic)

Code	Character	Description
80	209	Uppercase N with tilde
81	210	Uppercase O with grave
81	211	Uppercase O with acute
81	212	Uppercase O with circumflex
81	213	Uppercase O with tilde
81	214	Uppercase O with umlaut
80	215	Uppercase OE with diphthong
96	216	Uppercase O with slash
88	217	Uppercase U with grave
88	218	Uppercase U with acute
88	219	Uppercase U with circumflex
88	220	Uppercase u with umlaut
92	221	Uppercase Y with umlaut
98	222	Uppercase thorn (Icelandic)
86	223	Lowercase German sharp s
65	224	Lowercase a with grave
65	225	Lowercase a with acute
65	226	Lowercase a with circumflex
65	227	Lowercase a with tilde
65	228	Lowercase a with umlaut
65	229	Lowercase a with ring
95	230	Lowercase ae with ligature
6 7	231	Lowercase c with cedilla
70	232	Lowercase e with grave
70	233	Lowercase e with acute
70	234	Lowercase e with circumflex

Code	Character	Description
70	235	Lowercase e with umlaut
74	236	Lowercase i with grave
74	237	Lowercase i with acute
74	238	Lowercase i with circumflex
74	239	Lowercase i with umlaut
69	240	Lowercase eth (Icelandic)
80	241	Lowercase n with tilde
81	242	Lowercase o with grave
81	243	Lowercase o with acute
81	244	Lowercase o with circumflex
81	245	Lowercase o with tilde
81	246	Lowercase o with umlaut
80	247	Lowercase oe with diphthong
81	248	Lowercase o with slash
88	249	Lowercase u with grave
88	250	Lowercase u with acute
88	251	Lowercase u with circumflex
88	252	L owercase u with umlaut
92	253	Lowercase y with umlaut
174	254	Lowercase thorn (Icelandic)

@LN

@LN(*x*)

This function returns the natural logarithm (base e) of (positive) *x*. The log of a zero or negative argument is handled as an overflow error.

Example

The following expression returns -2.3025851:

```
@LN(.1)
```

The following SQL statement returns the natural logarithm of all PVAL column entries in the GEOM table:

```
SELECT @LN(PVAL) FROM GEOM;
```

@LOG

@LOG(*x*)

This function returns the (positive) base-10 logarithm of *x*. The log of a zero or negative argument is handled as an overflow error.

Example

The following expression returns -1:

```
@LOG(.1)
```

The following SQL statement returns the natural logarithm of all PVAL column entries in the GEOM table:

```
SELECT @LOG(PVAL) FROM GEOM;
```

@LOWER

@LOWER(*string*)

This function converts upper-case alphabetic characters to lower-case. Other characters are not affected.

Example

The following expression returns the string 'joyce':

```
@LOWER('JOYCE')
```

@MEDIAN

This function returns the middle value in a set of values. An equal number of values lie above and below the middle value.

The data type of the argument may be numeric, date/time, or character. If an argument is a character data type, the value must form a valid numeric or date/time value (only digits and standard editing characters). SQLBase automatically converts the value to the required data type. The data type of the result is the same as the input argument.

@MEDIAN finds the middle value with this formula:

```
( n + 1 ) / 2
```

For example, if there are 5 items, then the middle item is the third:

```
( 5 + 1 ) / 2 = 6 / 2 = 3
```

For example, if there are 6 items, then the middle item is between the third and the fourth:

```
( 6 + 1) / 2 = 7 / 2 = 3.5
```

The median is the arithmetic average of the third and fourth values.

The keyword DISTINCT eliminates duplicates. If DISTINCT is not specified, then duplicates are not eliminated. Be cautious when using DISTINCT because the result may loose its statistical meaning.

Null values are ignored.

Example

This example finds the middle salary for department 2500.

```
SELECT @MEDIAN(SALARY) FROM EMP, EMPSAL
   WHERE EMP.EMPNO = EMPSAL.EMPNO AND DEPTNO=2500;
```

@MICROSECOND

@MICROSECOND(*date*)

This function returns the microsecond value in a DATETIME or TIME value. If a microsecond quantity was not specified on input, zero is returned.

Example

The following expression returns 500000:

```
@MICROSECOND(12:44:01:500000)
```

@MID

@MID(*string, start-pos, length*)

This function returns a string of specified length from a string, starting with the character at *start-pos*. This function is similar to @SUBSTRING, except that it requires the third argument.

Example

The following expression returns the second character from a string, a '9':

```
@MID('P9-186', 1, 1)
```

@MINUTE

@MINUTE(*date*)

This function returns a number between 0 and 59 that represents the minute of the hour.

Example

The following expression returns 52:

```
@MINUTE(12/28/46 03:52:00 PM)
```

@MOD

@MOD(*x, y*)

This function returns the modulo (remainder) of *x*/*y*. Division by zero is an overflow error.

Example

The following expression returns 5:

```
@MOD(5,10)
```

The following SQL statement returns the remainder of all PVAL/WVAL column entries in the GEOM table:

```
SELECT @MOD(PVAL,WVAL) FROM GEOM;
```

@MONTH

@MONTH(*date*)

This function returns a number between 1 and 12 that represents the month of the year.

Example

The following expression returns 10 which represents October:

```
@MONTH(25-OCT-96)
```

@MONTHBEG

@MONTHBEG(*date*)

This function returns the first day of the month represented by the date.

Example

If the value in BIRTHDATE is '16-FEB-1947', then the following expression returns 01-FEB-1947:

```
@MONTHBEG(BIRTHDATE)
```

@NOW

@NOW

This function returns the current date and time. It returns the same value as the system keyword SYSDATETIME.

For example, if the date and time is January 12, 1996, 3:15 PM, this function would return 12-JAN-1996 03:15:00 PM.

@NULLVALUE

@NULLVALUE(*x, y*)

This function returns one of the following values specified by *y* if *x* is null:

- string
- number

- date (if the date is a constant. If you try to specify a date by a bind variable such as *1:*, the bind variable is read literally, since it is treated as a CHAR value.)

The data type of the returned value is the same as the data type of the *x* argument.

SQLBase converts the second parameter (*y* argument) to the first parameter's data type (*x* argument). An error results if SQLBase cannot convert this correctly.

Example

The following example returns "N/A" when the column is null:

```
@NULLVALUE(FNAME, 'N/A')
```

The following SQL statement:

```
SELECT @NULLVALUE(DEPTNO,'NOT ASSIGNED') FROM EMP;
```

returns the string 'NOT ASSIGNED' if the DEPTNO column value is null, and DEPTNO is a character column. If the column is numeric, the replacement value must be a number. For example, the following SQL statement:

```
SELECT @NULLVALUE(DEPTNO,9999) FROM EMP;
```

returns 9999 if a null exists in the DEPTNO column. DEPTNO is a numeric data type.

@PI

@PI

This function returns the value Pi (3.14159265). This function has no arguments but could be used as a numeric constant in a nested set of math functions.

Example

The following expression returns 31.4159265:

```
10 * @PI
```

The following SQL statement returns all PVAL column entries multiplied by the value Pi in the GEOM table:

```
SELECT (PVAL) * @PI FROM GEOM;
```

@PMT

@PMT(*principal, interest, periods*)

This function returns the amount of each periodic payment needed to pay off a loan principal (*prin*) at a periodic interest rate (*int*) over a number of periods (*n*).

@PMT uses this formula:

$$\frac{print * int}{(1 - (1 + int)^{-n})}$$

prin = principal

int = periodic interest rate

n = number of periods; term

Example

The following expression:

```
@PMT(50000,.125/12,30 * 12)
```

returns $533.628881 which is the value of a monthly mortgage payment for a $50,000, 30-year mortgage at an annual interest rate of 12.5%.

@PROPER

@PROPER(*string*)

This function converts the first character of each word in a string to uppercase and other characters to lower case.

The argument must be a CHAR or VARCHAR data type.

Example

The following expressions both return 'Johann Sebastian Bach':

```
@PROPER('JOHANN SEBASTIAN BACH')

@PROPER('johann sebastian bach')
```

@PV

@PV(*pmt, int, n*)

This function returns the present value of a series of equal payments (*pmt*) discounted at periodic interest rate (*int*) over the number of periods (*n*).

This function is useful when trying to decide the best way to receive a payment option, over time or immediately.

@PV uses this formula:

$$pmt * \frac{(1 - (1 + int)^{-n})}{int}$$

pmt = periodic payment

int = periodic interest rate

n = number of periods; term

Ordinary Annuity Example

The following expression:

```
@PV(50000,.12,20)
```

returns $373,472.181 which is what $1,000,000 paid equally ($50,000 at the end of each year) over 20 years at 12% is worth today.

Annuity Due Example

The following expression:

```
@PV(50000,.12,20) * (1+.12)
```

returns $418,288.843, which is what $1,000,000 paid equally ($50,000 at the beginning of each year) over 20 years at 12% is worth today.

@QUARTER

@QUARTER(*date*)

This function returns a number between 1 and 4 that represents the quarter. For example, the first quarter of the year is January through March.

Example

The following expression returns 1, which represents the first quarter:

```
@QUARTER(12-MAR-96)
```

@QUARTERBEG

@QUARTERBEG(*date*)

This function returns the first day of the quarter represented by the date.

Example

The following expression returns 01-JUL-1776:

```
@QUARTERBEG(04-JUL-1776)
```

The following SQL statement displays the first day of the quarter in which each employee was hired:

```
SELECT @QUARTERBEG (HIREDATE) FROM EMP;
```

@RATE

@RATE(*fv, pv, n*)

This function returns the interest rate for an investment of present value (*pv*) to grow to a future value (*fv*) over the number of compounding periods (*n*).

@RATE uses this formula:

$$((fv/pv)^{(1/n)}) - 1$$

fv = future value

pv = present value

n = number of periods; term

Example

The following expression:

```
@RATE(18000,10000,5 * 12)
```

returns .009844587 which is the periodic (monthly) interest rate calculated for a
$10,000 investment for 60 months (5 years) with a maturity value of $18,000
(compounded monthly).

@REPEAT

@REPEAT(*string, number*)

This function concatenates a string with itself for the specified number of times. This
creates a string of pattern repetitions.

This function returns nulls if specified in a select list. However, it can be used in a
WHERE clause and in other contexts.

Example

The following expression returns the value '$$$$$':

```
@REPEAT('$',5)
```

@REPLACE

@REPLACE(*string1, start-pos, length, string2*)

This function returns a string in which characters from *string1* have been replaced
with characters from *string2*. The replacement *string2* begins at *start-pos,* the position
at which characters of the specified *length* have been removed.

The first position in the string is 0.

Example

The following expression returns the value 'RALPH':

```
@REPLACE('RALF', 3, 1, 'PH')
```

@RIGHT

@**RIGHT(*string, length*)**

This function returns a specified number of characters starting from the end, or rightmost part, of a string.

Example

The following expression returns '186':

```
@RIGHT('P4-186', 3)
```

@ROUND

@**ROUND(*x, n*)**

This function rounds the number *x* with *n* decimal places. The rounding can occur to either side of the decimal point.

Example

The following expression returns 31.42:

```
@ROUND(@PI * 10,2)
```

The following expression returns 1200:

```
@ROUND(1234.1234,-2)
```

The following SQL statement returns the value of all PVAL column entries in the GEOM table, rounded to 2 decimal places to the RIGHT of the decimal point.t:

```
SELECT @ROUND(QVAL,2) FROM GEOM;
```

The following SQL statement returns the value of all PVAL column entries in the GEOM table, rounded to 2 decimal places to the LEFT of the decimal point:

```
SELECT @ROUND(QVAL,-2) FROM GEOM;
```

@SCAN

@SCAN(*string, pattern*)

This function searches a given string for a specified pattern and returns a number indicating the numeric position of the first instance of the pattern.

This function returns null if the column being scanned is null.

The first position in the string is position 0. The match is performed without regard to case.

If the result is -1, it indicates no match was found.

The @SCAN function can perform a case-insensitive match on columns of type CHAR, VARCHAR, and LONG VARCHAR.

Example

The following expression returns 1 as the start position of the character '-':

```
@SCAN('P-186', '-')
```

@SDV

This function computes the standard deviation for the set of values specified by the argument.

The data type of the argument may be numeric, date/time, or character. If an argument is a character data type, the value must form a valid numeric or date/time value (only digits and standard editing characters). SQLBase automatically converts the value to the required data type.

The keyword DISTINCT eliminates duplicates. If DISTINCT is not specified, then duplicates are not eliminated.

Note that this function produces double precision, which is not the same as an integer value.

Example

The following SQL statement returns the standard deviation of the SALARY column in the table EMPSAL.

```
SELECT @SDV(SALARY) FROM EMPSAL;
```

@SECOND

@**SECOND(***date***)**

This function returns a number between 0 and 59 that represents the second of the minute.

Example

The following expression returns 58:

```
@SECOND(12/28/46 03:52:58)
```

@SIN

@**SIN(***x***)**

This function returns the sine of x, where x is in *radians*.

Example

The following expression returns .841470985:

```
@SIN(1)
```

The following SQL statement returns the value of all PVAL column entries in the GEOM table:

```
SELECT @SIN(PVAL) FROM GEOM;
```

@SLN

@SLN(*cost, salvage, life*)

This function returns the straight-line depreciation allowance of an asset for each period, given the base cost, predicted salvage value, and expected life of the asset.

@SLN uses this formula to compute depreciation:

$$\frac{(c - s)}{n}$$

c = cost of the asset

s = salvage value of the asset

n = useful life of the asset

Example

The following expression:

```
@SLN(10000,1200,8)
```

returns $1100, which is the yearly depreciation allowance for a machine purchased for $10,000, with a useful life of 8 years, and a salvage value of $1200 after the 8 years.

@SQRT

@SQRT(*x*)

This function returns the square root of x (which must be zero or positive). The square root of a negative argument is handled as an overflow error.

Example

The following expression returns 3.16227766:

```
@SQRT(10)
```

The following SQL statement returns the square root of all PVAL column entries in the GEOM table:

```
SELECT @SQRT(PVAL) FROM GEOM;
```

@STRING

@**STRING(***number, scale***)**

This function converts a number into a string with the number of decimal places specified by *scale*. Numbers are rounded where appropriate.

Example

The following expression returns the character string '123.46':

```
@STRING(123.456, 2)
```

@SUBSTRING

@**SUBSTRING(***string, start-pos, length***)**

This function returns a desired portion of a string from a given argument string. The substring starts at the specified start position and is of the specified length. If the start position and length define a substring that exceeds the actual length of the string, the result is truncated to the actual length of the string. If the start position is beyond length of the string, a null string ('') is returned. The first character in a string is at start-pos 0.

The length parameter is optional.

Example

The following expression returns 'SMITH':

```
@SUBSTRING('DR. SMITH', 4, 20)
```

The following example returns the first 10 characters of the LNAME column in the EMP table where the length of the LNAME column exceeds 10 characters.

```
SELECT EMPNO, @SUBSTRING(LNAME, 0, 10) FROM EMP WHERE
  @LENGTH(LNAME) > 10;
```

The function is nearly the same as @MID$ except that if the third argument is left off, the function returns a string beginning with the start position.

The following expression returns 'R. SMITH':

```
@SUBSTRING ('DR. SMITH', 1)
```

@SYD

@SYD(*cost, salvage, life, period*)

This function returns the Sum-of-the-Years'-Digits depreciation allowance of an asset for a given period, given the base cost, predicted salvage value, expected life of the asset and specific period.

@SYD uses this formula to compute depreciation:

$(c - s) * (n - p + 1)$	c = cost of the asset
$(n * (n + 1)/2)$	s = salvage value of the asset
	p = period for which depreciation is being computed
	n = useful life of the asset

Example

The following expression:

```
@SYD(10000,1200,8,5)
```

returns $977.777778, which is the depreciation allowance for the fifth year for a $10,000 machine with a useful life of 8 years, and a salvage value of $1200 after the 8 years.

@TAN

@TAN(*x*)

This function returns the tangent of *x*, where *x* is in radians.

Example

The following expression returns .648360827:

```
@TAN(10)
```

The following SQL statement returns the tangent of all PVAL column entries in the GEOM table:

```
SELECT @TAN(PVAL) FROM GEOM;
```

@TERM

@TERM(*pmt, int, fv*)

This function returns the number of payment periods for an investment, given the amount of each payment *pmt*, the periodic interest rate *int*, and the future value *fv* of the investment.

@TERM uses this formula to compute the term:

ln (1 + (fv * int/pmt))	pmt = periodic payment
ln (1 + int)	fv = future value
	int = periodic interest rate
	ln = natural logarithm

Example

The following expression:

```
@TERM(2000,.10,100000)
```

returns 18.7992455, which is the number of years it will take for an investment to mature to an amount of $100,000. This is based on a yearly deposit of $2,000 at the end of each year to an account that earns 10% compounded annually.

@TIME

@TIME(*hour, minute, second*)

This function returns a time value given the *hour, minute,* and *second.* An hour is a number from 0 to 23; a minute is a number from 0 to 59; a second is a number from 0 to 59.

Example

The following expression returns 13:00:00:

```
@TIME(13,0,0)
```

@TIMEVALUE

@TIMEVALUE(*time*)

The function returns a time value, given a string in the form HH:MM:SS [AM or PM]. If the AM or PM parameter is omitted, military time is used.

Example

If the CHAR column APPT contains '18-JAN-1994 10:14:27 AM', then the following expression returns 10:14:27:

```
@TIMEVALUE(APPT)
```

@TRIM

@TRIM(*string*)

This function strips leading and trailing blanks from a string and compresses multiple spaces within the string into single spaces.

Example

The following expression returns 'JOHN DEWEY':

```
@TRIM('   JOHN    DEWEY ')
```

@UPPER

@UPPER(*string*)

This function converts lower-case letters in a string to upper-case. Other characters are not affected.

Example

The following expression returns 'E.E. CUMMINGS':

```
@UPPER('e.e. cummings')
```

@VALUE

@VALUE(*string*)

This function converts a character string that has the digits (0-9) and an optional decimal point or negative sign into the number represented by that string.

Example

The following expression returns the number 123456 which will be interpreted strictly as a numeric data type by any function to which it is passed:

```
@VALUE('123456')
```

@WEEKBEG

@**WEEKBEG(***date***)**

This function returns the date of the Monday of the week containing the date. This is the previous Monday if the date is not a Monday, and the date value itself if it is a Monday.

Example

If the value in DATECOL is 01/FEB/94, then the following expression returns 31-JAN-1994:

```
@WEEKBEG(DATECOL)
```

@WEEKDAY

@**WEEKDAY(***date***)**

This function returns a number between 0 and 6 (Saturday = 0 and Friday = 6) that represents the day of the week.

Example

The following expression returns 1 which represents SUNDAY:

```
@WEEKDAY(12/28/86)
```

The following SQL statement finds the day of the week on which each employee was hired.

```
SELECT @CHOOSE (@WEEKDAY(HIREDATE), 'Sat','Sun','Mon',
    'Tue', 'Wed', 'Thu', 'Fri'), @YEAR(HIREDATE) FROM EMP
    WHERE @YEARNO (HIREDATE) > 1990;
```

@YEAR

@YEAR(*date*)

This function returns a number between -1900 and +200 that represents the year relative to 1900. The year 1900 is 0, 1986 is 86, and 2000 is 100. Years before 1900 are negative numbers and 1899 is -1.

Example

The following expression returns 23.

```
@YEAR(12/28/1923)
```

The following SQL statement finds the day of the week on which each employee was hired.

```
SELECT @CHOOSE (@WEEKDAY(HIREDATE), 'Sat','Sun','Mon',
    'Tue', 'Wed', 'Thu', 'Fri'), @YEAR(HIREDATE) FROM EMP
    WHERE @YEARNO (HIREDATE) > 1990;
```

@YEARBEG

@YEARBEG(*date*)

This function returns the first day of the year represented by the date.

Example

If the value in HIREDATE is '16-FEB-1996', then the following expression returns 01-JAN-1996:

```
@YEARBEG(HIREDATE)
```

@YEARNO

@YEARNO(*date*)

This function returns a 4-digit number that represents a calendar year.

Example

If the column HISTORIC_DATE contains the value 04/JUL/1776, then the expression returns 1776:

```
@YEARNO(HISTORIC_DATE)
```

The following SQL statement finds the day of the week on which each employee was hired.

```
SELECT @CHOOSE (@WEEKDAY(HIREDATE), 'Sat','Sun','Mon',
    'Tue', 'Wed', 'Thu', 'Fri'), @YEAR(HIREDATE) FROM EMP
    WHERE @YEARNO (HIREDATE) > 1990;
```

Chapter 5

SQL Reserved Words

This chapter lists the SQL reserved words.

SQL Reserved Words

The following words are reserved in SQL.

You can use a reserved word as an identifier if it is enclosed in double quotes, but this is *not* recommended.

@ABS	@ACOS
@ASIN	@ATAN
@ATAN2	@CHAR
@CHOOSE	@CODE
@COS	@CTERM
@DATE	@DATETOCHAR
@DATEVALUE	@DAY
@DECIMAL	@DECODE
@EXACT	@EXP
@FACTORIAL	@FIND
@FULLP	@FV
@HALFP	@HEX
@HOUR	@IF
@INT	@ISNA
@LEFT	@LENGTH
@LICS	@LN
@LOG	@LOWER
@MEDIAN	@MICROSECOND
@MID	@MINUTE
@MOD	@MONTH
@MONTHBEG	@NOW
@NULLVALUE	@PI
@PMT	@PROPER
@PV	@QUARTER
@QUARTERBEG	@RATE
@REPEAT	@REPLACE
@RIGHT	@ROUND
@SCAN	@SDV
@SECOND	@SIN
@SLN	@SQRT
@STRING	@SUBSTRING
@SYD	@TAN
@TERM	@TIME
@TIMEVALUE	@TRIM
@UPPER	@VALUE
@WEEKBEG	@WEEKDAY
@YEAR	@YEARBEG

@YEARNO	ABORTxxxDBSxxx
ACTIONS	ADD
ADJUSTING	AFTER
ALL	ALTER
AND	ANY
APPEND	AS
ASC	ASCII
AT	ATTRIBUTE
AUDIT	AUTHORITY
AVG	BEFORE
BETWEEN	BUCKETS
BY	CALLSTYLE
CASCADE	CATALOG
CATEGORY	CDECL
CHAR	CHARACTER
CHECK	CLIENT
CLUSTERED	COLAUTH
COLUMN	COMMENT
COMMIT	COMPRESS
COMPUTE	CONNECT
CONTROL	COUNT
CR	CREATE
CREATOR	CURRENT
CURRVAL	DATABASE
DATEDATE	DATETIME
DAY	DAYS
DBA	DBATTRIBUTE
DBAREA	DEC
DECIMAL	DEFAULT
DEINSTALL	DELETE
DESC	DIF
DIRECT	DISABLE
DISTINCT	DISTINCTCOUNT
DOUBLE	DROP
DYNAMIC	EACH
ENABLE	EVENT
EVERY	EXECUTE
EXISTS	EXTERNAL
FLOAT	FOR
FORCE	FOREIGN
FROM	FUNCTION
GLOBAL	GRANT
GRANTEE	GROUP

HASHED	HAVING
HOUR	HOURS
ID	IDENTIFIED
IN	INDEX
INDEXES	INLINE
INSERT	INSTALL
INT	INTEGER
INTO	IS
IXNAME	KEEP
KEY	LABEL
LF	LIBRARY
LIKE	LIMIT
LOAD	LOCAL
LOCK	LOG
LONG	MAX
MESSAGE	MICROSECOND
MICROSECONDS	MIN
MINUTE	MINUTES
MODIFY	MONTH
MONTHS	NAME
NEW	NEXTVAL
NOT	NULL
NUMBER	OF
OFF	OLD
ON	ONLY
OPTION	OR
ORDER	OVERWRITE
PARAMETERS	PASCAL
PASSWORD	PCTFREE
PERFM	POST
PRECISION	PRIMARY
PRIVILEGES	PROCEDURE
PROCESS	PUBLIC
QUALIFIER	RAISE
REAL	REFERENCES
REFERENCING	REL
RENAME	RESOURCE
RESTRICT	RETURNS
REVOKE	ROLLBACK
ROW	ROWCOUNT
ROWID	ROWS
SAME	SAVEPOINT
SCHEMA	SECOND

SECONDS	SELECT
SEPARATE	SERVER
SET	SIZE
SMALLINT	SQL
START	STATEMENT
STATIC	STATISTICS
STDCALL	STOGROUP
SUM	SYNCREATOR
SYNNAME	SYNONYM
SYSDATE	SYSDATETIME
SYSDBSEQUENCE	SYSTIME
SYSTIMEZONE	SYSDBTRANSID
TABAUTH	TABLE
TBNAME	THREAD
TIME	TIMESTAMP
TIMEZONE	TO
TRANSACTION	TRIGGER
TYPE	UNLOAD
UNION	UNIQUE
UPDATE	USER
USERERROR	USING
VALUES	VARCHAR
VARIABLES	VIEW
WAIT	WHERE
WITH	WITHOUT
WORK	YEAR
YEARS	

Chapter 6

Referential Integrity

This chapter describes how referential integrity works, and how it affects SQLBase commands, and its components.

About referential integrity

Referential integrity ensures that all references from one table to another are valid. This prevents problems from occurring when changes in one table are not reflected in another.

To illustrate the concept of referential integrity, assume that you have a table called ENGINEERS where you store information about service engineer employees. You need to add an engineer to a new office in this table:

```
INSERT INTO ENGINEERS (EMPL_NUM,NAME,REP_OFFICE,
    TITLE,HIRE_DATE) VALUES (400,'Marv Epper',50,
    'Engineer', 10/1/93,NULL);
```

There's nothing inherently incorrect about this statement. However, if office 50 does not yet exist, this record could potentially corrupt the data integrity.

Every office value in the ENGINEERS table should be a valid office in the OFFICES table. This rule is called a referential integrity *constraint*.

Note that a valid reference is not the same as a correct reference. Referential integrity does not correct a mistake such as assigning an engineer to the *wrong* office; it only verifies that the office actually exists.

Sample service database

To demonstrate referential integrity, this chapter uses a small database for a camera company's service organization. This database contains the following tables:

- OFFICES
- ENGINEERS
- CUSTOMERS
- SERV_CALLS
- PRODUCTS

For a listing of the data in these tables, refer to the end of this chapter.

The benefits of referential integrity

Referential integrity is an important SQLBase feature. It takes care of data integrity and validation at the database level. For example, assume you need to enforce the following constraints in the sample service database:

- There can be only one manager per office.
- All employees must be associated with an office and manager.
- Each product has a manufacturer and product code.

•   All customers have a service representative assigned to them.

SQLBase can take care of these referential integrity constraints; you do not have to code them yourself in your application program. SQLBase, not the user, maintains and enforces the referential integrity rules.

Components

Referential integrity is the enforcement of all referential constraints. To understand how referential integrity works, you first need to be familiar with its main components:

•   primary key

•   foreign key

•   parent/child table

•   parent/child row

•   self-referencing table/row

Primary key

A table's *primary key* is the column or set of columns that uniquely identifies each row. In the OFFICES table, the OFFICE column is the primary key. It is a unique identifier since each office has a different number.

Primary Key

OFFICE	CITY	REGION	MGR	MAJ_ ACCOUNT
20	San Francisco	Western	103	1050
40	New York	Eastern	108	2500
10	Los Angeles	Western	100	3000
30	Chicago	Midwest	106	1001

Primary key for OFFICES table

Primary keys ensure the integrity of the data. If the primary key is correctly used and maintained, every row is different from every other row, and there are no empty rows. A table can have only one primary key. The primary key cannot contain any NULL values, and must be unique.

Composite primary key

Sometimes, more than one column is necessary to uniquely define a row. A table can have a *composite primary key* containing multiple columns. For example, the PRODUCTS table has a primary key containing two columns: MFR_ID and PRODUCT_ID. Neither column could be a primary key by itself, but together, these two columns uniquely identify each product.

Primary Key
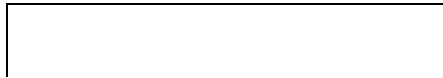
MFR_ID	PRODUCT_ID	DESCRIPTION
ACR	101	Tripod
ACR	102	Tripod2
MRP	101	Long Angle Lens
LMA	4211	Automatic Camera
LMA	4310	Regular Focus 1
LMA	4516	Regular Focus 2
MRP	600	Lens
MRP	601	Shutter
WRS	24c	Widget 1
WRS	25a	Widget 2

Composite primary key for PRODUCTs example

Candidate keys

A table can have more than one unique identifier that qualifies as a primary key. Each column that is a unique identifier for the table is called a *candidate key*, and each can be the primary key.

A candidate key must obey the following rules:

- • No two rows in the table can have the same value for the candidate key.

- The candidate key is not allowed to contain subsets that are unique. For example, the composite key MFR_ID/PRODUCT_ID is not a candidate key if either of its columns is also unique in the table.

You must choose yourself which candidate key is the primary key. The remaining candidate keys are called *alternate keys*.

Guidelines for defining primary keys

The following rules are not required, but are good guidelines for creating a primary key.

- *Unique identifier.* Create a primary key for every table that has a clear unique identifier, such as the OFFICE column in the OFFICES table. SQLBase does allow you to create a table without a primary key. However, it is strongly recommended that you never do this, except in the following situations:

  - There are not any referential rules applied to the table.

  - The table is not a parent table (see the following section on *Components*).

  - The index maintenance overhead clearly outweighs the benefits of a primary key.

- *Permanent value.* If there are child rows referencing the primary key (see the following section on *Components*), a primary key value should be permanent, and not updateable.

  For example, at first glance the MGR or CITY columns in the OFFICES table could also be primary key candidates since they are currently unique. However, if you open another office in the same city, or a manager leaves the company, the values in these columns would change. Neither of these columns would work well as a primary key, since their values may not always be unique or permanent.

- *Views.* An updateable view defined on a table with a primary key must include all columns of the primary key. Although this is only required if you use the view in an INSERT statement, the resulting unique identification of rows is also useful if the view is used for updating, deleting, or selecting.

  If you try to insert a row into a view that does not contain values for all of the primary key columns, the following message appears:

  ```
  NOT ENOUGH NON-NULL VALUES
  ```

  This message appears because all the primary key columns are defined as NOT NULL (since a primary key cannot contain NULL values).

- *Number of columns.* For composite primary keys, use only the minimum number of columns necessary to ensure uniqueness of the primary key. This is because every foreign key referencing this primary key must include the

same number of columns. For example, in the PRODUCTS table, you only need the manufacturer number and product number, not the description.

- *NOT NULL WITH DEFAULT.* When creating primary keys, you should not use the NOT NULL WITH DEFAULT option unless the primary key column(s) has a data type of TIMESTAMP or DATETIME.

The following rules are required when creating a primary key in SQLBase:

- *Unique index.* If a table has a primary key, you must also create a unique index on the primary key columns to make the table complete. See the following section *Primary key index* for more information.

- *Format.* The primary key format must obey the following rules:

  - Cannot contain more than 16 columns.

  - Sum of the column length attributes cannot be greater than 255 bytes.

  - Cannot contain LONG or LONG VARCHAR columns.

- *UPDATE WHERE CURRENT.* You cannot use an UPDATE WHERE CURRENT clause with a primary key column.

- *Self-referenced rows.* In a self-referencing row, you cannot update the primary key value. For more information on self-referenced rows, see the following section on *Self-referenced rows*.

Primary index

If a table has a primary key, you must also create a unique index on that table's primary key columns using the same column order as the primary key. This index is called the *primary index*. A table can have only one primary index. If a table has more than one unique index created on the primary key columns, the first index created is the primary index.

The primary index can be in either ascending or descending order. The table is in an *incomplete* state until you create the primary index. If the table is incomplete, you cannot perform tasks such as inserting or retrieving data, or creating foreign keys that reference the primary key.

Because of these limitations, create the primary index soon after creating the table. If a primary index is dropped later, the table becomes incomplete again, and you cannot perform any data operations on it until the primary index is recreated.

If you create the table first, and then modify the table later by adding the primary key with the ALTER TABLE statement, a unique index must already exist on the primary key columns.

Foreign key

A *foreign key* references a primary key in either the same or another table. The OFFICE column of the OFFICES table is an example of a primary key. The REP_OFFICE column of the ENGINEERS table is an example of a foreign key. The office value in the ENGINEERS table references the office value in the OFFICES table.

EMPL_ NUM	NAME	REP_ OFFICE
100	Paul Atkins	10
104	Bob Smith	20
107	Murray Rochester	30
102	Larry Sanchez	10
101	Sheila Brown	10
106	Sam Valdez	10

OFFICE	CITY	REGION
20	San Francisco	Western
40	New York	Eastern
10	Los Angeles	Western
30	Chicago	Midwest
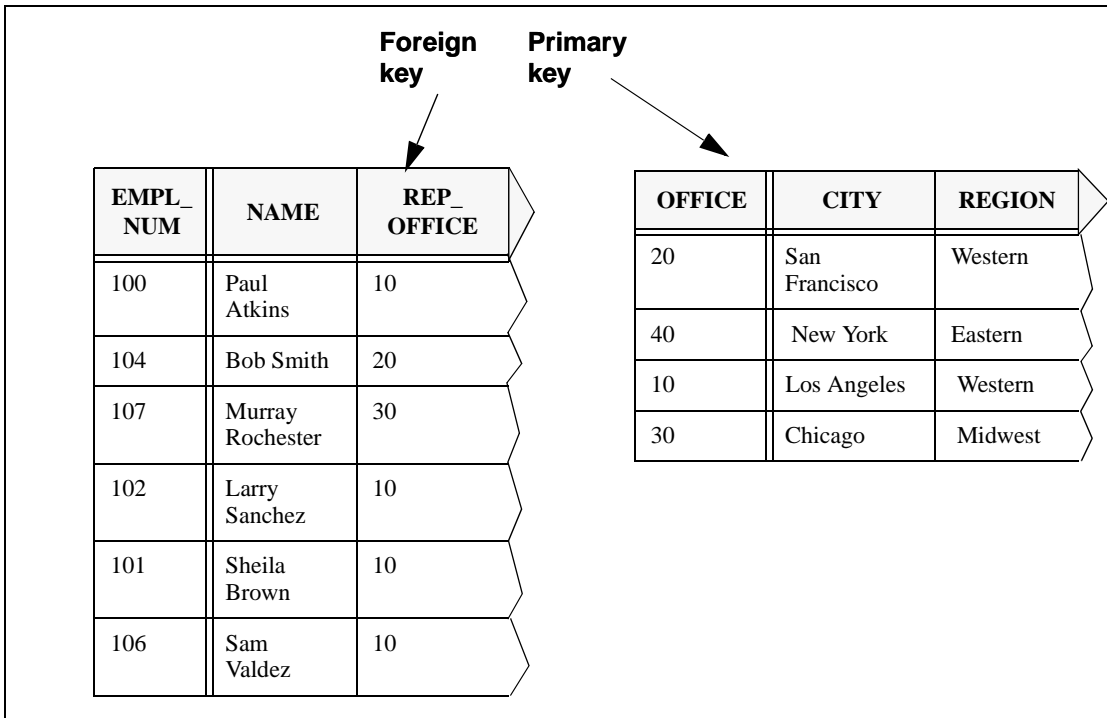
Example of a foreign key

Before creating a foreign key, you must first create both the primary key it references and also a unique index on that primary key.

Naming a foreign key

Each foreign key has a *constraint name*. This name identifies the foreign key. For example, a tokenized error message returns the constraint name when referencing a foreign key. The foreign key name is also required when you use the DROP FOREIGN KEY clause of the ALTER TABLE statement.

The constraint name is assigned when the foreign key is created (with CREATE TABLE or ALTER TABLE). You can assign the constraint name yourself; if you do

not, SQLBASE generates a constraint name from the name of the first foreign key column.

A foreign key constraint name can have up to eighteen characters. This means that if the first foreign key column name is more than eighteen characters, you must assign a constraint name yourself that does not violate this limit. Otherwise, SQLBase will not create the foreign key.

If there are multiple foreign keys referencing the same table, each foreign key must have a unique name. This ensures that every referential constraint is uniquely identified by a table name/constraint name combination.

For example, you could create a foreign key on the OFFICE.MGR column, and assign it a constraint name called HASMGR. If you do not assign the constraint name, SQLBase assigns MGR as a default.

Foreign key guidelines

In SQLBase, a foreign key must obey the following rules:

- *Matching columns.* A foreign key must contain the same number of columns as the primary key. The data types of the foreign key columns must match those of the primary key on a one-to-one basis, and the matching columns must be in the same order.

  However, the foreign key can have different column names and default values. It can also have NULL attributes. If an index is defined on the foreign key columns, the index columns can be in ascending or descending order, which may be different from the order of the primary key index.

- *Using primary key columns.* A column can belong to both a primary and foreign key.

- *Foreign keys per table.* A table can have any number of foreign keys.

- *Number of foreign keys.* A column can belong to more than one foreign key.

- *Number of columns.* A foreign key cannot contain more than 16 columns.

- *Parent table.* A foreign key can only reference a primary key in its parent table. This parent table must reside in the same database as the foreign key.

- *NULL values.* A foreign key column value can be NULL. A foreign key value is NULL if any column in the foreign key is NULL. See the following subsection on *Foreign keys and NULL values* for more information.

- *Privileges.* You must grant ALTER authority on a table to all users who need to define that table as the parent of a foreign key.

- *System catalog table.* The foreign key cannot reference a system catalog table.

- *Views.* A foreign key cannot reference a view.

- *Self-referencing row.* In a self-referencing row, the foreign key value can only be updated if it references a valid primary key value. See the following section on *Self-referencing tables* for more information.

Foreign key indexes

SQLBase does not require an index on a foreign key, but an index can increase database performance. A join with primary and foreign keys is fairly common, and creating an index on the foreign key can improve the performance of these joins.

SQLBase optimizes index checks by considering any index where a left-anchored partial key matches the dependent key. In particular, this method of index checking affects those referential integrity rules that involve locating dependent rows given for its parent key. Specifically,

- DELETE CASCADE (where dependent rows are located and deleted).

- DELETE SET NULL (where dependent rows are located and set to NULL)

- DELETE RESTRICT (where dependent rows are located and if any are found, deletion of the parent key is denied)

The following example shows you how SQLBase optimizes index checks.

Example:

Assume a parent table PT has a composite key (A, B) and a dependent table DT has a dependent composite key (X,Y). The dependency rule between PT and DT is a DELETE CASCADE, which means that when a row in PT is deleted, the corresponding dependent rows in DB are also deleted.

In order to locate the dependent rows in DT, SQLBase checks if an index on DT can be used. SQLBase not only considers an index on columns (X, Y) of DT, but also considers indexes defined on (X, Y, Z), (X, Y, A, B, C), etc. The closest matching index is chosen to enforce the referential integrity rule.

Foreign keys and NULL values

A foreign key column can have a NULL value, unlike a primary key column. Even though a NULL value does not match any value in a primary key, it satisfies the referential integrity constraint. This is also true for a multiple-column foreign key that contains part NULL/non-NULL values; SQLBase regards a foreign key value as NULL if any of its column values is NULL.

It is strongly recommended that you do not allow a foreign key to have partial NULL/ non-NULL values. Either all of the foreign key columns should allow NULL values, or none at all.

The following example with the PRODUCTS and SERV_CALLS tables demonstrate the problems with partial NULL foreign keys.

Example:

The composite key MFR_ID/PRODUCT_ID is a primary key in the PRODUCTS table. The composite key MFR/PRODUCT is a foreign key in the SERV_CALLS table referencing the PRODUCTS table.

Assume that the SERV_CALLS table allowed NULL values for the PRODUCT column. This means that you can enter a non-NULL value for the SERV_CALL.MFR column, and a NULL value in the SERV_CALL.PRODUCT column.

```
INSERT INTO SERV_CALLS VALUES (8000,
    9-4-93,2000,103,'WRS',NULL);
```

As a result, the row contains a foreign key value that does not match any primary key value in the PRODUCTS table.

CALL_NUM	CALL_DATE	CUST	REP	MFR	PRODUCT
8000	1993-00-04	2000	103	WRS	

Primary key   Foreign key

MFR_ID	PRODUCT_ID	DESCRIPTION
WRS	24c	Widget 1
WRS	25a	Widget 2

Example of partial NULL/non-NULL foreign key

SET NULL delete rule. The same situation applies with a SET NULL delete rule. With this rule, deleting a row from the PRODUCTS table sets the SERV_CALLS.PRODUCT column to NULL since it accepts a NULL value. Again, the row in the SERV_CALLS table does not match the PRODUCTS table. For more information on SET NULL, read *DELETE implications* on page *6-26*.

INSERT statement. With regards to referential integrity, SQLBase regards a row with partial NULL/non-NULL values as NULL. Once a row is defined as NULL, SQLBase does not perform any referential checks on it when you issue an INSERT statement.This means that SQLBase does not check the values in the foreign key's non-NULL columns to see if they match any values in the parent table.

Parent and child tables

Together, the primary key and foreign key create a *parent/child* relationship. The table containing the primary key is the *parent table*, while the table containing the foreign key is a *child table*. A child of a child is called a *descendent*.

In the following example, the PRODUCTS table is a parent of the SERV_CALLS table.

MFR_ID	PRODUCT_ID	DESCRIPTION
ACR	101	Tripod
ACR	102	Tripod2
MRP	101	Long Angle Lens
LMA	4211	Automatic Camera
LMA	4310	Regular Focus 1
LMA	4516	Regular Focus 2
MRP	600	lens
MRP	601	Shutter
WRS	24c	Widget 1
WRS	25a	Widget 2

Parent Table

PRODUCTS

Child Table

CALL_NUM	CALL_DATE	CUST	REP	MFR	PRODUCT
2133	1993-05-10	1000	102	ACR	102
6253	1993-05-02	3000	101	LMA	4516
7111	1993-05-09	1001	104	MRP	101
4250	1993-05-14	1050	109	MRP	101

Example of parent/child tables

To be a parent table, a table must have a primary key and primary index.

Some tables have no parent or child tables. These are called *independent tables.* Think carefully before using an independent table in your database design. Any reference to this table is neither validated nor verified.

Parent and child rows

A row belonging to a parent table that is referred to by a row belonging to the child table is a *parent row.* The row that refers to it is a *child row.* The child row must have at least one foreign key column value that is not NULL.

MFR_ID	PRODUCT_ID	DESCRIPTION
ACR	**102**	Tripod2

parent row

child row

CALL_NUM	CALL_DATE	CUST	REP	MFR	PRODUCT
2133	1993-05-10	1000	102	ACR	102

Example of parent/child rows with the PRODUCTS and SERV_CALLS tables
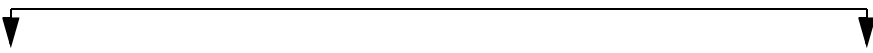
Not every row in a parent table is necessarily a parent row; it may not have any child rows that reference it. For example, on the previous page, the row in the PRODUCTS table whose description is Automatic Camera is not referenced by any of the rows in the SERV_CALL table.

Likewise, if a row in a child table has a NULL foreign key, it is not a child row.

Self-referencing tables and rows

A table can be a child of itself. This is called a *self-referencing table*. A self-referencing table contains both a foreign and primary key with matching values within the same table.

An example of a self-referencing table is the ENGINEERS table, where the foreign key MGR (MANAGER) references the primary key EMPL_NUM.

EMPL_NUM	NAME	REP_OFFICE	TITLE	HIRE DATE	MANAGER
100	Paul Atkins	10	Manager	1988-02-12	
104	Bob Smith	20	Sen. Engineer	1992-09-05	103
107	Murray Rochester	30	Sen. Engineer	1991-01-25	106
102	Larry Sanchez	10	Sen. Engineer	1989-06-12	100

If a row is a *self-referencing row,* its foreign key value is the same as its primary key value. This section does not show a self-referencing row.

The following restrictions apply to self-referencing tables and rows:

- The DELETE rule must be CASCADE.
- An INSERT statement with a subquery can only insert one row into a self-referencing table.
- You cannot use a DELETE WHERE CURRENT OF statement.
- To update the primary key, you must use one of the following methods:
  - Delete the row, and then reinsert it with the new primary and foreign key values

  OR

  - Update the foreign key value to another value or NULL (if permitted), and then update the primary key value.
- You can only update the foreign key in a self-referencing row if it references a valid primary key.

Delete-connected tables

Tables are *delete-connected* if deleting a row in one table affects the other table. For example, deleting an office from the OFFICES table affects the ENGINEERS table since each engineer is associated with an office.

Any table that is involved in a delete operation is delete-connected.

The following definitions apply to delete-connected tables:

- A self-referencing table is delete-connected to itself.

- A child table is always delete-connected to its parent table no matter what DELETE rule you specify.

- A table is delete-connected to its grandparent and great-grandparent tables when the delete rule between the parent and grandparent, or the grandparent and the great-grandparent, is CASCADE. The following figure illustrates this concept.
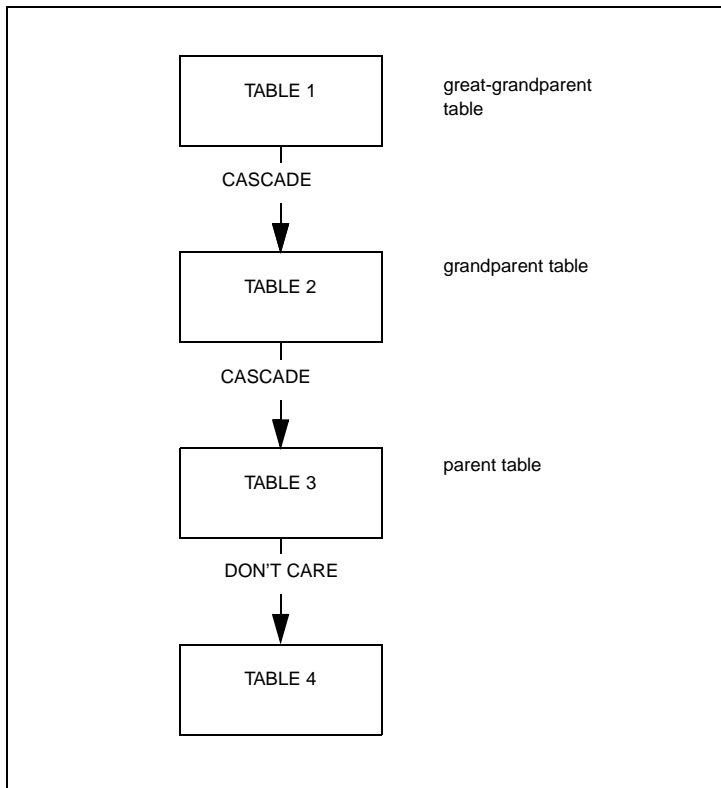
Delete-connected tables

In this figure, TABLE 4 is delete-connected to its grandparent table, TABLE 2, since the delete rule between TABLE 2 and TABLE 3 is CASCADE. TABLE 4 is also delete-connected to its great-grandparent table, TABLE 1, since the delete rule between TABLE 1 and TABLE 2 is CASCADE. The delete rules between TABLE 4 and its parent, TABLE 3, do not affect these delete-connections.

For information on restrictions for delete-connected tables, read the section *Delete-connected table restrictions on page 6-27*.

How to create tables with referential constraints

Use the CREATE TABLE or ALTER TABLE statement to create or alter tables with primary keys or foreign keys, and to establish referential constraints.

Using the CREATE TABLE statement

In the following example, the CREATE TABLE command creates the ENGINEERS table with a primary and foreign key:

```
CREATE TABLE ENGINEERS
  (EMPL_NUM INTEGER NOT NULL,
  NAME VARCHAR(24) NOT NULL,
  REP_OFFICE INTEGER,
  TITLE VARCHAR(15),
  HIRE_DATE DATE NOT NULL,
  MANAGER INTEGER,
  PRIMARY KEY (EMPL_NUM),
  FOREIGN KEY WORKSIN (REP_OFFICE)
  REFERENCES OFFICES ON DELETE RESTRICT);
```

Issues for primary key

As a general rule, you should specify the primary key when you create the table with the CREATE TABLE statement, rather than adding the key later with the ALTER TABLE statement.

Remember to create a unique index on the table after creating the primary key, or the table will be incomplete. Read the section on *Creating a primary index* on page *6-16*.

Issues for foreign key

You can use CREATE TABLE to create a foreign key while creating the table. Remember, however, that the foreign key must reference a table with an existing primary key and primary index. In the example above, the foreign key WORKSIN references the OFFICES table. The OFFICES table must already have an existing primary key, and a primary index created on the primary columns.

To create a foreign key, you must have the ALTER privilege on both the table containing the foreign key and the table containing the primary key.

When you create a foreign key, you can also specify a DELETE rule for the foreign key. If you do not specify the DELETE rule yourself, SQLBase assigns a default DELETE rule of RESTRICT. Read the section *DELETE implications* on page *6-23* for more information.

You cannot specify an UPDATE rule.

Using the ALTER TABLE statement

You can use the ALTER TABLE statement to create a primary and foreign key after you create the tables. Since the parent table must exist, some foreign key constraints can only be defined with the ALTER TABLE statement, such as a self-reference.

To add the foreign key ISFOR to the SERV_CALLS table after creating the table with CREATE TABLE, use this ALTER TABLE command:

```
ALTER TABLE SERV_CALLS FOREIGN KEY ISFOR
    (MFR,PRODUCT) REFERENCES PRODUCTS ON DELETE
    RESTRICT;
```

Before using ALTER TABLE to add a primary key, you must create a unique index on the primary key columns.

Creating a primary index

Since a table is incomplete until you create a primary index, create the index soon after creating the table. For example, to create the primary index for the OFFICES table, enter the following command:

```
CREATE UNIQUE INDEX OFFICE_IDX ON OFFICES (OFFICE);
```

If you add the primary key later with ALTER TABLE, a unique index must already exist on the primary key columns.

If you are loading database information with the LOAD command, you should create the index after the load for performance reasons.

Reporting referential integrity

There are three SQLBase system catalog tables that contain referential integrity information. For a description of the columns in these tables, read the appendix on *System catalog tables*.

- SYSADM.SYSFKCONSTRAINTS (Foreign key constraints)

  This table contains information about a table's foreign keys, such as the constraint name, column(s) of the foreign key, and the parent table it references.

  ```
  SELECT * FROM SYSFKCONSTRAINTS
      WHERE NAME='SERV_CALLS';
  ```

CREATOR	NAME	CON-STRAINT	FKCOLS EQNUM	REFS COLUMN	REFDTB CREATOR	REFDTB NAME	REFD COLUMN
SYSADM	SERV_CALLS	ISFOR	1	MFR	SYSADM	PRODUCTS	MFR_ID
SYSADM	SERV_CALLS	ISFOR	2	PRODUCT	SYSADM	PRODUCTS	PRODUCT_ID

SYSADM.SYSFKCONSTRAINTS table

- SYSADM.SYSPKCONSTRAINTS (Primary key constraints)

  This table contains information about a table's primary key columns, such as the column name of the primary key and the table name.

  ```
  SELECT * FROM SYSPKCONSTRAINTS WHERE
      NAME='PRODUCTS';
  ```

CREATOR	NAME	PKCOLSEQNUM	COLNAME
SYSADM	PRODUCTS	1	MFR_ID
SYSADM	PRODUCTS	2	PRODUCT_ID

SYSADM.SYSPKCONSTRAINTS table

- SYSADM.SYSTABCONSTRAINTS (Table constraints)

  This table contains information about all constraints pertaining to a specific table, such as the name and type of constraint (primary or foreign key), delete rule for a foreign key, and any customized user error messages (read the section *Customizing SQLBase error messages* on page *6-30* for more information).

  ```
  SELECT * FROM SYSTABCONSTRAINTS WHERE
    NAME='SERV_CALLS';
  ```

CREATOR	NAME	CONSTRAINT	TYPE	DELETE RULE	USRERR INSDEP	USRERR UPDDEP	USRERR DELPAR	USRERR UPDPAR
SYSADM	SERV_CALLS	ISFOR	F	R	0	0	0	0
SYSADM	SERV_CALLS	PRIMARY	P	0	0	0	0	

SYSADM.SYSTABCONSTRAINTS table

Implications for SQLBase operations

Referential constraints have special implications for some SQLBase operations. This section describes how referential integrity affects the SQLBase INSERT, UPDATE, DROP, SELECT, and DELETE commands.

Views share the referential constraints of their base tables.

INSERT

SQLBase enforces the following rules when you insert data into a table with one or more foreign keys:

- Each non-null value you insert into a foreign key column must match a value in the primary key.

- If any column in the foreign key is null, SQLBase regards the entire foreign key as null. SQLBase does not perform any referential checks on an INSERT statement with a NULL foreign key.

- You cannot insert values into a parent or child table if the parent table is no longer complete (for example, if you dropped the primary index).

You can insert data into the parent table at any time without it affecting the child table. For example, adding a new office to the OFFICES table does not affect the ENGINEERS table.

UPDATE

If you are updating a child table, every non-NULL foreign key value that you enter must match a valid primary key value in the parent table. If the child table references multiple parent tables, the foreign key values must all reference valid primary keys.

The only UPDATE rule that can be applied to a parent table is RESTRICT. This means that any attempt to update the primary key of the parent table is restricted to cases where there are no matching values in the child tables.

SQLBase enforces the following rules on an UPDATE statement:

- An UPDATE statement that assigns a value to a primary key *cannot* specify more than one record.

- An UPDATE statement with a WHERE CURRENT OF clause cannot update a primary key, or columns of a view derived from a primary key.

DELETE

You can specify a delete rule for each parent/child relationship created by a foreign key in a SQLBase application. The delete rule tells SQLBase what to do when a user tries to delete a row from the parent table. You can specify one of three delete rules:

- RESTRICT
- CASCADE
- SET NULL

If you execute a DELETE statement against a table, you cannot specify a subquery that references the same table. For an example of this rule, see the section *Delete-connected table restrictions on page 6-27*.

DELETE RESTRICT

This rule prevents you from deleting a row from the parent table if the row has any child rows. You can delete a row if there are no child rows.

For the sample service database, a DELETE RESTRICT rule is appropriate for the relationship between a service call and the product that is serviced. You should not be able to delete product information from the database if there are still open service calls against the product.

```
ALTER TABLE SERV_CALLS FOREIGN KEY
    (MFR,PRODUCT) REFERENCES SERV_CALLS ON DELETE
    RESTRICT;
```

If you do not specify a DELETE rule, RESTRICT is the default, since it has the least potential for damage.

DELETE CASCADE

This rule specifies that when a parent row is deleted, all of its associated child rows are automatically deleted from the child table(s). Deletions from the parent table *cascade* to the child table. If any part of the delete fails, the whole delete operation fails. The delete is also propagated to descendent tables.

A DELETE CASCADE rule is appropriate for the relationship between a service call and the customer who is being serviced. You probably delete a customer row from the database only if the customer is inactive or ends its relationship with the company; in this case, all of the customer's service calls should also be deleted.

```
ALTER TABLE SERV_CALLS FOREIGN KEY (CUST)
    REFERENCES CUSTOMERS ON DELETE CASCADE;
```

Be careful using the CASCADE rule, since it can delete an extensive amount of data if it is used incorrectly.

DELETE CASCADE does not delete a parent row if a child or descendent row has a DELETE RESTRICT rule.

For a self-referencing table, CASCADE is the only DELETE rule allowed.

DELETE SET NULL

This rule specifies that when a parent row is deleted, the foreign key values in all of its child rows should automatically be set to NULL.

If an engineer leaves the company, any customers serviced by that engineer become the responsibility of an unknown engineer until they are reassigned.

```
ALTER TABLE CUSTOMERS FOREIGN KEY HASREP
    (SERV_REP) REFERENCES ENGINEERS ON DELETE SET
    NULL;
```

For a foreign key, you can use the SET NULL option only if at least one of the columns of the foreign key allows NULL values. The default is RESTRICT.

DROP

Dropping a table drops both its primary key and any foreign keys. When you drop a parent table or its primary key, the referential constraint is also dropped.

Before you drop a primary or foreign key, consider the effect this will have on your application programs.   Dropping a key drops the corresponding referential relationship. It also drops the DELETE rule for a foreign key. In addition, the primary key of a table is a permanent, unique identifier of the entities it describes, and some of your programs might depend on it. Without a primary or foreign key, your programs must enforce these referential constraints.

Note that dropping a primary or foreign key is not the same as deleting its value.

Use the ALTER TABLE statement to drop a primary or foreign key.

Dropping a primary key

If you have ALTER privilege on both the parent and child tables, you can drop a primary key. The following example drops a primary key:

```
ALTER TABLE OFFICES DROP PRIMARY KEY;
```

This statement drops the primary key of the OFFICES table. It also drops the parent/ child relationship with the ENGINEERS table.

If a user has ALTER privilege on a table, you cannot revoke this privilege if he has already created a foreign key that references that table.

Dropping a primary key does not drop the primary index. The index remains a unique index on the former primary key's columns.

Dropping a primary index

Dropping a primary index results in an incomplete table. To create a complete table definition, create another unique index on the columns of the primary key.

Referential constraints remain even if you drop the primary index.

Dropping a foreign key

The following SQL statement drops the foreign key ISFOR from the SERV_CALLS table:

```
ALTER TABLE SERV_CALLS
DROP FOREIGN KEY ISFOR;
```

To drop a foreign key, you must have ALTER privilege on both the parent and dependent tables.

SELECT

Because a SELECT statement does not change actual data values, it is not affected by referential integrity.

Cycles of dependent tables

In the sample service database, the ENGINEERS table contains the REP_OFFICE column, which references the OFFICES.OFFICE column. The OFFICES table also contains a foreign key on the MGR column, which references the ENGINEERS.EMPL_NUM column.

EMPL_ NUM	NAME	REP_ OFFICE
100	Paul Atkins	10
104	Bob Smith	20
107	Murray Rochester	30
102	Larry Sanchez	10
101	Sheila Brown	10
106	Sam Valdez	30

Engineers table

OFFICE	CITY	MGR
20	San Francisco	103
40	New York	108
10	Los Angeles	100
30	Chicago	106
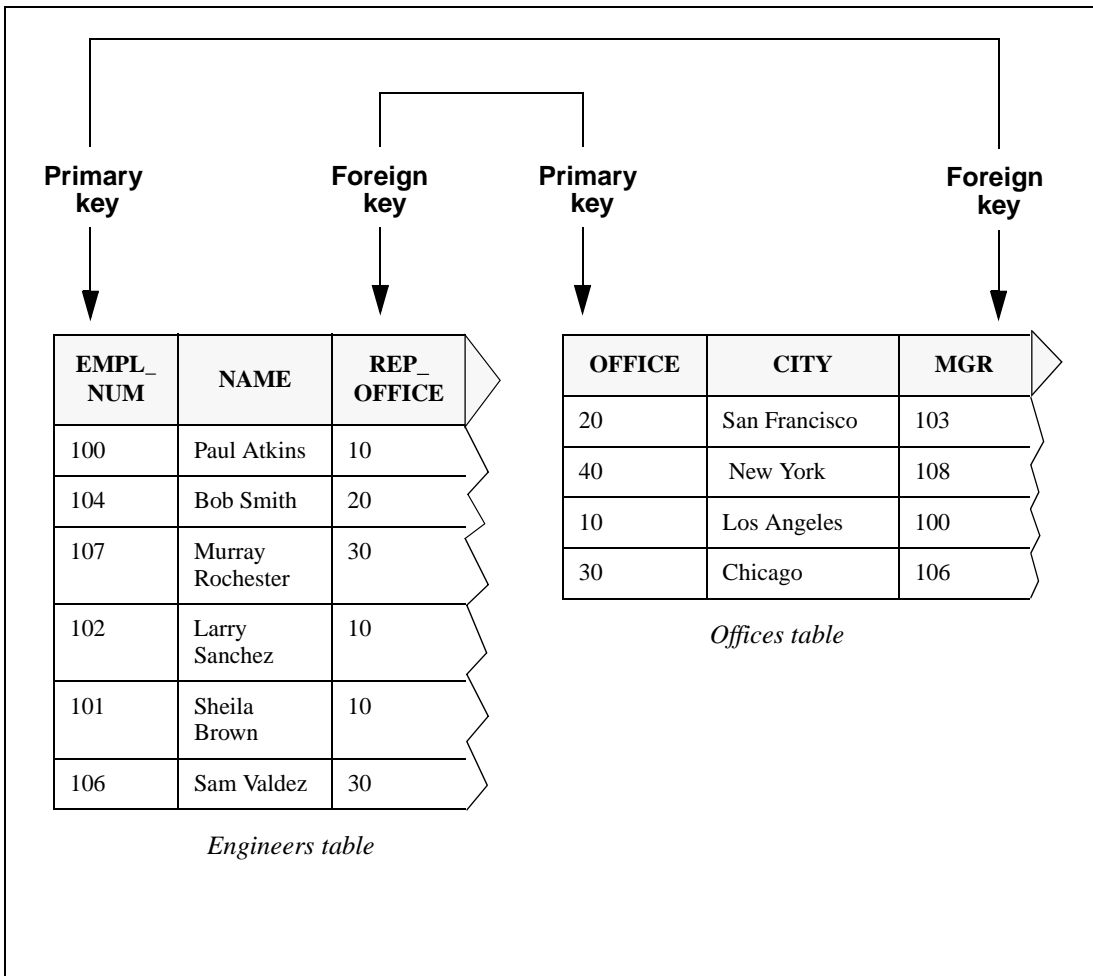
Offices table

Example of a referential cycle

Both tables have a foreign key that reference each other's primary key. These two relationships form a *referential cycle*. This means that any given row in the ENGINEERS table references a row in the OFFICES table, which refers to a row in the ENGINEERS table, and so on. This example shows a cycle of two tables, but you can create cycles with more tables.

INSERT implications

This kind of cyclical relationship can cause problems for an INSERT statement. For example, assume you have just hired a new senior engineer, Ronald Casey (employee 112) who will be managing a new office in Boston (office 50)

```
INSERT INTO ENGINEERS (EMPL_NUM, NAME,
    REP_OFFICE, TITLE, HIRE_DATE) VALUES
    (112,'Ronald Casey', 50,'Manager',8-15-93);

INSERT INTO OFFICES VALUES
    (50,'Boston','Eastern',112, NULL);
```

The first insert into the ENGINEERS table fails, because it refers to office 50, which does not exist yet. Reversing the statements does not help either, since manager 112 does not exist yet.

To avoid this insert dilemma, at least one of the foreign keys in a referential cycle must permit NULL values. You can then accomplish the two-row insertion with two INSERT and one UPDATE statements:

```
INSERT INTO ENGINEERS VALUES (112,'Ronald Casey',
    NULL,'Manager',8-15-93,NULL);

INSERT INTO OFFICES VALUES
    (50,'Boston','Eastern',112,NULL);

UPDATE ENGINEERS
SET REP_OFFICE=50
WHERE EMPL_NUM=112;
```

DELETE implications

Referential cycles can also cause problems for a DELETE operation. To illustrate this, this section uses the following three tables:

- OFFICES
- CUSTOMERS
- ENGINEERS

These three tables have a referential cycle relationship. The CUSTOMERS table is a parent of the OFFICES table, OFFICES is a parent of ENGINEERS, and ENGINEERS is a parent of CUSTOMERS.

The following three examples demonstrate what happens if you delete a row in the CUSTOMERS table with different DELETE rules.

The following diagram shows the relationships between the tables if you create each foreign key with the DELETE CASCADE rule.
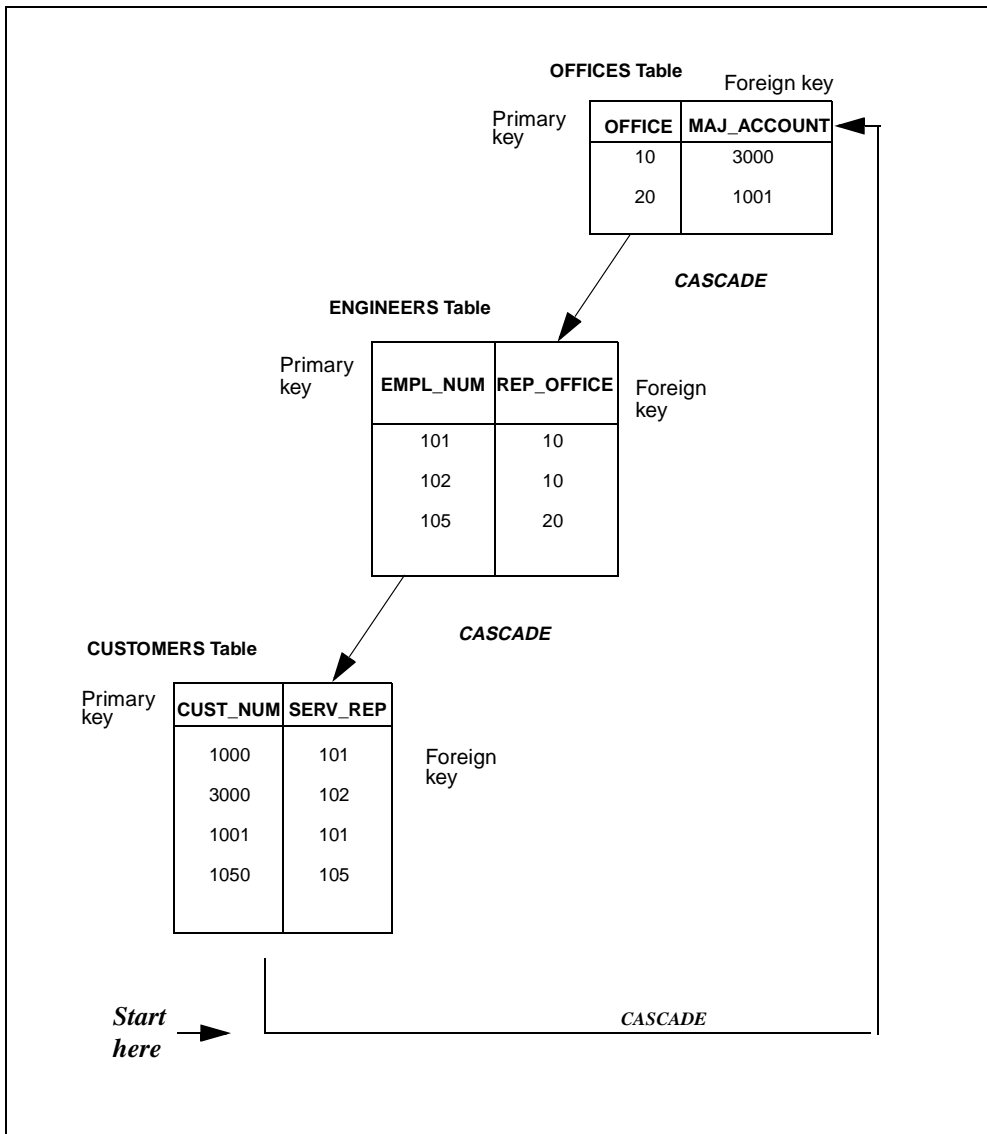
Referential cycles with DELETE CASCADE

Using the CASCADE rule, the following delete cycle starts:

1. Delete customer 3000 from the CUSTOMER table.

2. This deletes office 10 from the OFFICES table.

3. This deletes engineers 101 and 102 from the ENGINEERS table.

4. This deletes customers 1000 and 1001 from the CUSTOMERS table.

5. This deletes office 20 from the OFFICES table.

6. This deletes engineer 105 from the ENGINEERS table, and so on.

To break this cycle of cascaded deletes, SQLBase has the following requirements:

- In a cycle with only two tables, neither delete rule can be CASCADE.
- In cycles of more than two tables, at least one of the delete rules must be RESTRICT or SET NULL.

These rules prevent a table from becoming delete-connected to itself.

The following diagram shows the relationships between the tables if you create each foreign key with the DELETE RESTRICT rule.
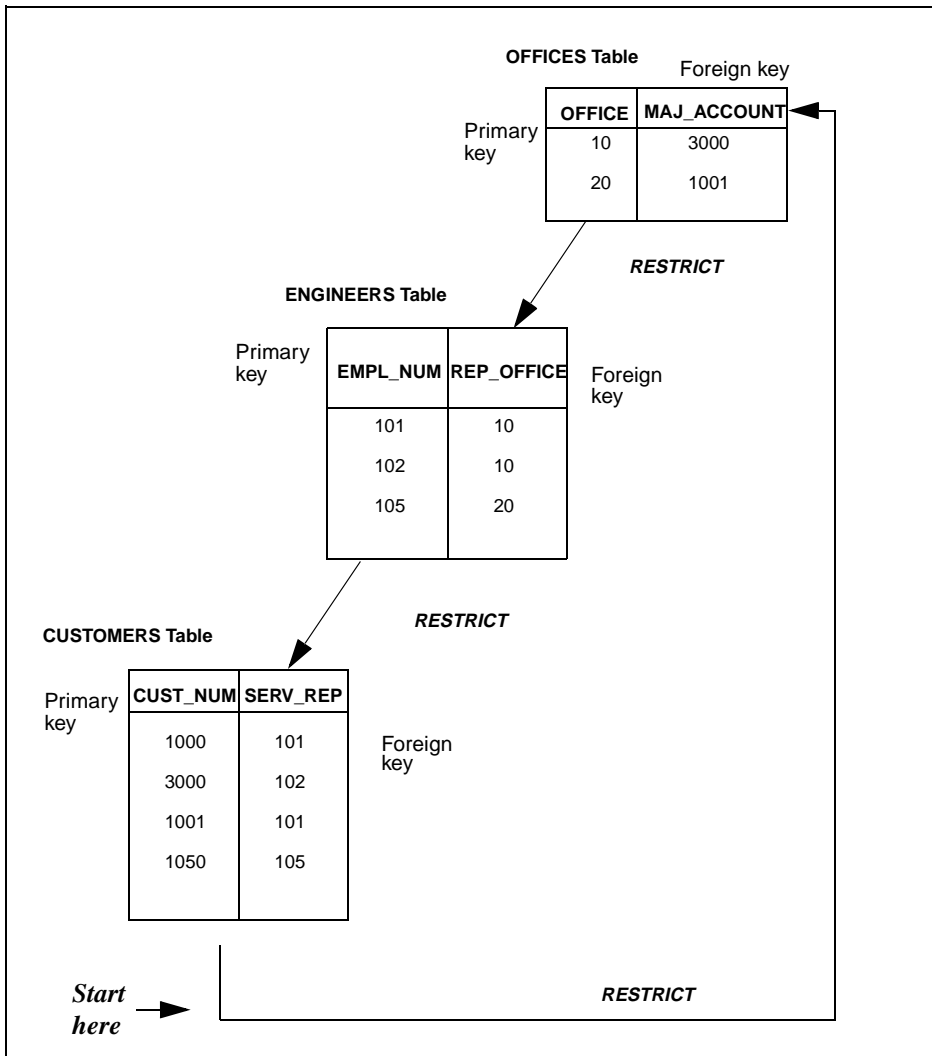
Referential cycles with DELETE RESTRICT

With this rule, you cannot delete any customers, since they are all parent rows in the other tables.

You should not specify the RESTRICT rule for all the relationships in a referential cycle, unless you want to prevent users from deleting any data.

Delete-connected table restrictions

The following restrictions apply to delete-connected tables.

- If a DELETE operation involves a table that is referenced in a subquery, the last delete rule in the path to that table must be RESTRICT.

  A basic rule of SQL is that the result of an operation must not depend on the order in which rows of a table are accessed. That means that a subquery of a DELETE statement cannot reference the same table that rows are deleted from.

  For example, if there were no referential constraints, you could insert this row into the OFFICES table:

  ```
  INSERT INTO OFFICES VALUES
      (15,'ANYTOWN','MIDWEST',333,NULL)
  ```

  Of course, this enters an office with a non-existing manager. With no referential constraints defined, you could delete this row. For example, you could delete all rows from the OFFICES table whose manager is not listed correctly in the ENGINEERS table.

  ```
  DELETE FROM OFFICES WHERE MGR NOT IN (SELECT
      EMPL_NUM FROM ENGINEERS);
  ```

  However, if you define a foreign key in the ENGINEERS table that referenced the OFFICES table, the subquery breaks the rule that it cannot reference the same table that rows are deleted from (the OFFICES table). The results of this command depends on the order in which rows are accessed. SQLBase forces this statement to fail with an error message.

- If two tables are delete-connected via two or more distinct referential paths, the paths (or last part of the path) must have the same delete rule, and it cannot be SET NULL.

  The following figures illustrates this rule. The first shows valid referential structures with delete-connected tables:
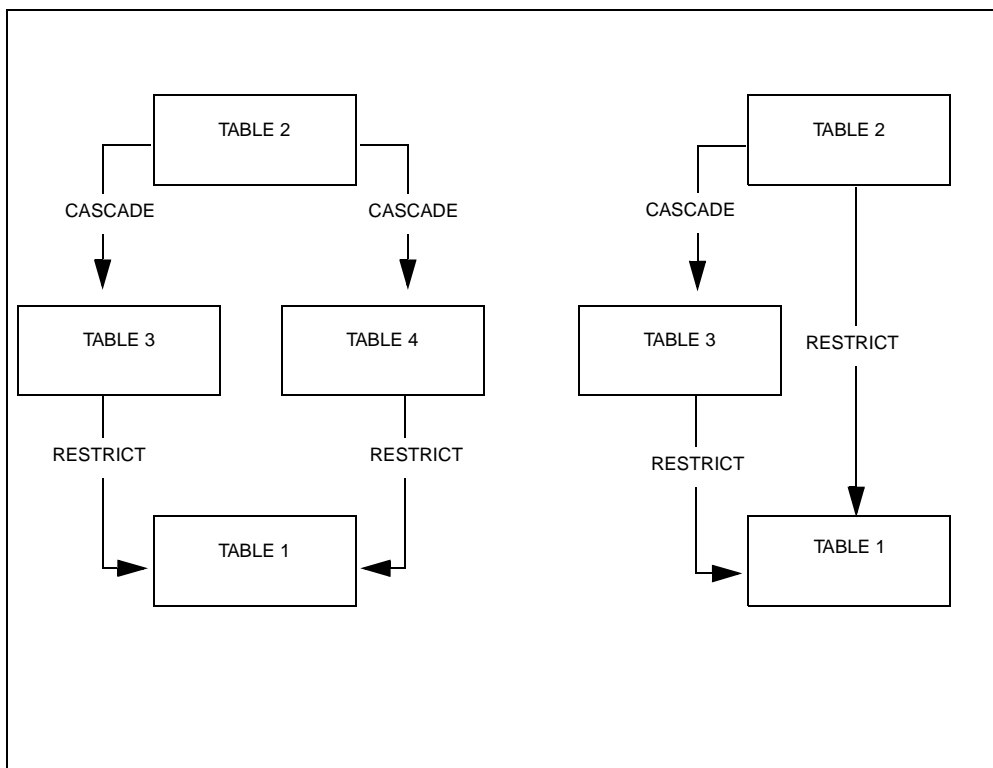
Valid delete-connected structures

In this figure, all the referential structures have valid delete-connections. In both structures, table 1 has identical delete rules on its relationships, and the last delete rule is not SET NULL.

The following figure shows invalid structures:
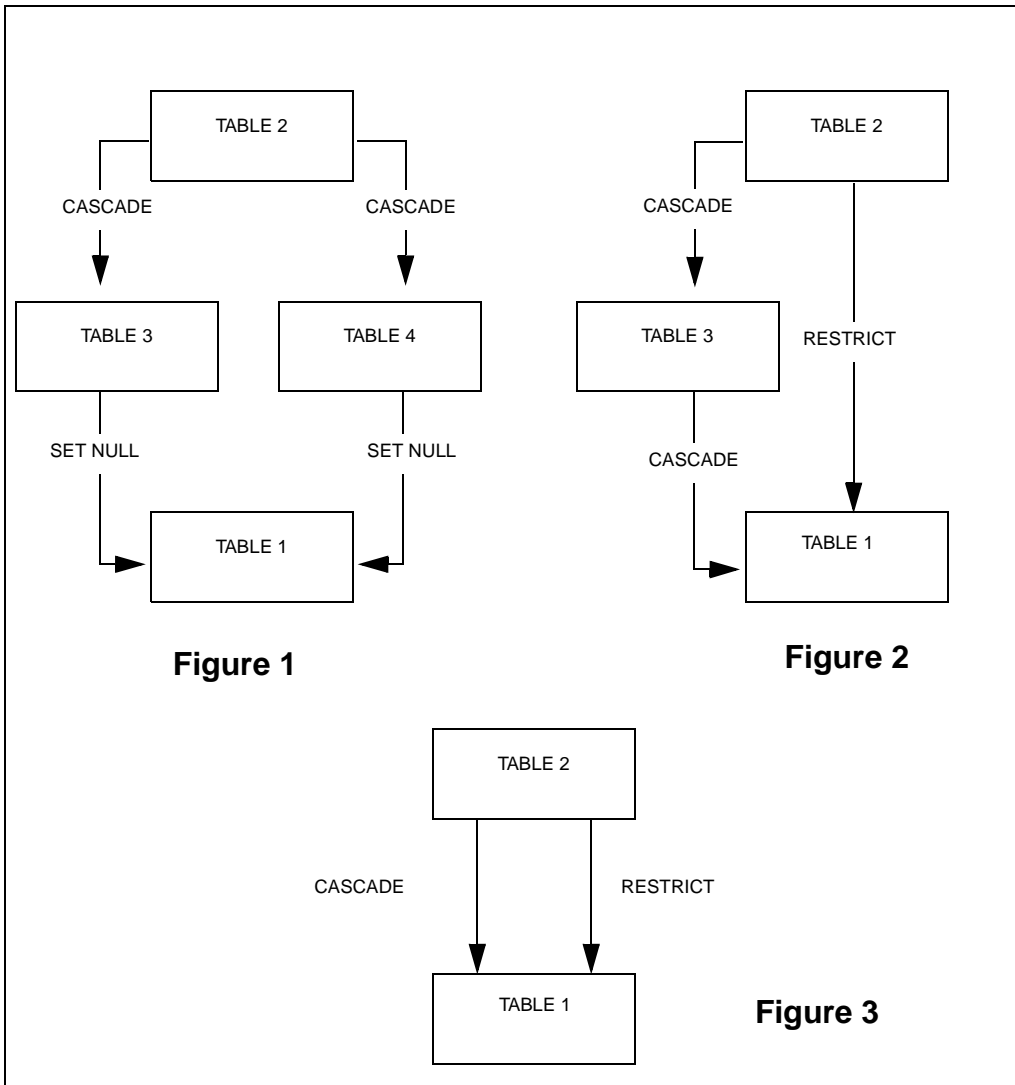
Invalid referential structures

In Figure 1, table 1 has identical rules of SET NULL. In Figure 2, the last two rules are not the same. In Figure 3, two tables are connected by two different types of delete rules.

The problem with the SET NULL rule was discussed in the earlier subsection on *Foreign keys and NULL values* in the *Components* section. Allowing SET NULL rules in multiple paths could result in partial NULL/non-NULL foreign keys. By only allowing CASCADE and RESTRICT, the child row is either deleted (CASCADE) or remains the same (RESTRICT).

SQLTalk commands and referential integrity

When running the following SQLTalk commands, keep in mind that SQLBase does not enforce referential integrity during their execution. This means that all your data must be valid before executing the commands.

SQLTalk command	Referential integrity impact
LOAD	SQLBase turns off all referential integrity checks before starting the load process, and turns the checks back on after the load.
CHECK DATABASE	Does not check if any tables were in the Pending state, or perform any other referential checks
REORGANIZE	SQLBase turns off all referential integrity checks before starting the reorganize process, and turns the checks back on after the reorganization.
COPY	SQLBase turns off all referential integrity checks before starting the copy process, and turns the checks back on after the copy.

Customizing SQLBase error messages

There are several error messages in SQLBase specific to referential integrity. This section shows how you can create new referential integrity error messages that are customized for certain tables.

Several default SQLBase messages appear when you violate referential integrity rules For example, the following message appears when an insert into a child table fails because there was no parent row in the parent table:

```
EXE UFV - unmatched foreign key values"
```

The following message appears when an update into a child table fails because there was no parent row in the parent table containing the new set of values:

```
"EXE UFV - unmatched foreign key values"
```

The following message appears when you attempt to delete a parent row that has associated child rows:

```
"EXE CDR - cannot delete row until all the dependent
  rows are deleted"
```

The following message appears when you attempt to update a parent row that has associated child rows:

```
"EXE CUR - cannot update row until all the dependent
  rows are deleted"
```

To make these messages more specific, you can create new customized messages by editing the error message file, *error.sql*

Editing the error messages

To customize the error messages for referential integrity, use the following steps:

1. Add the customized error message to the *error.sql* file.

2. Use the ALTER TABLE statement to associate the message with a particular operation on a specific primary or foreign key. The following diagram shows the syntax of this command to add, drop, or modify user-defined error messages for primary or foreign keys.
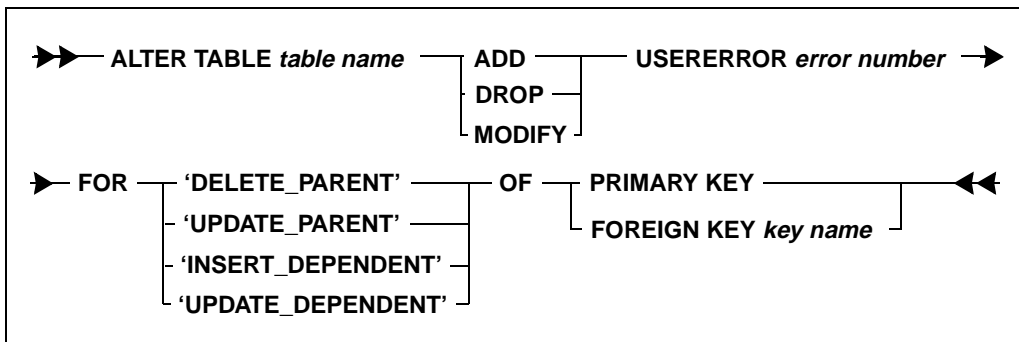
ALTER TABLE syntax

The USERERROR <error number> clause is the number associated with the message in error.sql. If you are dropping an error message (DROP), do not enter the error number with this clause.

You can create a customized error message for the following operations:

• deleting a parent row

• updating a parent row

- inserting a child row
- updating a child row

You can customize one error message each per parent/child and child/parent relationship. You can specify error messages for more than one child table if there are multiple child/parent relationship.

To demonstrate how to create customized error messages, this section uses the PRODUCTS and SERV_CALLS tables.

Primary key error messages

If a user attempts to delete a product from the PRODUCTS table that still has open service calls associated with it, the DELETE fails with the default error message.

```
DELETE FROM PRODUCTS WHERE MFR_ID='LMA'.
Error: EXE CDR - cannot delete row until all the dependent
       rows are deleted
```

This message is not very helpful since it is so general. To customize it, create a new message in the *error.sql* file:

```
20000 xxx xxx Product cannot be deleted while there are
still open service calls on it.
```

Then, use the ALTER TABLE statement to add the new message:

```
ALTER TABLE PRODUCTS ADD USERERROR 20000 FOR
    'DELETE_PARENT' OF PRIMARY KEY;
```

If a user now tries to delete a product that still has open service calls against it, the new message appears:

```
DELETE FROM PRODUCTS WHERE MFR_ID='LMA'.
Error: Product cannot be deleted while there are still open
       service calls on it.
```

Foreign key error messages

In the following example, if a user attempts to insert a new service call into the table that does not reference a valid product, the command fails with the following default error message:

```
INSERT INTO SERV_CALLS VALUES (2133,5-10-
    93,1000,102,'PRR',100,)
Error: EXE UFV unmatched foreign key values
```

To customize this message, create a new message in the *error.sql* file:

```
20001 xxx xxx Service call must reference a valid product
       number.
```

Then, use the ALTER TABLE statement to add the new error message:

```
ALTER TABLE SERV_CALLS ADD USERERROR 20001 FOR
    'INSERT_DEPENDENT' OF FOREIGN KEY ISFOR;
```

If a user now tries the same operation, the following error message appears:

```
INSERT INTO SERV_CALLS VALUES (2133,5-10-
    93,1000,102,'PRR',100);
Error: Service call must reference a valid product number.
```

Service database tables

This section shows the tables from the sample service database with their columns and values.

OFFICE	CITY	REGION	MGR	MAJ_ ACCOUNT
20	San Francisco	Western	103	1050
40	New York	Eastern	108	2500
10	Los Angeles	Western	100	3000
30	Chicago	Midwest	106	1001

OFFICES table

EMPL_ NUM	NAME	REP_ OFFICE	TITLE	HIRE DATE	MANAGER
100	Paul Atkins	10	Manager	1988-02-12	
104	Bob Smith	20	Sen. Engineer	1992-09-05	103
107	Murray Rochester	30	Sen. Engineer	1991-01-25	106
102	Larry Sanchez	10	Sen. Engineer	1989-06-12	100
101	Sheila Brown	10	Engineer	1990-10-10	100
106	Sam Valdez	30	Manager	1990-04-20	
105	Rob Jones	20	Engineer	1991-09-08	103
103	Anna Rice	20	Manager	1985-07-10	

EMPL_ NUM	NAME	REP_ OFFICE	TITLE	HIRE DATE	MANAGER
108	Mary Adams	40	Manager	1988-08-10	
109	Nancy Bonet	40	Sen. Engineer	1989-11-12	108
110	Richard Park	40	Engineer	1990-11-14	108
111	Dan Chester	40	Engineer	1987-03-22	111

ENGINEERS table

CUST_NUM	COMPANY	SERV_ REP	CREDIT_ LIMIT
1000	Acme Camera	101	5000
2500	Photo-1 Shop	110	3000
1001	Best Photography	106	1000
1050	Johnson's Camera Company	105	8050
2000	Sue's Family Photo	103	5000
3000	1-Hour Quick Photo	102	3000

CUSTOMERS table

CALL_NUM	CALL_DATE	CUST	REP	MFR	PRODUCT
2133	1994-05-10	1000	101	ACR	102
6253	1994-05-02	3000	102	LMA	4516
7111	1994-05-09	1001	106	MRP	600
4250	1994-05-14	1050	105	MRP	600

SERV_CALLS table

MFR_ID	PRODUCT_ID	DESCRIPTION
ACR	101	Tripod
ACR	102	Tripod2
MRP	101	Long Angle Lens
LMA	4211	Automatic Camera
LMA	4310	Regular Focus 1
LMA	4516	Regular Focus 2
MRP	600	Lens
MRP	601	Shutter
WRS	24c	Widget 1
WRS	25a	Widget 2

PRODUCTS table

Chapter 7

Procedures and Triggers

This chapter describes procedures and provides you with the information necessary to create procedures of your own. It covers the following topics:

- What is a procedure?
- Format of a procedure
- Data types supported in procedures
- System constants supported in procedures
- How to generate, store, and execute procedures
- Using SAL functions in procedures
- Error handling
- Procedure examples (contained in the \Centura\sp.sql directory)
- Triggers

What is a procedure?

A SQLBase procedure is a set of Scalable Application Language (SAL) and SQL statements that is assigned a name, compiled, and optionally stored in a SQLBase database.

SQLBase procedures can be static or dynamic. *Static procedures* must be stored (at which time they are parsed and precompiled) before they are executed. *Dynamic procedures* contain dynamic embedded SQL statements, which are parsed and compiled at execution time. For this reason, they do not have to be stored before they are executed.

There are several different types of procedure implementations:

- *Stored procedures:* compiled and stored in the database for later execution. They can be static or dynamic. You can define triggers on stored procedures.

- *Non-stored procedures:* compiled for immediate execution.

- *Inline procedures:* used optionally in triggers. You may want to specify the INLINE clause of the CREATE TRIGGER command to call inline procedure text. When you create the trigger, SQLBase stores these inline procedures in the system catalog.

SQLBase's implementation of procedures will be familiar to anyone already using Centura Team Developer, a graphical application development system. SQLBase provides a set of SAL functions that you can embed in procedures, and the flow control language of procedures is the same as Team Developer programs. However, you do *not* need the Team Developer product to use these functions; they are provided by SQLBase.

SQLBase also provides preconstructed procedures as useful tools to help you maintain your database. See *Appendix B* of the *Database Administrator's Guide* for a description of SQLBase-supplied procedures.

Why use procedures?

Procedures offer a number of benefits:

- They simplify applications by transferring processing to the server.

- They reduce network traffic by storing the SQL statements to be executed on the backend where the procedures are processed. The frontend need only call the procedure and wait for results.

- They provide more flexible security, giving end-users privileges on data which they might not otherwise be allowed to access.

Storing procedures provide these additional benefits:

- They improve runtime performance because the procedural logic is precompiled. In the case of static stored procedures, the SQL statements are also precompiled; as a result, the SQL execution plans are predetermined.

- You have a centralized location of precompiled programs, which different sites can then access for their own customized applications. This facilitates control and administration of database applications.

- You can store a procedure and then retrieve and execute this procedure from a variety of front-ends, such as SQLTalk, Team Developer, or a SQL/API application.

- You can invoke an external function within a stored procedure, providing you with the flexibility to extend the functionality of your stored procedures, or add functionality to your existing applications by creating plug and play external components. Read *Chapter 9, External Functions* for details.

When used in conjunction with triggers, procedures also can implement business rules that are not possible from the database server through SQL declarative referential integrity. For examples and more information on triggers, read *Triggers* on page *7-54*.

How stored procedures are different from stored commands

SQLBase already allows you to store often-used SQL statements in *stored commands* for future execution. However, a stored command can only contain a single SQL statement. Procedures, on the other hand, allow you to create a program using procedural logic, data typing, and variables using multiple SQL statements.

Unlike stored commands, stored procedures themselves never become invalid, although the stored commands within procedures may become invalid. This means you do not need to automatically recompile the procedure with EXECUTE RECOMPILE, or flag it to be recompiled with ALTER COMMAND.

Note: When using procedures with Team Developer programs, be aware that there are some implementation issues you must address. These issues are discussed in the section *Using procedures with Centura Team Developer applications* on page *7-43*.

Format of a procedure

SQLBase procedures follow a format and syntax similar to a Team Developer program. A SQLBase procedure has the following elements:

- *Name.* This is the name of the procedure, which can be different from the name under which you store the procedure.

- *Parameters.* You can define parameters for input and output to the procedure.

- *Local Variables.* You can define local variables for temporary storage.

- *Action section.* Use this section to control both the conditions under which the statements are executed and the order in which they are executed.

Unlike Team Developer, the elements of SQLBase procedures are case insensitive.

```
Name          ━━━▶    PROCEDURE: myproc
Parameters    ━━━▶    Parameters
                          Number: nInputVar1
                          Number: nInputVar2
                          Number: nInputVar3
                          Receive Number: nOutputVar1
                          Receive Number: nOutputVar2
Local Variables ━━━▶  Local Variables
                          Sql Handle: hSqlCur1
                          Sql Handle: hSqlCur2
                          Number: nInd
Actions       ━━━▶    Actions
                          Call SqlConnect (hSqlCur1)
                                    .
                                    .
                                    .
```
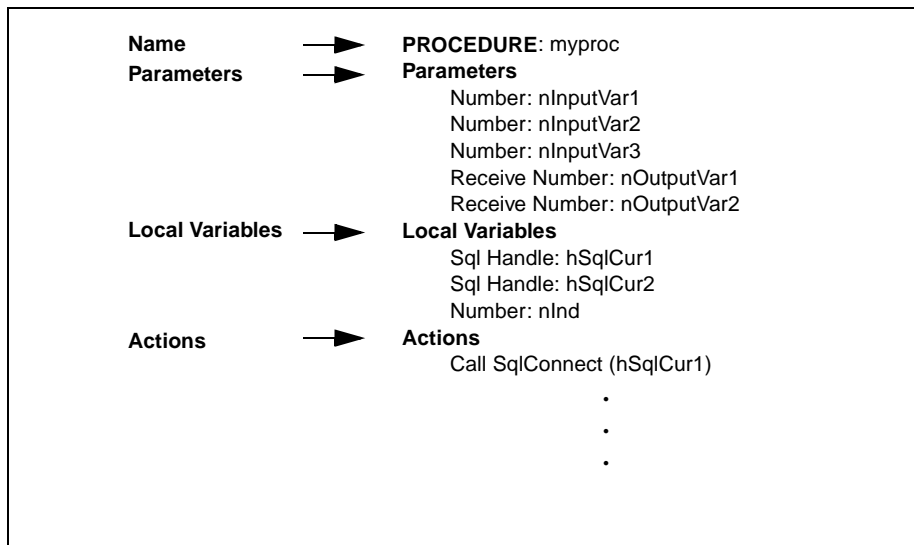
This example shows a sample procedure and its format.

Name

Every procedure has a name. For example:

```
PROCEDURE: WithDraw
```

The procedure name is a long identifier, and can contain up to 18 characters.

Note: Even though the colon is optional, you must supply it if your procedures are to be compatible with Team Developer.

When you store a procedure, you give it an additional name that lets you refer to the procedure as well as access it once it is stored (this parallels the syntax for stored commands). You can assign a stored name that differs from the procedure name. For example:

```
STORE WDPROC
PROCEDURE: WithDraw
Parameters
...
```

Note that you cannot replace an existing procedure with one that uses the same stored name. As in the example, assume you have stored procedure WithDraw under WDPROC. You cannot replace WDPROC with a procedure that uses the same stored name unless you have erased WDPROC first using the SQLTalk ERASE command.

Parameters

Parameters enable you to provide input to and receive output from a procedure. This section is optional; you do not have to define parameters for a procedure. You supply the values for all the parameters when you execute the procedure.

Declare a parameter using this syntax:

```
[Receive] DataType [:] ParameterName
```

Note: Even though the colon is optional, you must supply it if your procedures are to be compatible with Team Developer.

For example:

```
Parameters
Boolean: bDone
Date/Time: dtBirthDate
Number: nCount
Receive Number: nTotal
String: strLastName
```

See the *Data types supported in procedures* on page *7-9* for information on valid data types for parameters.

Output parameters in procedures must be preceded with the keyword *Receive*:

```
Receive Number: nTotal
```

SQLTalk accepts values for binding for input parameters. For output receive parameters, you must supply a place holder, with or without a value, for all binds which map to those parameters. If the receive parameter is used strictly as output, you can use a comma (,) with no leading space as a placeholder.

On the other hand, a SQL/API application uses bind values for input, and sets buffers to receive output values. In the SQL/API, an output parameter's value (generated by an executing function such as *sqlexe)* can be retrieved with the *sqlssb* function (Set Buffer) before the procedure starts executing, and then by the *sqlfet* function (Fetch) after the procedure passes control back to the invoker.

Note: In SQLTalk, output strings default to 80. This means you should resize the column(s) generated from the procedure with the COLUMN command.

You cannot pass an array as a parameter. All parameters passed into a procedure keep the values that were passed in, whether null or not null.

Local variables

Local variables perform several functions in SQL statements:

- They store data.
- They bind input data to a SQL statement. Variables used in this way are called bind variables.
- They specify where to put the output of a SQL SELECT statement. The SELECT statement's INTO clause specifies the variables where query data is placed. Variables in an INTO clause are called into variables.

This section is optional; you do not have to define local variables for a procedure.

The *Receive* keyword is not supported for local variables.

Declare a local variable using the same syntax as parameters:

```
DataType [:] LocalVariableName
```

Local variables are available to and accessible by only the procedure in which they are defined. They are also automatic, which means that they are created when the procedure executes and destroyed when the procedure ends.

Data you store in a variable are active across all stages of a procedure; their initial values persist across multiple fetch and execute statements, and are destroyed only when the procedure closes. Once the procedure closes, however, these values are not retained for future invocations. See the section on the ON directive for information on procedure states.

Variable buffers are allocated dynamically.

In addition to those data types supported for parameters, the Local Variables section also supports Sql Handles and File Handles. For example:

```
Sql Handle: hSqlCur1
File Handle: hFileActive
```

> **Note:** Even though the colon is optional, you must supply it if your procedures are to be compatible with Team Developer.

If you do not initialize a local variable, SQLBase assigns it a default value based on its data type when the procedure is invoked and before it takes control. For default value information, read the section *Data types supported in procedures* on page *7-9.*

Like parameters, you cannot pass an array as a local variable in a stored procedure.

Actions

This section contains statements to be executed depending upon the state of the procedure. It also contains logic flow language that controls the order in which SQLBase executes the statements.

Read *Appendix A* for a detailed description of the SAL functions you can include in a procedure.

Unlike Team Builder, you cannot include user-defined functions in procedures. However, your procedure can invoke another procedure that performs the work of your desired function.

Statement blocks

A *block* in the Actions section contains a set of statements to be executed in successive order. All the statements in a block are either of the same indentation level or enclosed within Begin and End statements.

Indentation

Indentation is an important element of logic flow. Use it to control the order in which SQLBase executes blocks of statements in a procedure.

SQLBase is very strict about indentation, and a change in indentation is interpreted as a block change. For example, when defining parameters, make sure that all of them are indented by the same amount:

```
Parameters
  Boolean: bDone
  Date/Time: dtBirthDate
  Number: nCount
```

Defining them according to the following example will produce an error:

```
Parameters
    Boolean: bDone
        Date/Time: dtBirthDate
  Number: nCount
```

You can use spaces or tabs to implement indentation. If you are using spaces, one or more spaces defines a specific indentation.

Note: Do not mix spaces and tabs for indentation. For example, four spaces may appear to have the same indentation as a tab in your on-line editor, but the four spaces represent four levels of indentation, while a tab only represents one.

This is an example of valid indentation:

```
Loop Outerloop
  If I3 > 0
    If NOT SqlExecute (hSqlCur1)
      Return 201
    Set I3 = I3 - 1
  Else
    Break Outerloop
```

Using Begin and End statements (block delimiters)

Another way to achieve the same level of control is to use *block delimiters* to surround a set of statements. To use block delimiters, begin a set of statements with Begin, and end with End. This allows you to reduce the number of indentation layers in your program.

Using Begin and End statements reduces the number of indentation levels in the previous example:

```
On Procedure Execute
  Loop Outerloop
  Begin
  If I3 > 0
    If NOT SqlExecute (hSqlCur1)
      Return 201
    Set I3 = I3 - 1
  Else
    Break Outerloop
  End
```

Block delimiters are only allowed in a procedure's Actions section.

Note: The If, Else, Else if, and Loop statements require either indentation or a Begin and End statement.

Data types supported in procedures

You must specify one of the following data types when defining parameters and local variables in procedures.

The following table lists valid data types supported in procedures and their default value. It also lists their SQL standard naming prefix. Although not required, using these prefixes in the names of variables will help make your procedure self-documenting.

Data type	Default Value	Suggested Name prefix	Example	Comments
Boolean	FALSE	b	bOk	
Sql Handle	none	hSql	hSqlCur1	Supported only for local variables.
Date/Time	null	dt	dtStartDate	
String	null string	s (or) str	strLastName	Use the Long String data types for strings longer than 254 bytes
Long String	null string	s (or) str	strLastName	Supports strings longer than 254 bytes
Number	0	n	nSalary	
Window Handle	0	hWin	hWinActive	Bind to the variable using the program data type SQLNUM. The same holds for set select buffer. Cannot be used for any arithmetic operation.
File Handle	0	hFile	hFileActive	Supported only for local variables. Cannot be used for any arithmetic operation

Note the following restrictions:

- You cannot pass an array as a parameter to a procedure.
- Unlike Team Developer, you cannot use user-defined constants in a procedure. However, you can use system constants. Read the next section, System constants supported in procedures for details.

All data types can be an alternate form called a *receive* data type, which identify output parameters. Receive data types allow you to pass data to a procedure by reference rather than value. This means that SQLBase passes the variable's storage address and the procedure has access to the original value which it can then change. For example:

```
Parameters
  Receive Boolean: bOrderFilled
...
Actions
  Set bOrderFilled = TRUE
```

Note: All parameters passed into a procedure keep the values that were passed in, whether null or not null.

Unless otherwise noted, procedure data types conform to SQLBase data type formats. Note that these may be different from Team Developer data type formats.

Boolean

Use this data type for variables that can be TRUE (1) or FALSE (0). For example:

```
Local Variables
  Boolean: bDone
...
Actions
  Set bDone = FALSE
```

Date/Time

Use this data type for date and/or time variables. For example:

```
Parameters
  Date/Time: dtBirthday
...
Actions
  If dtBirthday > 07/01/1983
```

Number

Use this data type for numbers with up to 15 digits of precision. For example:

```
Parameters
  Number: nMonth
...
Actions
  If nMonth = 3
...
```

Sql Handle

Use this data type to identify an existing connection to a database. All access to a database requires a Sql Handle. For example:

```
Local Variables
  Sql Handle: hSqlCur1
...
Actions
  Call SqlConnect (hSqlCur1)
```

String

Use this data type for character data. Unlike Team Developer, the maximum length of a procedure string is 64 Kbytes; however, if a string is used as a receive parameter, its length cannot exceed 254 characters on return from the procedure. If its length exceeds 254 characters, SQLBase issues an error message. Use the Long String data type to return strings longer than 254 characters,

Enclose literal strings in single quotes. For example:

```
PROCEDURE: CLIENTPROC
Parameters
  Receive Date/Time: dtAppt
  Receive String: sSelect
Local Variables
  Sql Handle: hSqlCur1
  Number: nInd
Actions
  Call SqlConnect(hSqlCur1)
  Set sSelect = 'Select max(APPT) from CLIENT into :dtAppt '
  Call SqlPrepare(hSqlCur1, sSelect)
  ......
```

Long String

Use this data type for character data to return strings greater than 254 bytes or to bind the string to a LONG VARCHAR column type. Note that the behavior of a Long String data type is identical to the String data type with the following exceptions:

- When used to return data (Receive Long String), the data type is identical to Long Varchar. For example, if you use *sqldes*() to describe the parameter, the data type returned will be SQLDLON. You must use the read long primitives to fetch this data.

- When used to bind data, SQLBase uses the write long primitives to bind to the string variable. SQLBase treats the target column as a Long Varchar.

Enclose literal strings in single quotes. For example:

```
Variables
  Long String: sLong
...
  Set sLong = 'Long String'
```

Window Handle

Use this data type to store window handles. A window handle identifies a single instance of a particular window. This data type supports the SAL and WINAPI functions that use and manipulate window handles. If this data type is used in the parameter section of the procedure (that is, input/output), bind to the variable using the program data type SQLPNUM. The same holds for set select buffer. For example:

```
PROCEDURE: CLIENTPROC
Parameters
  Window Handle: hWind
Actions
  Call SalSendMsg(hWind, ...)

....
```

File Handle

Use this data type to store file handles. A file window identifies an open file. This data type supports the SAL file manipulation functions. For example:

```
Local Variables
  File Handle: hFileActive
...
Actions
  Call SalFileOpen (hFileActive, ...)
```

System constants supported in procedures

You can use the following standard system constants:

- The null constants: STRING_Null, NUMBER_Null, and DATETIME_Null

  You can check for nulls within procedures using null constants. For example, you can create a boolean expression, such as:

  ```
    IF (A = NUMBER_Null)
    IF (S = STRING_Null)
  ```
  If the variable is null, the expression evaluates to TRUE.

- The TRUE and FALSE boolean constants.

- The Fetch_Delete, Fetch_EOF, Fetch_Ok, and Fetch_Update constants.

- The DBP parameters: DBP_AUTOCOMMIT, DBP_BRAND, DBP_PRESERVE, DBP_VERSION, DBP_LOCKWAITTIMEOUT, DBP_ROLLBACKTIMEOUT.
- The DBV_BRAND database brands: DBV_BRAND_DB2, DBV_BRAND_ORACLE, and DBV_BRAND_SQL.

For details on these constants, read the constant descriptions in *Appendix A*.

Note: System constants in SQLBase are case insensitive. Case sensitivity that appears in the system constants listed in this section apply only to Team Developer.

Using SAL statements

Use Scalable Application Language (SAL) statements to control the logic flow of the statements in a procedure. SQLBase provides the following SAL statements:

- Break
- Call
- If, Else, and Else If
- Loop
- On
- Return
- Set
- When SqlError
- While

Break

The Break statement terminates a Loop statement. If you specify a loop name, that particular loop terminates. This allows you to break out of more than one level of loop. If you do not specify a loop name, the BREAK statement breaks out of the most recently-entered loop.

Syntax

```
Break [loopname]
```

Example

```
Loop
  Set nOutput2 = nOutput2 + nInput2
  If nOutput2 > nInput2 + 10
    Break
```

Call

The Call statement executes a SAL function. SAL functions are listed in the following section *SAL functionality in SQLBase*.

Syntax

```
Call FunctionName (Parameters, ... )
```

Example

```
Call SqlImmediate ( 'DELETE FROM CUSTOMER WHERE \
     CUSTNO = 1290' )
```

Be aware that using the Call statement means that the function's return value is lost. However, if an error is returned, SQLBase passes control to the closest error handle. Read *Error handling* on page *7-45*.

If, Else, and Else If

The If, Else, and Else If statements execute other statements based on the outcome of an expression. The Else and Else If parts are optional. For each If statement, you can code as many Else If sections as you want, but there can be only one Else section.

Indentation determines the conditional flow of control.

Syntax

```
If Expression1
  <statement(s)>
Else If Expression2
  <statement(s)>
Else
  <statement(s)>
```

If *Expression1* evaluates to TRUE, the first set of statements executes. If *Expression1* evaluates to FALSE, *Expression2* is evaluated. If *Expression2* evaluates to TRUE, the second set of statements executes. If *Expression2* evaluates to FALSE, the third set of statements executes.

Example

```
If nMonthly_Salary < 1000
  Set nTax_Rate = 10
Else If nMonthly_Salary < 2000
  Set nTax_Rate = 20
Else
  Set nTax_Rate = 25
```

Loop

The Loop statement repeats a set of statements until a Break or Return statement is executed.

Syntax

```
Loop [loopname]
```

The loopname is optional. Specifying a loopname lets you refer to that loop in a later Break statement.

Examples

```
Loop
  If nCount = 100
    Return 1
  Set nCount = nCount + 1
```

and:

```
Loop Outer
  If I3 > 0
    If NOT SqlExecute ( hSqlCur )
      Return 201
    Set I3 = I3 – 1
  Else
    Break Outer
```

On <procedure state>

The ON directive identifies the procedure's current state, such as startup or executing. When a procedure is at a specific state, the statements indented underneath it are processed. The state of a procedure changes as the procedure execution progresses. A procedure can be at any of the following states:

- Procedure Startup
- Procedure Execute
- Procedure Fetch
- Procedure Close

Using ON directives is optional. If you do not specify an ON directive in a procedure, SQLBase processes the *entire* procedure when the calling program issues an execute command. In other words, not specifying any ON directive in a procedure's Actions section is equivalent to including only an On Procedure Execute section under Actions (see the following paragraphs).

The default state (On Procedure Execute) is often adequate for many procedures. However, there are two situations in particular which do require one or more specific ON <procedure states>:

- If you wish to repeatedly execute a procedure, such as when supplying different parameter values, it can be more efficient to code an On Procedure Startup state that contains commands requiring only a single execution (for example, database connections and variable assignments.) This avoids unnecessary multiple executions of these commands.

- When you are fetching multiple rows, an On Procedure Fetch state is required.

SQLBase processes the Procedure Startup and Procedure Close sections only once. The Procedure Execute and Procedure Fetch sections can be processed as many times as you want. Local variables values are retained through multiple execute and fetch operations; the values are only destroyed at the close section.

SQLBase only allows you to specify On directives at the topmost level of the Actions section. In other words, you cannot nest an On directive within a statement block or between Begin and End statements.

To retrieve all the output data generated by the procedures, you must declare as many output variables as the number of items you want returned.

The following paragraphs describe the different procedure states.

Procedure Startup. A procedure is in procedure startup state after the following two steps are completed:

1. The calling program compiles the procedure (for example, with the SQL/API *sqlcom* function).

2. The calling program executes the procedure for the first time (for example, with the first SQL/API *sqlexe* function).

After processing the commands in the Procedure Startup stage, the first execute command from the calling program also processes the commands in the Procedure Execute stage. In other words, the calling program's first execute command processes both the Procedure Startup and Procedure Execute sections.

However, *subsequent* execute commands from the calling program only process the Procedure Execute stage; they do not process the Procedure Startup section again.

Procedure Execute. A procedure is in procedure execute state after the following two steps are completed:

1. The calling program first executes the procedure.

2. The Procedure Startup section is processed.

The Procedure Execute section is processed and reprocessed each time the calling program issues subsequent execute commands.

Procedure Fetch. If the calling program issues a FETCH command (for example, with the SQL/API *sqlfet* function) and you have a Procedure Fetch section, the statements in the Procedure Fetch section are processed. The Procedure Fetch section is processed and reprocessed each time you issue a FETCH command.

You must include a Procedure Fetch section to fetch multiple rows in your procedure. It is recommended that you also include a Return statement (see the following section on *Return*) to first return 0 while fetching is in progress, and then return 1 when the fetch is finished.

To retrieve all the output data generated by the procedures, you must declare as many output variables as the number of items you want returned.

Note that for each row returned by a procedure, the On Procedure Fetch section is executed. With multi-row buffering, therefore, a FETCH command from the client can cause the On Procedure Fetch section to be executed several times (as many times as the number of rows that fit into the buffer, or until end of the fetch). See the following *Examples* section which contains a procedure that demonstrates multi-row buffering behavior. Although multi-row buffering is a performance feature, it can result in unexpected behavior.

For example, you may expect that a single fetch command from the client causes the Procedure Fetch section to issue a COMMIT each time it returns a row. But instead, you find with multi-row buffering that the On Procedure Fetch section issues several COMMITs for the first row returned to the client.

If needed, you can have the On Procedure Fetch section generate exactly one row for each fetch call from the client, by setting the FETCHTHROUGH mode ON at the client. The default is OFF.

There are two ways to set FETCHTHROUGH mode:

- From SQLTalk, use the SET FETCHTHOUGH ON command
- From SQL/API, use *sqlset* function with the SQLPFT parameter

Procedure Close. Finally, when the calling program either issues a disconnect command (for example, with the SQL/API *sqldis* function) or you compile another command on the same cursor that the calling program was using to execute the procedure, the Procedure Close section is processed.

Syntax

```
On <procedure state>
  <statement(s)>
```

Examples

This section shows examples of various procedure states using the ON directive. You can find most of the examples shown in this section in the directory \Centura\SP.SQL. These examples use the following PRODUCT_INVENTORY table:

```
create table PRODUDCT_INVENTORY (NAME varchar(25),
INVENTORY decimal (3,0), WHEN date);
insert into PRODUDCT_INVENTORY values (:1,:2,:3)

\

JF 12R,132,13-OCT-1992
DJ Y5Y,165,11-OCT-1992
DJ Y5Y,159,12-OCT-1992

/
```

Example with ON PROCEDURE states. This example prepares, executes, and fetches results from a procedure called PRODUDCT_INPROC.

```
    PREPARE
    PROCEDURE: PRODUDCT_INPROC
    Parameters
      String: sName
      Receive Number: nINVENTORY
    Local Variables
      Sql Handle: hSqlCur1
      String: sSelect
      Number: nInd
    Actions
①   On Procedure Startup
        Call SqlConnect(hSqlCur1)
        Set sSelect = 'Select INVENTORY from PRODUDCT_INVENTORY \
          where NAME = :sName into :nINVENTORY'
        Call SqlPrepare(hSqlCur1, sSelect)
②   On Procedure Execute
          Call SqlExecute(hSqlCur1)
③   On Procedure Fetch
          If NOT SqlFetchNext(hSqlCur1, nInd)
            Return 1
          Else
            Return 0
④   On Procedure Close
          Call SqlDisconnect(hSqlCur1)

⑤ perform PRODUDCT_INPROC

    \

    JF 12R,,
```

```
      /
⑥  FETCH 1;
⑦  perform PRODUDCT_INPROC
      \
      DJ Y5Y,,
      /
⑧  FETCH 2;
⑨  SELECT * from PRODUDCT_INVENTORY;
```

1. This state is processed only once on the first EXECUTE by the calling program. If the calling program re-executes the procedure, the commands in this section are not processed again. This reduces procedure performance overhead.

2. This state is processed every time the calling program issues an EXECUTE command. If there are no ON <procedure states> coded, the procedure defaults to this state.

3. This state is processed every time the calling program issues a FETCH command, and is essential to fetching multiple rows.

4. This state is processed only 1) after the procedure has finished all processing, or 2) if another command is compiled or executed on the calling program's current cursor, or that cursor becomes disconnected.

5. The calling program executes the procedure for the first time. The On Procedure Startup and On Procedure Execute states are processed. Note that the second comma used in the SQLTalk PERFORM command for the binding of the Procedure provides the required placeholder for the procedure's Receive parameter nINVENTORY. You must provide either a placeholder comma or an argument value for all procedure parameters.

6. This the first fetch issued by the calling applications. The On Procedure Fetch state is processed multiple times until end-of-fetch or until the buffer is full.

7. The calling program executes the procedure for the second time with a different bind value. Only the On Procedure Execute state is processed.

8. The calling program issues another fetch, this time with a different bind value. Two rows are returned to the client.

9. The On Procedure Close state is processed for the previous procedure.

Example with no On Procedure states. The next example compiles, executes and fetches a single row from a procedure which defaults to the On Procedure Execute state for all code under Actions.

```
      PROCEDURE: PRODUDCT_INPROC
      Parameters
        Receive Number: nSumINVENTORY
      Local Variables
        Sql Handle: hSqlCur1
        String: sSelect
        Number: nInd
      Actions
①     Call SqlConnect(hSqlCur1)
      Set sSelect = 'Select max(INVENTORY) from
      PRODUDCT_INVENTORYinto :nSumINVENTORY'
      Call SqlPrepare(hSqlCur1, sSelect)
      Call SqlExecute(hSqlCur1)


②     If NOT SqlFetchNext(hSqlCur1, nInd)
        Return 1
      Else
        Return 0
      Call SqlDisconnect(hSqlCur1)
      \

      ,

      /
```

1. Since there are no On Procedure statements, the entire procedure defaults to the On Procedure Execute state.

2. There is no On Procedure Fetch state in this procedure. This means that the calling program can FETCH from the ON Procedure Execute state by embedding SAL fetch calls like SqlFetchNext. However, in this instance you can only fetch and return to the caller a single row (even if within the procedure the fetch is in a loop). In this case, the caller's FETCH will only return the receive parameter values and perform no other processing.

Example with single row fetch and multiple row result. This example generates a single row fetch and then manipulates that data in order to produce a multiple row result. In this case the output is only indirectly tied to the database. This is a good method to produce "what-if" scenarios. In general, any fetches from the calling application do not necessarily have to have database sources within the procedure.

```
      PROCEDURE: PRODUDCT_INPROC
      Parameters
        String: sName
        Receive Number: nCurrentIN
        Receive Number: nDays
      Local Variables
        Sql Handle: hSqlCur1
```

```
     String: sSelect
     Number: nMaxINVENTORY
     Number: nInd
  Actions
    On Procedure Startup
      Call SqlConnect(hSqlCur1)
      Set sSelect = 'select max(INVENTORY) \
        from PRODUDCT_INVENTORY
        where NAME = :sName into :nMaxINVENTORY'
      Call SqlPrepare(hSqlCur1, sSelect)
    On Procedure Execute
      Call SqlExecute(hSqlCur1)
①    Call SqlFetchNext(hSqlCur1, nInd)
      Set nCurrentIN = nMaxINVENTORY
    On Procedure Fetch
②    If nCurrentIN < 200
        Set nCurrentIN = nCurrentIN + 10
        Set nDays = nDays + 1
        Return 0
      Else
        Return 1
    On Procedure Close
      Call SqlDisconnect(hSqlCur1)
  \
  DJ Y5Y,,,

  /
```

1. Because an On Procedure Fetch state is also coded, this single row fetch is not returned to the caller and is only used internally by the procedure for subsequent processing.

2. This statement lists the inventory by day (10 daily increase) until the inventory is greater than 200, starting from the historical maximum inventory. In this case, the caller is not directly fetching from the database.

Example of fetch with default multi-row buffering behavior. This example generates a multi-row buffer when a single fetch has been issued against the procedure. This example is only intended to show the affect of multi-row buffering.

```
create table X (COL1 int);

TABLE CREATED

insert into X values(:1)

\
1
```

```
2
3
/

PROCESSING DATA

1
2
3

3 ROWS INSERTED

create table Y (COL1 int);

TABLE CREATED

-- Set FETCHTHROUGH ON at client before executing
-- this procedure if you want to maintain 6.0.0 procedure
-- fetch semantics:

prepare
procedure: MROWBUF1
Parameters
     Receive Number: nCol1
Local Variables
     Sql Handle: hSqlCur1
     Number: nInd
Actions
     On Procedure Execute
          Call SqlConnect( hSqlCur1 )
          Call SqlPrepareAndExecute( hSqlCur1, 'select\
            COL1 from X into :nCol1')
        ! 1 fetch from client causes On Procedure Fetch
        ! to be executed multiple times
     On Procedure Fetch
          If NOT SqlFetchNext( hSqlCur1, nInd )
             Return 1
          Else
             Call SqlImmediate('insert into Y values \
                 (:nCol1)')
             Return 0
       On Procedure Close
          Call SqlDisconnect( hSqlCur1 )
;

STATEMENT PREPARED
```

```
perform;

PROCESSING DATA

STATEMENT PERFORMED

fetch 1;

NCOL1
=====
1

1 ROW RETRIEVED FROM PROCEDURE

-- 3 rows should be inserted into Y because 1
-- fetch from client causes On Procedure Fetch
-- to be executed 3 times in this case.

select * from Y;

COL1
====
1
2
3

3 ROWS SELECTED
```

Example of data manipulation at the server if no data needs to be fetched at the client. This example is the recommended method for achieving the same results in the previous example. This example omits the On Procedure Fetch section.

```
drop table y;
create table y;
procedure: MOVE_DATA
Local Variables
     Sql Handle: hSqlCur1
     Number: nInd
     Number: nCol1
Actions
      ! Omission of On Procedure section defaults
     ! to On Procedure Execute
     Call SqlConnect( hSqlCur1 )
     Call SqlPrepareAndExecute( hSqlCur1, 'select \
       COL1 from X into :nCol1')
     While SqlFetchNext( hSqlCur1, nInd )

       Call SqlImmediate('insert into Y values (:nCol1)');
```

```
0 ROWS RETRIEVED FROM PROCEDURE

-- Same result as earlier example without the
-- need for a client fetch:

select * from Y;

COL1
====
1
2
3

3 ROWS SELECTED
```

Return

The Return statement breaks the flow of control and returns control to the calling program.

The exception is when a Return is executed from the When SqlError section. In this situation, control is returned back to the procedure with the boolean return (TRUE/FALSE). This becomes the return value for the failed SAL Sql* function. The procedure then resumes execution according to the Boolean return.

If you do not specify a Return statement in a procedure, one of the following codes is returned to the calling program:

- If a SQL error occurs and there is no When SQLError block, the procedure returns the error code. If there is a When SQLError block and a return statement within the block, the procedure does not return the error code.

- If no error occurs, the procedure returns 0.

Note: If the calling program performs fetches in a loop and expects an end-of- fetch return from the procedure, the On Procedure Fetch section must be coded with an appropriate return (usually Return 1) or the or the calling program will go into an endless loop

Syntax

```
Return <expression>
```

The expression is mandatory, and can be anything that evaluates to a number.

- If you code a Return statement in a When SqlError block (see the section on When SqlError), you can only return a boolean such as TRUE or FALSE.

• If you code a Return statement outside of a When SqlError block, you can only return integer values. You can code these as either constants or variables. You cannot return a string, date/time, or SQL Handle local variable type.

Example

```
On Procedure Startup
  When SqlError
    Set nRcd = SqlError(hSqlCur1)
    If nRcd = 601
      Return FALSE
    Else
      Return TRUE
  .....
....
On Procedure Fetch
  If NOT SqlFetchNext(hSqlCur1, nInd)
    Return 1
  Else
    Return 0
....
```

Set

The Set statement assigns a value to a variable. You can set a variable to the value of another variable.

Syntax

```
Set VariableName = Expression
```

Example

```
!Declare two variables for End-of-File and Return Code
Local Variables
  Boolean: bEOF
  Number: nRCD
...
Actions
  Set bEOF = FALSE
  Set nRCD = 0
```

Trace

The Trace statement prints the value of one or more variables. Use it when debugging a procedure to check the values of variables. For example, code a Trace statement immediately before and after a command that you expect will change the value of a variable.

This statement is different from the SQLTalk SET TRACE command, which is issued independently of the procedure and traces every statement the procedure executes. You do not need to run SET TRACE ON to use the Trace statement.

By default, output from the Trace function is sent to the Process Activity screen for a multi-user server, and is not displayed for a single-user engine. Generally, you will want to direct the output to a file on the server with the SQLTalk SET TRACEFILE command.

Syntax

```
Trace Variable1, Variable2, ..., VariableN
```

Example

This example shows a procedure using the Trace statement to trace the values of two variables nCount and nRcd. It traces the values at different points in the procedure.

```
PROCEDURE: TRPROC
Local Variables
  Number: nCount
Actions
  Trace nCount
  Loop
    Set nCount = nCount + 1
    Trace nCount
    If nCount > 10
      Trace nCount
      Return 0
;
```

When SqlError

The When SqlError statement declares a local error handler. To learn more about local error handling, see the *Error Handling* section later in this chapter.

Syntax

```
When SqlError
  <statement(s)>
```

Example

This example demonstrates local error handling with SqlError. It uses the following tables JF and PRODUDCT_INVENTORY:

```
CREATE TABLE JF 12R,132,13-OCT-1992 (NAME varchar(25),
    INVENTORY decimal (3,0), WHEN date);
INSERT INTO PRODUDCT_INVENTORY values
    (JF 12R,132,13-OCT-1992);
```

```
                COMMIT;
```

This examples also uses the following stored command INVENTORY_QUERY:

```
    STORE INVENTORY_QUERY
        SELECT INVENTORY from PRODUDCT_INVENTORY
        where NAME = :1;
```

To create the error condition, the stored command is dropped prior to procedure execution. The procedure's When SqlError section traps error #207 (Command not found for retrieval) and fixes the problem of the missing stored command.

```
    ERASE INVENTORY_QUERY;
    PROCEDURE: ILPROC
    Parameters
      String: sName
      Receive Number: nINVENTORY
    Local Variables
      Sql Handle: hSqlCur1
      Number: nInd
      Number: nRcd
    Actions
      On Procedure Startup
①       When SqlError
            Set nRcd = SqlError(hSqlCur1)
            If nRcd = 207
              Call SqlStore(hSqlCur1, 'INVENTORY_QUERY', \
                'select INVENTORY \
                from PRODUDCT_INVENTORY \
                where NAME = :1 into :2')
            Call SqlCommit(hSqlCur1)
②           Call SqlRetrieve(hSqlCur1, 'INVENTORY_QUERY', \
                ':sName', ':nINVENTORY')
            Return TRUE

          Call SqlConnect(hSqlCur1)


          Call SqlRetrieve(hSqlCur1, 'INVENTORY_QUERY', \
          ':sName', ':nINVENTORY')
      On Procedure Execute
        Call SqlExecute(hSqlCur1)
      On Procedure Fetch
        If NOT SqlFetchNext(hSqlCur1, nInd)
          Return 1
        Else
          Return 0
      On Procedure Close
        Call SqlDisconnect(hSqlCur1)
```

```
\
DJ Y5Y,,
/
```

1. This exception handling routine can detect the SQL error generated by the
    SqlRetrieve call, and handle this error by restoring the non-existing stored
    command. In order to continue processing the procedure, the error handler returns
    TRUE back to the procedure, and executes the stored command. If other SQL
    errors are encountered, no Return is executed; control (along with the SQL error
    code) is immediately returned to the calling program.

2. This call will fail due to the non-existing stored command. In this example, When
    SqlError forces SqlRetrieve to return TRUE, and the procedure continues to
    execute successfully.

While

The While statement repeats until the expression being evaluated becomes FALSE.

Syntax

```
While Expression
  <statement(s)>
```

Example

```
...
While nInputVar3 > 0
  If NOT SqlExecute ( hSqlCur1 )
   Return 201
  Set nInputVar3 = nInputVar3 - 1
  ...
```

Comments

Comment lines allow you to include explanations in a procedure. A comment starts at
the beginning of a line with an exclamation point (!) and ends with a carriage return
or line feed character. Comments and code are not allowed on the same line.

You do not need to follow indentation rules for comments.

Syntax

```
! Comment line
```
Example

```
! These are comment lines; SQLBase does not attempt to
! execute them.
```

Operators

These operators are supported in procedures and, excluding string concatenation, are listed according to precedence:

Operator	Description
( )	Parentheses
unary -	Unary
*, /	Numeric: multiply, divide
+, -	Numeric: add, subtract
>, <, >=, <=	Relational: greater than, less than, greater than or equal to, less than or equal to
=, !=	Relational: equal to, not equal to
&	Bitwise AND
|	Bitwise OR
NOT	Boolean NOT
AND	Boolean AND
OR	Boolean OR
||	Concatenate string

Continuation lines and concatenation

Use a backslash (\) at the end of a line to continue a statement on the next line. For example:

```
Actions
  Set sUpdate = 'UPDATE Checking \
    Set Balance = Balance - :nAmount \
    WHERE AccountNum = :nAccount'
```

You can also use the double line symbol (\|\|) to concatenate strings. For example:

```
Set sWhere = 'where INVENTORY <200'
Set sSelect = 'SELECT name, inventory \
  from PRODUDCT_INVENTORY' ||sWhere || 'into :sName,
  :nINVENTORY'
```

How to generate, store, execute and drop procedures

This section describes how to generate, store, execute, and drop procedures through SQLTalk. It also describes debugging a procedure and security issues.

You can also perform these functions through the SQL/API (see the section *Using SQL/API functions with procedures* later in the chapter for a list of associated functions), through SQLConsole with the Procedures Editor, and through Team Builder with the following functions:

> *SqlStore*
> *SqlRetrieve*
> *SqlExecute*
> *SqlDropStoredCmd*

See the documentation for these products for more detail. In addition, SQLBase provides an online sample application called *sp.app* that demonstrates calling procedures in a Team Developer application.

Like all other SQL commands, procedures can be stored, retrieved, compiled, and executed through applications such as the SQL/API.

Generating a procedure

To generate a procedure, use the SQL PROCEDURE command. Read the *SQL Language Reference* for detailed information on this command. When you generate the procedure, specify the SAL statements and any parameters and local variables in the Actions section.

If you turn on result sets on the client, you can scroll forwards and backwards through the result set returned by a procedure. The result set for procedures is preserved across COMMIT and ROLLBACK operations, even if preserve context mode is off. In SQLTalk, result set mode is OFF by default.

Note: To avoid necessary performance degradation, keep result set mode OFF if you are not using scrollable cursors. When set to ON, SQLBase builds the result set for a procedure.

Following are restrictions to note when generating procedures:

- If you are using a network version of SQLBase, you cannot create procedures that perform SQL commands that use a SET SERVER command. These commands are:

    CREATE DATABASE
    DROP DATABASE

```
CREATE STOGROUP
DELETE
INSTALL DATABASE
DEINSTALL DATA
```

- Recursion and nesting limits of procedures are determined by various settings in your system, such as available memory.

- You cannot include DDL commands in static procedures. The only exception is if you are specifying the LOAD...ON SERVER.. command in the procedure and the file you are loading contains DDL commands.

- Restriction mode to filter a result set is not supported for procedures.

Procedure Validation

Typically, a procedure performs some action on a table, or contains stored commands that reference tables in a single SQL statement. Note that SQLBase allows users to drop or alter a table even if it is referenced in a procedure or in a stored command that is contained in a procedure.

If the object a procedure references is changed or no longer exists, the procedure remains valid. However, SQLBase issues a runtime error about the missing objects when the procedure is executed. In addition, if you attempt to load a static procedure that references a dropped or altered object, SQLBase also issues errors when it cannot locate the missing or altered objects.

To execute and load procedures successfully, be sure to recreate any referenced object that is dropped, or restore any referenced object that is altered to its original state (as known by the procedure).

Procedure Example

The following procedure updates and returns bank account balances. In this example, the procedure is executed through SQLTalk. This example uses the following table:

```
CREATE TABLE CHECKING (ACCOUNTNUM number, BALANCE
     number);
PROCEDURE: WITHDRAW
Parameters
  Number: nAccount
  Number: nAmount
  Receive Number: nNewBalance
Local Variables
  String: sUpdate
  String: sSelect
Actions
  Set sUpdate = 'UPDATE CHECKING \
    set BALANCE = BALANCE - :nAmount \
```

```
      where ACCOUNTNUM = :nAccount'
    Call SqlImmediate(sUpdate)
    Set sSelect = 'SELECT BALANCE from CHECKING \
      where ACCOUNTNUM = :nAccount \
      into :nNewBalance'
    Call SqlImmediate(sSelect)
  \
  1,50,,
  /
```

Remember to follow the indentation guidelines when creating your procedure. Read the section *Indentation* on page *7-7* for more information. Also, if you are breaking a long line to span multiple lines, you must use a backslash (\) at the end of the line as a continuation marker.

Of course, typing a long procedure directly into the SQLTalk interface is time-consuming, especially if you make typing errors. Generally, you will want instead to create a script that contains the PROCEDURE command. You can then use the SQLTalk RUN command to run this script in SQLTalk.

Static versus dynamic procedures

A procedure is either dynamic or static. Dynamic is the default.

The following table lists important differences between static and dynamic procedures.

Feature	Dynamic	Static
Require storing to execute?	No	Yes
Parse/precompile procedural logic?	Yes	Yes
Parse SQL at store time?	No	Yes
Precompile SQL at store time?	No	Yes
Dynamic SQL support?	Yes	No
*SQL performance	Slower	Faster
Use for Triggers?	No	Yes

* Performance of Dynamic procedures can be enhanced by retrieving previously stored SQL commands (SqlRetrieve as opposed to SqlPrepare).

Static procedures. SQLBase compiles and optimizes (determines the query plan) the SQL statements embedded in a *static* stored procedure. These statements and

their associated query execution plans are kept in the database. Static procedures must be stored before they can be executed.

You must be sure that a static procedure's embedded SQL commands meet the following criteria:

- They are not data definition language (DDL) statements.
- They are string literals and contain no variables other than bind or INTO variables.

The first requirement means that you cannot include a CREATE, ALTER, or DROP command in a static procedure. However, the procedure can contain a LOAD.. ON SERVER command that has DDL statements.

The second requirement means that SQLBase must know what the command string is at compilation time. For example, you cannot include the following excerpt in a static stored procedure:

```
Set sCmd = 'select * from employee'
...
SqlPrepare (cur, sCmd)
```

You must specify the actual command string itself:

```
SqlPrepare (cur, 'select * from employee')
```

As another example, this statement meets the static requirements:

```
Select Col1, Col2 from sysadm.Table1 into :Out1, :Out2
```

but these do not:

```
    Set sColumns = 'Col1, Col2'
    ...
    Set SELECT = 'Select' || sColumns || 'from sysadm.Table1
         into :Out1, :Out2 '
```

Note that as with any statement that contains bind variables, SQLBase must determine the optimal access method without all the necessary information.

While static procedures do not provide the flexibility of dynamic procedures, they do optimize and parse SQL statements before storing and hence yield higher performance at runtime.

The following example shows a static stored procedure:

```
    STORE STATICS
    PROCEDURE: STATIC_SQL static
    Parameters
      Receive String: sName
      Receive Number: nINVENTORY
    Local Variables
```

```
              Sql Handle: hSqlCur1
              Number: nInd
           Actions
             On Procedure Startup
               Call SqlConnect(hSqlCur1)
①            Call SqlPrepare(hSqlCur1, 'select NAME, INVENTORY from \
                 PRODUDCT_INVENTORY into :sName, :nINVENTORY')
             On Procedure Execute
               Call SqlExecute(hSqlCur1)
             On Procedure Fetch
               If NOT SqlFetchNext(hSqlCur1, nInd)
                 Return 1
               Else
                 Return 0
             On Procedure Close
               Call SqlDisconnect(hSqlCur1)

           /
           execute STATICS

           \

           ''
           /
```

1. When static procedures are executed, the SqlPrepare statement is not reprocessed since all SQL statements within a static procedure are precompiled. If you have already stored the SQL statement either using the SQLTalk STORE command or within the procedure using the SqlStore() function, SqlRetrieve() can be substituted.

Dynamic procedures. A *dynamic* procedure can contain dynamic embedded SQL statements. Because the dynamic SQL string components can change, the SQL statements cannot be precompiled.

Unlike static procedures, dynamic procedures do not have to be stored before they are executed.

Note that since SQL statements in a dynamic stored procedure are not parsed until execution, SQLBase does not catch any SQL errors in the procedure when you store it.

The previous examples of invalid embedded SQL statements for static procedures are acceptable for dynamic procedures:

```
Set sCmd = 'select * from employee'
...
SqlPrepare (cur, sCmd)
```

```
and:
Set sColumns = 'Col1, Col2'
...
   Set SELECT = 'Select' || sColumns || 'from sysadm.Table1
        into :Out1, :Out2 '
```

To improve dynamic procedure performance and avoid the possibility of runtime
errors, you can store SQL commands outside of the procedure (as opposed to using
SqlPrepare within the procedure) and then retrieve and execute them within the
procedure. However, this prevents you from using dynamic SQL for that particular
SQL statement.

The following example shows a dynamic SQL procedure. This procedure cannot be
static because of the symbolic string substitution of the SQL statement found in the
SqlPrepare() call. The SQL statements in a dynamic procedure are not precompiled
and so are not optimized or parsed when stored.

```
     store DYNAMITE
     procedure: DYNAMIC_SQL
     Parameters
       Number: nOver200
       Receive String: sName
       Receive Number: nINVENTORY
     Local Variables
       Sql Handle: hSqlCur1
       String: sWhere
       String: sSelect

       Number: nInd
     Actions
       On Procedure Startup
         Call SqlConnect(hSqlCur1)
            If nOver200 = 1
          Set sWhere = 'where INVENTORY > 200'
         Else
           Set sWhere = 'where INVENTORY < 200'
         Set sSelect = 'select NAME, INVENTORY \
                 from PRODUDCT_INVENTORY ' || sWhere || ' into
     :sName, :nINVENTORY'
         Call SqlPrepare(hSqlCur1, sSelect)
(1)      On Procedure Execute
         Call SqlExecute(hSqlCur1)
       On Procedure Fetch
         If NOT SqlFetchNext(hSqlCur1, nInd)
           Return 1
         Else
           Return 0
       On Procedure Close
```

```
        Call SqlDisconnect(hSqlCur1)

/
execute DYNAMITE

\

1,,,

/
```

The advantage to dynamic procedures is that they are more flexible than static procedures. You can run and rerun a dynamic stored procedure with embedded dynamic SQL by using string substitution to produce different SQL commands at run time.

Determining whether to store a procedure as dynamic or static. If you have a stored procedure that contains SQL statements, some of which would benefit from static storage and others which would benefit from dynamic storage, consider breaking the procedure into several smaller static and dynamic procedures. For example, you might have a main static stored procedure that calls several dynamic stored procedures.

Storing a procedure

Storing a procedure stores it in the system catalog for future execution. You can then later retrieve and execute it.

When you create the procedure with the PROCEDURE command, you specify whether it is a dynamic or static stored procedure; dynamic is the default. When you actually store the procedure, SQLBase also stores the procedure's execution plan.

You can store a procedure under a different name than the one it is created with. For details, read the section *Name* on page *7-4*.

Note: You cannot replace an existing procedure with a procedure that uses the same stored name. You must first use the SQLTalk ERASE command to erase the existing procedure before storing the new one.

You must store a procedure as static if you plan to use it in a trigger.

Use the SQLTalk STORE command to store a procedure. You issue this command at the same time you generate the procedure text with PROCEDURE. For example:

```
STORE WD_PROC
PROCEDURE: WITHDRAW
Parameters
  Number: nAccount
  Number: nAmount
```

```
  Receive Number: nNewBalance
Local Variables
  String: sUpdate
  String: sSelect
Actions
  Set sUpdate = 'UPDATE CHECKING set \
    BALANCE = BALANCE - :nAmount where \
    ACCOUNTNUM =:nAccount'
  Call SqlImmediate(sUpdate)
  Set sSelect = 'SELECT BALANCE from CHECKING \
    where ACCOUNTNUM = :nAccount \
    into :nNewBalance'
  Call SqlImmediate(sSelect)
;
```

Generally, you will want to include the STORE command in a script file and then run the script file.

Executing a procedure

Issuing a PROCEDURE command by itself automatically compiles and executes a procedure. You can also run the SQLTalk PREPARE or RETRIEVE commands in conjunction with the PERFORM command to compile/execute or retrieve/execute the procedure in two separate steps.

To retrieve and execute a stored procedure in one step, use the EXECUTE command. This command accepts input values and retrieves data as well as executes the stored procedure. For example:

```
EXECUTE WD_PROC
\
1,50,,
/
```

Note: Stored commands embedded in procedures can become invalid if their underlying database object changes. However, a stored procedure itself never becomes invalidated.

Runtime Errors

Stored commands embedded in procedures can become invalid if the stored command, or its underlying objects are dropped or altered. In this case, SQLBase still executes the procedure, but issues a runtime error about any missing or altered objects.

Similarly, SQLBase also issues a runtime error if is it is unable to find tables that are referenced in the stored procedure. Note that SQLBase allows users to delete or alter tables that are referenced in existing stored procedures.

Dropping a procedure

To drop a procedure from the database, use the SQLTalk ERASE command. For example:

```
ERASE WD_PROC;
```

Debugging a procedure

Within the procedure, you can use the SAL Trace statement to check the values of individual variables. See the Trace statement documentation for more information on this statement.

The SQLTalk SET and SHOW commands also have TRACE and TRACEFILE options to help trace procedure statements. These are run independently of the PROCEDURE command:

SQLTalk command	Description
SET TRACE ON/OFF	Enables or disables statement tracing.
SET TRACEFILE <filename>/OFF	If this is set to a file name, SQLBase directs statement trace output to a file on the server; an Off value directs the output to the server's Process Activity screen.
SHOW TRACE	Determines whether statement tracing is enabled or disabled.
SHOW TRACEFILE	Determines whether statement trace output is being directed to a file on the server or to the server's Process Activity screen.

For example:

```
SET TRACE ON;
RUN example.sql;
PROCEDURE: WithDraw
Parameters
  Number: nAccount
  Number: nAmount
...
SET TRACE OFF;
```

Security

To grant privileges to other users for stored procedures, use the SQLTalk GRANT EXECUTE command. You can grant either your own privileges to other users, or grant them privileges of their own. To revoke users' privileges, use the REVOKE EXECUTE command.

Read the *SQLTalk Language Reference* for information on these commands.

SAL functionality in SQLBase

You can embed any of the following functions in a procedure. User-defined functions are not supported. Note that while these functions are similar to Team Developer functions, they are SQLBase-specific. You do *not* need Centura's Team Developer program to use these functions. See the Appendix for a complete description and syntax for these functions.

Team Developer system variables (such as SqlDatabase) are not supported. Also, unlike Team Developer, SQLBase procedures are *not* case sensitive.

Team Developer Function	Description
SqlClearImmediate	Disconnects the Sql Handle used by SqlImmediate.
SqlClose	Closes a named cursor.
SqlCommit	Commits the current SQL transaction.
SqlConnect	Connects a Sql Handle to a database.
SqlDisconnect	Disconnects a Sql Handle from a database.
SqlDropStoredCmd	Deletes a stored command or stored procedure.
SqlError	Gets the most recent error code for the specified Sql Handle.
SqlExecute	Executes a SQL statement, stored command, or stored procedure.
SqlExists	Checks if a specified row or rows exist.
SqlFetchNext	Fetches the next row in a result set.
SqlFetchPrevious	Fetches the previous row in a result set.
SqlFetchRow	Fetches a specific row from a result set.
SqlGetErrorPosition	Gets the offset of an error within a SQL statement.

Team Developer Function	Description
SqlGetErrorText	Gets the message text for a SQL error number.
SqlGetModifiedRows	Returns the number of rows changed by an INSERT, UPDATE, or DELETE statement.
SqlGetParameter	Returns a database parameter.
SqlGetParameterAll	Returns a database parameter.
SqlGetResultSetCount	Returns the number of rows in a result set.
SqlGetRollbackFlag	Returns the database rollback flag.
SqlImmediate	Compiles and executes a SQL statement.
SqlOpen	Names a cursor and executes a SQL statement.
SqlPrepare	Compiles a SQL statement or non-stored procedure for execution.
SqlPrepareAndExecute	Compiles and executes a SQL statement or non-stored procedure.
SqlRetrieve	Retrieves a stored command or stored procedure.
SqlSetIsolationLevel	Sets the isolation level.
SqlSetLockTimeout	Sets the timeout period on waiting for a lock.
SqlSetParameter	Sets a database parameter.
SqlSetParameterAll	Sets a database parameter.
SqlSetResultSet	Turns results set mode on and off.
SqlStore	Compiles and stores a command or procedure.

Related SQLTalk commands

Use the following SQLTalk commands to compile, prepare, and execute procedures. For information on these commands, read the *SQLTalk Language Reference.*

Command	Description
ERASE	Erases a stored command/stored procedure.
EXECUTE	Executes a stored command or stored procedure.
PERFORM	Executes either a prepared SQL command/non-stored procedure, or retrieved stored command/stored procedure.
PREPARE	Compiles a SQL command or non-stored procedure.
SET TRACE	Enables or disables statement tracing.
SET TRACEFILE	Directs statement trace output to a file on the server or to the server's Process Activity screen.
SHOW TRACE	Determines whether statement tracing is enabled or disabled.
SHOW TRACEFILE	Determines whether statement trace output is being directed to a file on the server or to the server's Process Activity screen.
STORE	Compiles and stores a command or procedure (and its execution plan) for later execution.

Using SQL/API functions with procedures

You can use the following SQL/API functions to manipulate procedures. For information on these functions, read the *SQL/API Reference Manual.*

SQL/API Function	Description
sqlbnd	Bind input data by name.
sqlbnn	Bind input data by number.
sqlbnv	Get the number of input parameters.
sqlcbv	Clear bind variables.

SQL/API Function	Description
sqlcex	Compile and execute a non-stored command or non-stored procedure.
sqlcom	Compile a non-stored command or non-stored procedure.
sqlcty	Return the command type.
sqldes	Describe output parameters in terms of internal data types and lengths.
sqldii	Describe an INTO variable.
sqldsc	Describe output parameters in terms of external data types and lengths.
sqldst	Drop a stored command or stored procedure.
sqlepo	Retrieve error position.
sqlexe	Execute a command or procedure that has either been previously-compiled or stored.
sqlfet	Fetch next row from result set.
sqlget	Return the statement trace status (enabled/disabled) with the SQLPTRC parameter, and the file name of the trace output file with the SQLPTRF parameter.
sqlnbv	Retrieve number of bind variables.
sqlnii	Get the number of INTO variables.
sqlnsi	Get the number of output parameters.
sqlret	Retrieve a stored command or stored procedure.
sqlset	Enable/disable statement tracing with the SQLPTRC parameter, and redirect trace output to a file or the Process Activity (F2) screen with the SQLPTRF parameter.
sqlssb	Set up buffers for output parameter data.
sqlsto	Store a procedure or SQL command.

Using procedures with Centura Team Developer applications

This section discusses implementation issues to consider when using procedures in Centura Team Developer applications.

Default for Result Sets in Stored Procedures

To emulate scrollable results sets in Team Developer, the default for result sets in stored procedures is turned ON when you issue a Call SqlConnect (hsql). This cursor has results sets turned ON so that scrollable result sets are available when you issue a SqlFetchPrevious. Note that normally in SQLTalk, result sets in stored procedures is turned OFF by default.

Note: If you are NOT using SqlFetchPrevious in your procedures, you can improve performance by explicitly turning results sets OFF in procedures with SqlSetResultSet.

Calling a SQLBase Procedure

To call a SQLBase procedure from Team Developer, use the SAL *SqlRetrieve* call. You must follow these rules:

- All SQLBase procedure parameters must have a representative Team Developer variable/visual object in the bind list (third parameter) o*f SqlRetrieve()*.

- All Receive parameters of a SQLBase procedure that are used as output for the calling Team Developer application must be represented by a Team Developer variable/visual object in the into list (fourth parameter) of *SqlRetrieve*.

Note: There is an exception to these *SqlRetrieve()* rules when using Team Developer List and Combo boxes. These are discussed in the following paragraphs.

For example, assume you are populating two Team Developer data fields, df1 and df2, with the following procedure which returns rows from a SELECT:

```
Procedure: PRODUCTS
Parameters
  Number: nInventory
  Receive String: sName
  Receive Date/Time: dtWhen
Actions
  ...
```

```
Call SqlPrepareAndExecute(hSqlCur1, 'select NAME, WHEN
  from PRODUDCT_INVENTORY \
  where INVENTORY = :nInventory into :sName, \
  :dtWhen')
...
```

Within Team Developer, code the following lines. Notice the bind list.

```
...
Set nInvent = 200
Call SqlRetrieve( hCur1, 'PRODUCTS', ':nInvent, :df1,
  :df2', ':df1, :df2')
Call SqlExecute( hCur1 )
Call SqlFetchNext( hCur1, nInd )
...
```

For table windows, the third parameter for *SalTblPopulate* is passed as a null. This is the same method used when a stored command is the data source for a table window.

```
...
Set nInvent = 200
Call SqlRetrieve( hCur1, 'PRODUCTS', ':nInvent, :df1,
  :df2', ':df1, :df2')
Call SalTblPopulate( tbProducts, hCur1, '',
  TBL_InventNormal
```

For list boxes and combo boxes, both the fourth parameter (into list string) for *SqlRetrieve* and the third parameter for *SalListPopulate* is are passed as nulls. This is the same method used when a stored command is the data source. Secondly, since Team Developer has no method of referencing individual database columns in a list or combo box, you must create dummy variables to represent the procedure Receive parameters within the bind list. Backend result sets do not need to be turned off.

```
...
Window Variables:
  String: sDummy1
  String: sDummy2
  Number: nInvent
...
Message Actions
...
set nInvent = 200
Call SqlRetrive( hCur1, 'PRODUCTS', ':nInvent, :sDummy1,
  :sDummy2', '')
Call SaListPopulate( hWndItem, hCur1, '')
```

To learn more about using SQLBase procedures with Team Developer, run the Team Developer application *sp.app* shipped with SQLBase.

Error handling

By default, SQLBase handles a SQL error by terminating execution of the procedure, and returning an error code to you.

You can override this default SQL error processing using the When SqlError statement. This enables you to specify a *local* error handler as you can in Team Developer. A local error handler is only effective for statements in the same statement block as that in which the error handler is declared.

However, if the When SqlError returns control back to the procedure, it is the procedure's responsibility to check the return from the failed SQL statement and process accordingly. If there is no return from the When SqlError construct, both the control and the SQL error code are immediately returned to the caller.

Unlike Team Developer, procedures do not allow you to specify a global error handler.

Put a When SqlError statement in a procedure's Actions section:

- Before a procedure SAL function.
- At the same indent level as the procedure SAL function.

The following flowchart shows the steps that SQLBase follows when a SQL error occurs during the execution of a procedure.

1. SQLBase looks for When SqlError in the procedure's Actions section.

2. If there is no When SqlError statement, SQLBase breaks the procedure execution, and returns control to the calling program. The error code is returned to the calling program.

3. If there is a When SqlError, SQLBase performs the statements in the When SqlError section.

4. You can use a Return statement to specify that either a TRUE or FALSE value be returned by the procedure SAL function on which the error occurred. If you do not specify a Return statement, the procedure breaks execution, and both control and the error code are returned to the calling program.

   - If the Return statement returns FALSE, FALSE becomes the return value of the failed Sql* function. Procedure execution continues as if the failed SQL statement returned FALSE.

   - If the Return statement returns TRUE, TRUE becomes the return value of the failed Sql* function. Procedure execution continues as if the failed SQL statement returned TRUE.
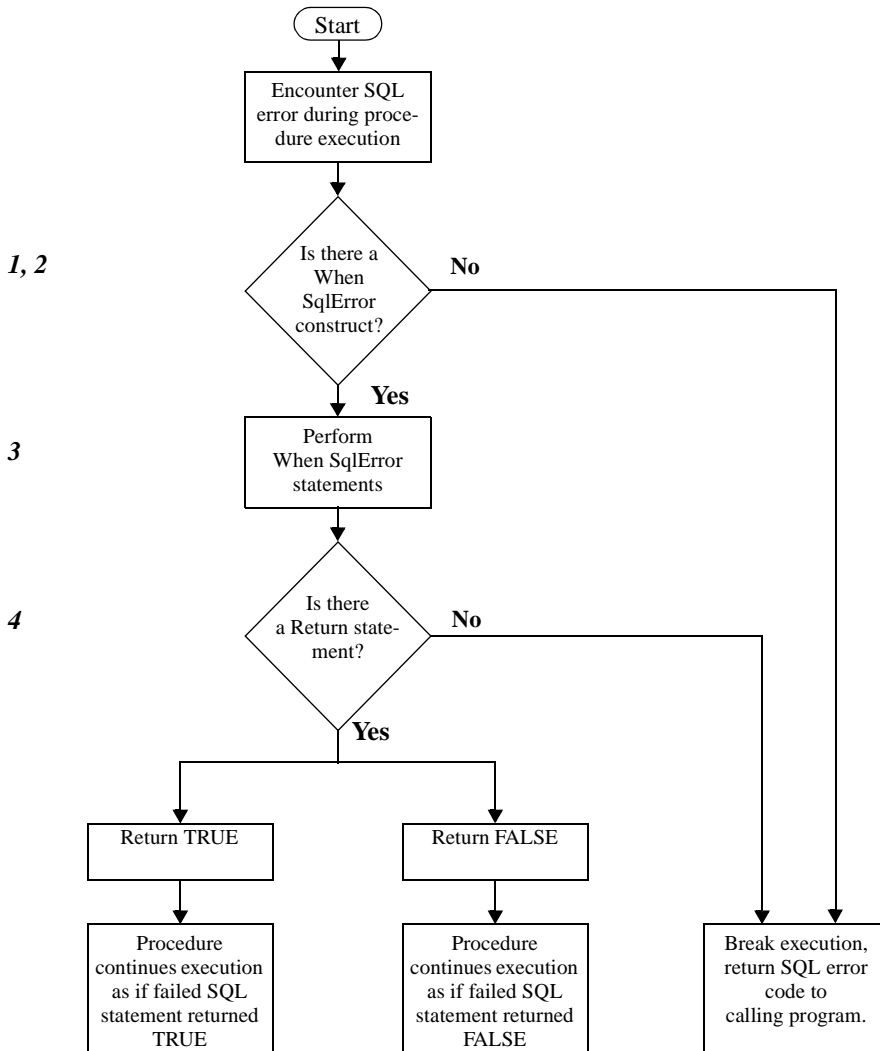
This example uses the tables JR and PRODUDCT_INVENTORY:

```
CREATE TABLE JF 12R,132,13-OCT-1992 (NAME varchar(25),
    INVENTORY decimal (3,0), WHEN date);
INSERT INTO PRODUDCT_INVENTORY values
    (JF 12R,132,13-OCT-1992);
COMMIT;
```

This examples also uses the following stored command INVENTORY_QUERY:

```
STORE INVENTORY_QUERY
    SELECT INVENTORY from PRODUDCT_INVENTORY
    where NAME = :1;
```

In this example, the When SqlError construct tests for two error conditions:

- If the stored command does not exist, error code 207 is returned.
- If the table used in the stored command does not exist, error code 601 is returned.

In this example, error code 601 is returned because the table required for the stored command is dropped prior to procedure execution.

```
    DROP TABLE PRODUDCT_INVENTORY
    PROCEDURE: ILPROC
    Parameters
      String: sName
      Receive Number: nINVENTORY
    Local Variables
      Sql Handle: hSqlCur1
      Number: nInd
      Number: nRcd
      Boolean: bCond
    Actions
      On Procedure Startup
①        When SqlError
            Set nRcd = SqlError(hSqlCur1)
            If nRcd = 207
              Call SqlStore(hSqlCur1, 'INVENTORY_QUERY', \
                'select INVENTORY from PRODUDCT_INVENTORY \
              where NAME = :1 into :2')
              Call SqlCommit(hSqlCur1)
              Call SqlRetrieve(hSqlCur1, 'INVENTORY_QUERY', \
                ':sName', ':nINVENTORY')
              Return TRUE
            Else If nRcd = 601
              Return FALSE
          Call SqlConnect(hSqlCur1)


②        Set bCond = SqlRetrieve(hSqlCur1,\
            'INVENTORY_QUERY',':sName', ':nINVENTORY')
③        If NOT bCond
          Return 6302
      On Procedure Execute
          Call SqlExecute(hSqlCur1)
```

```
       On Procedure Fetch
            If NOT SqlFetchNext(hSqlCur1, nInd)
              Return 1
            Else
              Return 0
     On Procedure Close
            Call SqlDisconnect(hSqlCur1)
\
JF 12R,,
/
```

1. This exception handling routine Returns FALSE because the required table for the stored command was dropped prior to the execution of the procedure. Returning FALSE (as opposed to executing no Return from When SqlError) allows the procedure to provide additional processing, such as returning a user defined error.

2. Since the When SqlError construct returned FALSE, the return value for bCond is set to FALSE.

3. The When SqlError construct sets bCond to FALSE. This returns control back to the calling application with the user defined error 6302 "PRODUDCT_INVENTORY table is missing - see DBA".

Procedure examples

This section is a series of examples that demonstrate the different elements of procedures. They use a table called CHECKING. You can run these and other examples online using the *sp.sql* SQLTalk script, which is provided in the Centura directory with your SQLBase software. In addition, the *sp.app* sample application provided in the Centura directory demonstrates using procedures in Team Developer.

Example 1 - Procedure IF/Else statement

This next example adds an IF/Else statement to the procedure; this checks to see if the balance is negative.

```
STORE WITHDRAW
PROCEDURE: WITHDRAW
Parameters
  Number: nAccount
  Number: nAmount
  Receive Number: nNewBalance
  Receive Boolean: bOverDrawn
Local Variables
```

```
      String: sSelect
      String: sUpdate
    Actions
      Set sSelect = 'SELECT BALANCE from CHECKING \
        where ACCOUNTNUM = :nAccount \
        into :nNewBalance'
      Call SqlImmediate(sSelect)
      Set nNewBalance = nNewBalance - nAmount
      If (nNewBalance < 0)
        Set bOverDrawn = TRUE
      Else
        Set bOverDrawn = FALSE
        Set sUpdate = 'UPDATE CHECKING \
        set BALANCE = BALANCE - :nAmount \
        whereACCOUNTNUM = :nAccount'
        Call SqlImmediate(sUpdate)
    ;

    EXECUTE WITHDRAW
    \
    1,100,,,
    /
```

Example 2- Using SQL handles and ON statements

The next example adds SQL handles and ON statements to the procedure.

```
    STORE WITHDRAW
    PROCEDURE: WITHDRAW
    Parameters
      Number: nAccount
      Number: nAmount
      Receive Number: nNewBalance
      Receive Boolean: bOverDrawn
    Local Variables
      Sql Handle: hSqlSelect
      Sql Handle: hSqlUpdate
      String: sSelect
      String: sUpdate
      Number: nStatus
    Actions
      On Procedure Startup
        Set sSelect = 'SELECT BALANCE from CHECKING \
          where ACCOUNTNUM = :nAccount \
          into :nNewBalance'
        Set sUpdate = 'UPDATE CHECKING \
          set BALANCE = BALANCE - :nAmount \
          whereACCOUNTNUM = :nAccount'
```

```
                    Call SqlConnect(hSqlSelect)
                    Call SqlPrepare(hSqlSelect, sSelect)
                    Call SqlConnect(hSqlUpdate)
                    Call SqlPrepare(hSqlUpdate, sUpdate)
                  On Procedure Execute
                    Call SqlExecute(hSqlSelect)
                    Call SqlFetchNext(hSqlSelect, nStatus)
                    Set nNewBalance = nNewBalance - nAmount
                    If (nNewBalance < 0)
                      Set bOverDrawn = TRUE
                    Else
                      Set bOverDrawn = FALSE
                      Call SqlExecute(hSqlUpdate)
                  On Procedure Close
                    Call SqlDisconnect(hSqlSelect)
                    Call SqlDisconnect(hSqlUpdate)
                ;

                EXECUTE WITHDRAW
                \
                1,100,,,
                /
```

Example 3 - Doing a fetch

This example adds a fetch operation to the procedure.

```
STORE WITHDRAW
PROCEDURE: WITHDRAW
Parameters
  Number: nAccount
  Number: nAmount
  Receive Number: nNewBalance
Local Variables
  Sql Handle: hSqlSelect
  String: sSelect
  Number: nStatus
  Boolean: bEOF
Actions
  On Procedure Startup
    Set sSelect = 'SELECT BALANCE from CHECKING \
      where ACCOUNTNUM = :nAccount \
      into :nNewBalance'
    Call SqlConnect(hSqlSelect)
    Call SqlPrepare(hSqlSelect, sSelect)
  On Procedure Execute
    Call SqlExecute(hSqlSelect)
    ! Internal fetch - column is not returned to the calling
```

```
      ! program since there is an On Procedure Fetch state
      ! which does return values to the calling program.

    Call SqlFetchNext(hSqlSelect, nStatus)
  On Procedure Fetch
    If (nNewBalance > 100)
      Set nNewBalance = nNewBalance * 1.005
      Set nNewBalance = nNewBalance - 100
      Set bEOF = FALSE
    Else
      Set bEOF = TRUE
    Return bEOF
  On Procedure Close
    Call SqlDisconnect(hSqlSelect);

EXECUTE WITHDRAW
\
1,100,,
/
```

Example 4 - Calling a stored procedure from within another procedure

This example shows how one stored procedure can call another stored procedure. The calling stored procedure is DYNAMIC and the called stored procedure is STATIC. Nesting procedures can enhance the modularity of code by creating common routines that perform specialized tasks. These tasks can then be called by any number of different procedures or calling programs.

This example uses the following two tables PRODUCTIVE and RATE:

```
create table PRODUCTIVE
(
NAME   varchar(25),
DEPT   varchar(2),
BUILD  varchar(1),
PRICE  integer
);

insert into PRODUCTIVE values('BM J18','TT','M',66);

create table RATE
(
RATE     varchar(12),
PER_DAY  double precision
);
```

```
insert into RATE values(:1, :2)
\
"LEVEL H",300,
"LEVEL B",190,
"LEVEL T",150,
"LEVEL I",25,
/
```

This is the syntax of the static stored procedure PRODUCT_COUNT, which determines the current population of the PRODUCT table. It is called by the dynamic procedure ADJUST_RATE.

```
STORE PRODUCT_COUNT
PROCEDURE: PRODUCT_COUNT static
Parameters
  Receive Number: nCount
Local Variables
  Sql Handle: hSqlCur1
  Number: nInd
Actions
  Call SqlConnect( hSqlCur1 )
  Call SqlPrepareAndExecute( hSqlCur1, \
    'select count(*) from PRODUCTIVE into :nCount' )
  Call SqlFetchNext(hSqlCur1, nInd)
;
```

The following dynamic stored procedure ADJUST_RATE calls the PRODUCT_COUNT stored procedure. Based on the current PRODUCT population, the DAILY rates are determined.

```
store ADJUST_RATE
procedure: ADJUST_RATE dynamic
Parameters
  Receive String: sRate
  Receive Number: nPerDay
Local Variables
  Sql Handle: hSqlCur1
  Number: nPop
  Number: nInd
  String: sAdjust
  String: sUpdate
  String: sSelect
Actions
  On Procedure Startup
    Call SqlConnect( hSqlCur1 )
        Set nPop = 0
①   Call SqlRetrieve( hSqlCur1, \
```

```
            'PRODUCT_COUNT', ':nPop', ':nPop' )
          Set sSelect = 'Select RATE, PER_DAY from RATE \
              into :sRate, :nPerDay'
        On Procedure Execute
          Call SqlExecute( hSqlCur1 )
②        Call SQLFetchNext(hSQlCurl, nInd)
③        If nPop > 1
            Set sAdjust = 'set PER_DAY = PER_DAY * 1.15'
          Else
            Set sAdjust = 'set PER_DAY = PER_DAY * 1.05'
          Set sUpdate = 'Update RATE ' || sAdjust

④        Call SqlPrepareAndExecute( hSqlCur1, sUpdate )
⑤        Call SqlPrepareAndExecute( hSqlCur1, sSelect )
        On Procedure Fetch
          If NOT SqlFetchNext( hSqlCur1, nInd )
            Return 1
          Else
            Return 0
        On Procedure Close
          Call SqlDisconnect( hSqlCur1 )

      ;

⑥ column 1 width 15;
   execute ADJUST_RATE

   \
   ''
   /
```

1. Retrieve the stored procedure to get the current PRODUCT count. Notice the bind list must include (in proper order) variables which represent all parameters declared in the called stored procedure PRODUCT_COUNT. Secondly, the into list must include variables which map to those Receive parameters of the called procedure that return output to procedure ADJUST_RATE.

2. Fetch a single row value into the *nPop* local variable.

3. Use dynamic SQL to build the update statement based on the PRODUCT population.

4. Update the RATE table.

5. Now select the new rates from the RATE table.

6. SQLTalk requires string columns to be resized.

Triggers

This section provides an overview of triggers which use stored procedures. For detailed information about triggers, see the documentation on the CREATE TRIGGER command in this manual.

What is a trigger?

A *trigger* activates a stored or inline procedure that SQLBase automatically executes when a user attempts to change the data in a table. You create one or more triggers on a table, with each trigger defined to activate on a specific command (an INSERT, UPDATE, or DELETE). Attempting to modify data within the table activates the trigger that corresponds to the command. For details on the trigger execution order before a single data manipulation statement is executed, read the Section *DML Execution Model* in Chapter 1.

Triggers enable you to:

- Implement referential integrity constraints, such as ensuring that a foreign key value matches an existing primary key value.

- Prevent users from making incorrect or inconsistent data changes by ensuring that intended data modifications do not compromise a database's integrity.

- Take action based on the value of a row before or after modification.

- Transfer much of the logic processing to the backend, reducing the amount of work that your application needs to do as well as reducing network traffic.

Creating Triggers

You can only use inline or static stored procedures with triggers. In addition, you must first store the static procedure with the STORE command; a trigger cannot call a non-stored procedure.

Use the SQL CREATE TRIGGER command to create a trigger. You can disable an existing trigger by using the ALTER TRIGGER command. This command causes SQLBase to ignore the trigger when an activating DML statement is issued. With this command, you can also enable a trigger that is currently inactive.

You can easily disable all triggers defined on a table by using the stored procedure \Centura\rep_trig.sql included with SQLBase.

To access the stored procedure, you must have SYSADM authority and run the file REP_TRIG.SQL against the database that contains the triggers you want to enable or disable. This file creates the stored procedure SYSADM.SYSPROC_ALTTABTRIG.

To use the stored procedure, provide the owner and name of the table that contains the trigger and specify whether to enable or disable the triggers in the table. When you

execute the procedure, it retrieves the names of all triggers belonging to the table and enables or disables each trigger one by one. Through the receive parameter, the procedure returns the number of triggers that it processed.

For example, to disable all triggers on table T1 created by USER1, run:

```
EXECUTE SYSPROC_ALTTABTRIG
\
USER1, T1, DISABLE, 0
/
```
To delete a trigger from the system catalog, use DROP TRIGGER.

Note: To see an online triggers tutorial, run the *triggers.sql* script that is installed with SQLBase.

Trigger example

The following example shows how an insert statement can invoke a trigger to insert data into a history table. The trigger calls an inline procedure called *proc_newpres*.

This trigger uses the following PRESIDENT and ELECTION tables:

```
CREATE TABLE PRESIDENT
     (PRES_NAME varchar(20) not null, BIRTH_DATE date,
     YRS_SERV integer, DEATH_AGE integer,
     PARTY varchar (20),STATE_BORN varchar(20));

CREATE TABLE ELECTION (ELECTION_YEAR smallint,
     CANDIDATE varchar(20),VOTES float,
     WINNER_LOSER_INDIC char(1));

CREATE TRIGGER TRG_NEWPRES
     after insert on SYSADM.PRESIDENT
     (execute inline (1792,'Jefferson T',4,'L')

PROCEDURE: PROC_NEWPRES static
  Parameters
    Number: nElecYear
    String: sCandidate
    Number: nVotes
    String: sWinLose
  Local Variables
    Sql Handle: hSqlCur
  Actions
    On Procedure Startup
      Call SqlConnect(hSqlCur)
      Call SqlPrepare(hSqlCur, 'Insert into \
        sysadm.election values \
        (:nElecYear, :sCandidate, :nVotes, :sWinLose)')
```

```
      On Procedure Execute
        Call SqlExecute(hSqlCur)
      On Procedure Close
        Call SqlDisconnect(hSqlCur)
    )
    for each statement;
```

This trigger is invoked when you INSERT into the PRESIDENT table, as in the following example:

```
INSERT into PRESIDENT values ('Jefferson T',
    13-Apr-1743,8,83,'Demo-Rep','Virginia');
```

Security

When a user invokes a trigger, he/she assumes the privileges of the owner of the table on which the trigger is defined. The user invoking the trigger must have privileges to do the DML command that causes the trigger to be activated.

You can only create a trigger which uses a stored procedure under one of the following conditions:

- You have either DBA or SYSADM privileges.
- You are the owner of the stored procedure.
- You have been granted EXECUTE privileges for that stored procedure.

Error handling in triggers

If a trigger calls a stored procedure and the procedure performs validation logic which returns an error code, the trigger returns the error code to the calling SQL statement, which displays it. A procedure's error will "bubble" all the way to the trigger. This means that the error appears no matter how the trigger is invoked.

Chapter 8

External Functions

This chapter describes external functions. It provides the information you need for developing external functions and invoking them from within a SQLBase stored procedure.

The following topics are covered:

- What is an external function?
- How to declare external functions
- Using external data types
- Calling external functions
- Developing external functions
- Modifying external function definitions
- Error handling
- System Catalog tables for external functions
- Scripts and DLLs for external functions
- External function example

What is an External Function?

An external function is a user-defined function that resides in an "external" DLL (Dynamic Link Library) that is invoked from within a SQLBase stored procedure. You can create your own external function in a language of your choice, such as C, C++, and so forth.

You use the CREATE EXTERNAL FUNCTION command to define external functions calls by specifying such information as the function's name, its arguments, DLL where it resides, compiler callstyle, and execution mode.

From within a SQLBase stored procedure, you can use a CALL statement to invoke the external function, or you can embed the function invocation in SAL expressions. On invoking the external function, SQLBase looks for the function's name in the catalog, loads the appropriate DLL that is specified for the function, and then calls the function. Figure 9.1 illustrates how an external function *MyFunc* is invoked from within a stored procedure.

Why use external functions?

The ability to call external functions within SQLBase enhances the power of the SQLBase server. It provides you with the flexibility to extend the functionality of your stored procedures, or add functionality to your existing applications by creating plug and play external components. You can:

- Use existing SAL functions as external functions
- Execute application programs that call C/API functions directly on the server by converting them into external functions.
- Maintain a centralized library of functions that can be used with different applications and at different sites as needed.

Calling external functions from stored procedures extends functionality with no impact on the application or the server. Your components are dynamically plugged in and behave like built-in functions. Using external functions, you achieve maximum flexibility and performance with minimal programming effort.
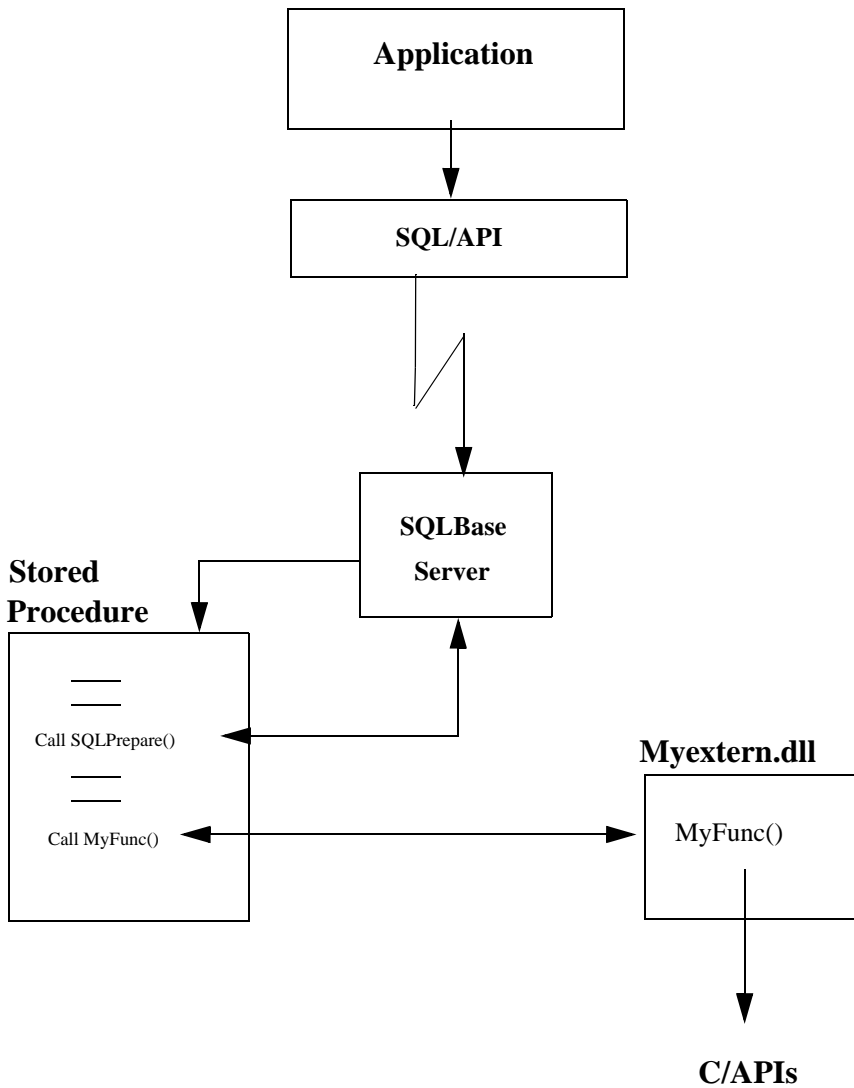
External Function Invocation

Security

If you have DBA authority, you can create, drop, and modify external functions and create synonyms for them. When a user invokes a stored procedure and it calls an external function, the user must have privileges to execute the external function. You, as the creator of the external function, or another DBA, can **grant execute privileges** to other users so they can execute external functions.

If a user is granted **execute with creator privileges** on a procedure that calls external functions, then the user does not need execute privileges on any external function invoked within the procedure. Only the CREATOR of the procedure needs to have execute privileges on the external function.

If the user is granted **execute with grantee privileges** on a stored procedure, the user must also have execute privileges on the external functions invoked within the procedure. For details on setting up security for external functions, see the *Database Administrator's Guide*.

SQLBase checks for privileges on external functions at procedure compile and retrieval time.

How to declare external functions

In order for SQLBase to recognize an external function in your stored procedure, you must declare the external function with the CREATE EXTERNAL FUNCTION command. The full syntax for this command is described in Chapter 3.

This example shows the use of an external function named *myfunc* in your stored procedure. To make this function known to SQLBase, it is declared with the CREATE EXTERNAL FUNCTION command:

```
CREATE EXTERNAL FUNCTION MYFUNC
  Parameters (int, lpint)
  Returns ()
  Callstyle CDECL
  Library myfunc.dll
  Execute in same thread;
```

Note that **function name** and **library name** are mandatory. If your function uses parameters, you can optionally specify the external data types for the parameters and if the function returns a value, you can optionally specify the external data type for the return value. For details, read *Using external data types* on page *8-10*.

After the function is declared, *myfunc* is then called within stored procedure *P1* as shown in the following example:

```
PREPARE
  Procedure P1
```

```
      local variables
        receive number: n1
        receive number: n2
      Actions
        call myfunc(n1,n2)
    ;
```

Function name

Function name specifies the name of the function as known and referenced within SQLBase. Function names are similar to other database object names, except they can be up to 64 characters in length.

If you specify the function name without quotes, you must begin the function name with an alpha, "a...z" character. By default, the characters are uppercased.

For example if an external function is named **myfunc** and is not enclosed in double quotes, SQLBase converts the name to uppercase, as in **MYFUNC**. When this external function is called in a stored procedure, **MYFUNC** must be specified in upper case.

You must specify a function name in double quotes if:

- the name contains special characters
- the name starts with an alpha character
- the case of the name is to be preserved

Note that if you enclose the name in double quotes, the case of the name is preserved. For example, if you want the external function name to remain in lower case, you can specify the external function name in double quotes, as in "myfunc".

If you do not provide the external name clause, the function name is also used to specify the external name of the function in the library.

Naming Restrictions

Please note the following restrictions when choosing a function name.

- Function names cannot be the same as procedure names and vice versa.
- Functions names cannot be the same name used in any of the SQLBase aggregate functions (for example, min, max, avg, etc., or any functions beginning with the @ symbol, such as @ASIN, @ATAN, @CHAR, etc.)
- Function names cannot begin with **SQL**.
- If the external name is not used in the function definition, then the function name must match the exported name in the DLL.
- If the external name is used in the function definition, then the external name must match the exported name in the DLL.

Note: If you are using WINAPI functions, check the exported names for the function. We recommend that you use the external name clause to make the function name match the exported name. See the examples under *External Name* on page *8-7* for details.

Library

Library is the file specification of the dynamic linked library (DLL) in which the function resides. SQLBase checks for the existence of the library at function invocation time, rather than function creation time.

You must provide a fully qualified path name for the file, or else be sure the PATH environment variable is set to point to the location of the file in your operating system.

Note: The directory from which SQLBase executes is considered the current working directory.

Specify the library name as a string with up to 254 characters. You can include special characters in the string. If the library name contains spaces, you must delimit the name in single quotes (for example, 'lib name').

You may be specifying the name of a DLL provided for Microsoft Window API functions. The DDLs are typically stored in the system directory.

Parameters and return data types

When you create an external function, you must specify external data types for any parameters and return values used in the function. If there are no parameters for the external function, omit the PARAMETERS clause, or provide empty parentheses **( )** in the declaration. Similarly, If there is no return type from the external function, omit the RETURNS clause, or provide empty parentheses **( )** in the declaration.

The data type for parameters and returns tells SQLBase the format (both size and pass by value/reference) to use when passing data to the external function and the format to expect when receiving data from the function.

The external type typically corresponds to a standard Microsoft data type. There are some data types that do not correspond to any Microsoft data type. Read *Non-Microsoft data types* on page *8-16* for details.

Once the external function is defined with the correct parameter and return types, SQLBase automatically converts the stored procedure data types into the external data representation.

Declaring External Data types

You specify parameter and return data types in the CREATE EXTERNAL FUNCTION command using the following format:

```
CREATE EXTERNAL FUNCTION MYFUNC
  Parameters (int, lpint)
  Returns (int)
  Library myfunc.dll
  Execute in same thread;
```

In the PARAMETERS clause, you specify receive data types, which are passed to functions by reference (pass by reference). Typically the external data types for receive parameters are prefixed by **LP** (LPINT, LPWORD, etc.)

In the RETURNS clause, you can only specify data types that are passed to functions by value (pass by value). Examples of external data types used to pass parameters by value are INT, LONG, CHAR, etc.

For a list of external data types used to pass parameters by value and by reference, read *Using external data types* on page *8-10*.

Parameters and return values must be compatible in size and type to the function prototype in the DLL.

Note: SQLBase requires the function definition that you create with CREATE EXTERNAL FUNCTION to push parameters on to the stack before calling the function and to read the return value provided by the function. If the parameters are not specified properly, this will cause stack corruption which can result in server failure.

For details on external function calls, read *Calling External Functions* on page *8-17*.

External Name

External name is an optional clause to specify the name of the function in the specified dynamic link library (DLL). Defining an external name enables a function name referenced in the stored procedure to be different from the name used to reference the same function in the DLL.

Specify the external name as a string with up to 254 characters. You can include special characters in the string. The external name is case-sensitive and must be identical to the exported function name in the DLL.

Examples

You may need to define an external name to:

- indicate the function is used by more than one stored procedure or application

  For example, you may want to provide an external name that clearly identifies what stored procedure uses the function.

- make the external name match the exported name of the function assigned by your operating system.

  For example, in 16-bit systems the exported name of a WINAPI function is uppercase. To keep the call for the function name in the stored procedure the same as the API call, you would use the external name clause and make the external name match the exported name as in the following:

```
create external function "SendMessage"

...

library USER.EXE

external name SENDMESSAGE

...;
```

- indicate the correct version of the API function to use.

  For example, in 32-bit systems, WINAPI functions have unicode (double byte character) support which means that WINAPI functions have different internal implementations depending on the character set that is used.

  The version of the function that supports the ASCII character set has an **A** appended to it, while the one that supports the double byte character set has a **W** appended to it. The two external names for the SendMessage function, are **SendMessageA** and **SendMessageW**. If you want to indicate SendMessageA as the version to use, the external function definition is:

```
create external function "SendMessage"
parameters (...)
returns(...)
library USER32.DLL
external name SendMessageA
....;
```

Note: Normally, the compiler converts a call to the correct version of the function. However, external function calls are made without the use of a compiler; hence, you must provide the correct version name.

Callstyle

Depending on the platform you are using, you must specify a callstyle which determines how SQLBase invokes your external function. SQLBase manipulates the call stack based on the callstyle that you have defined for the external function.

Note: Be sure to specify the correct callstyle for your platform in the external function definition. An incorrect callstyle will corrupt the call stack and result in server failure.

Win16 Platforms

On 16-bit platforms, there are two available call styles:

- pascal - the compiler pushes parameters from left to right and the callee pops the stack before return. This callstyle applies to all Windows API calls.

- cdecl - the compiler pushes parameters from right to left and the callee pops the stack after return from the called function. This is the default compiler call style for 16-bit and 32-bit compilers.

Win32 Platforms

On 32-bit platforms, there are two available call styles:

- stdcall - the compiler pushes parameters from right to left and the callee pops the stack before return. This is the default for all 32-bit Windows API calls.

- cdecl - the compiler pushes parameters from right to left and the callee pops the stack after return from the called function. This is the default compiler call style for 16-bit and 32-bit compilers.

Execution Mode

Execute In specifies the execution mode to use for your platform. If you are using a 16-bit platform, you are not required to specify this clause. SQLBase sets 16-bit platforms for **same thread** mode which is the only mode allowed on 16-bit platforms. If you specify **separate process mode** on a 16-bit platform, you will receive a parsetime error.

If you are using a 32-bit platform, you can change the default mode **separate process** setting by specifying **same thread** mode. Read *Choosing an Execution Mode for Win32* on page *8-20* for details on using execution modes.

Using external data types

When you declare an external function, you specify the parameters to the external function and return type from the external function.

SQLBase uses the external data type that you specify in the CREATE EXTERNAL FUNCTION command to format the actual parameters and return values in the form expected by the external function.

SQLBase automatically converts the stored procedure data types into the external data representation.

Parameters and External Data types

You can pass parameters by reference or by value. The external data type defines whether the parameter is passed by value or by reference.

Pass by reference

Receive data types are passed to external functions by reference. This means that the called function has access to the original value; the called function can change the original value. Any change in value made to the data type within the external function is reflected in SQLBase when the parameter is returned. Typically, the names for the receive data types start with "LP" which means "Long Pointer" (for example, LPINT).

Pass by value

Return values are passed to external functions by value. This means that the called function only has access to a copy of the value; the called function can only change the copy of the value. You can identify return value data types because they do not have the prefix "LP" (for example, INT).

Providing external data types

SQLBase uses the external data type to allocate bytes on the stack when an application calls the function in the DLL. If there are parameters for the external function and/or the function has a return value, you must specify an external data type for each parameter and/ or return value that represents the number of bytes that the function expects.

The external data types for Number and Date/Time are easier to understand because they are fixed-length. The external data types for strings are more complex because they are variable in length. Detail on strings are described in the section "String Data *String data type* on page *8-12*.

To choose an external data type for a parameter or return value, you need to also know the external data types that are available for each SQLBase internal data type

(NUMBER, BOOLEAN, etc). This information is provided in the sections that follow. Note that the external data types available are:

- Standard Microsoft Windows and C scalar data types such as LONG, INT, DWORD, and HWND.
- External SAL data types such as HSTRING
- Structures with one or more of the above data types such as NUMBER and DATETIME.

Each section also indicates those external data types used to pass parameters by value and those used to pass parameters by reference. The names of external data types are UPPERCASE.

Numeric and boolean data types

Specify one of these external data types when you pass a Number or Boolean internal data type:

Note: By specifying an external data type that is prefixed by LP (such as, LPINT, LPWORD, etc.,) you indicate the parameter is passed by reference.

External Datatypes (Passed By Value)	Corresponding C scalar data type	External Data types (Passed By Reference)	Corresponding C scalar data type
BYTE	unsigned char	LPBYTE	unsigned char*
CHAR	char	LPCHAR	char*
DOUBLE	double	LPDOUBLE	double*
DWORD	unsigned long	LPDWORD	unsigned long*
FLOAT	float	LPFLOAT	float*
INT	int	LPINT	int*
UINT	unsigned int	LPUINT	unsigned int*
LONG	signed long	LPLONG	signed long*
WORD	unsigned short	LPWORD	unsigned short*
BOOL	int	LPBOOL	int*
NUMBER	SQLBase internal representation for numeric types.	LPNUMBER	SQLbase internal representation for numeric types.

External Datatypes (Passed By Value)	Corresponding C scalar data type	External Data types (Passed By Reference)	Corresponding C scalar data type
LPARAM	unsigned int		
WPARAM	long		

The following rules apply when you specify external data types for an internal numeric or boolean data type.

- *NUMBER and LPNUMBER* are non-Microsoft data types. Both are SQLBase internal representation for numeric types.The definitions for them are defined in either SQL.H included with SQLBase or SWTYPE.H included with Centura.

- NUMBER data type consists of two fields, a 1-byte length field and a 12-byte character array containing the internal representation of the number.

- LPNUMBER is a pointer to the NUMBER data type

- NUMBER and LPNUMBER are used in only two cases:

  - When calling SAL functions within SQLBase that use these data types. Because a script is provided to create all external function definitions for SAL functions, you will never need to specify these data types.

  - When calling functions that in turn call SQL/API functions that use the internal numeric representation. See the *SQL/API Programming Reference* manual for those functions that use internal representation.

- Memory representation for such datatypes as INT, UNIT, LPINT, LPUINT, may differ between 16-bit and 32-bit platforms. To obtain the precise memory representation of a specific data type and to resolve memory issues, consult your C Compiler documentation for your platform.

String data type

Strings are buffers that can contain text or binary data. Text is null terminated. The most important thing about the string data type is that you must be aware of its length.

When a string data type (other than HSTRING and LPHSTRING) is passed to an external function, SQLBase makes a copy of the string and passes a pointer (string address) to that copy on the stack. In case the string is passed by value, that copy is discarded on return. If passed by reference, the string is copied back to its original location. Note that SQLBase only passes to the stack the address or pointer to the string even if the string is passed by value.

Specify one of these external data types when you pass a string internal data type:

Note: By specifying an external data type that is prefixed by LP (such as, LPSTR, LPBINARY, etc.) you indicate that the parameter for the data type is passed by reference.

External Data types (Passed By Value)	Corresponding C scalar data type	External Data types (Passed By Reference)	Corresponding C scalar data type
LPCSTR	char * (null terminated)	LPSTR	char* (null terminated)
BINARY	struct {char*; long;}	LPBINARY	pointer to struct {char*;long;}
HSTRING	Centura Builder handle	LPHSTRING	Centura Builder handle
LPCVOID	binary data	LPVOID	binary data

The following rules apply when you specify external data types for the internal string.

- LPSTR data type is treated as a pointer to a null terminated string. When a string is passed as LPSTR, the external function can modify the string up to the maximum buffer size allocated for the string. The string may grow in size as long as the new length does not exceed the buffer allocated for the string.

    You can allocate buffers by calling SalStrSetBufferLength() or malloc(). SQLBase looks for the null terminator on return and copies the data up to the null terminator back into buffer space. If SQLBase does not find a null terminator within buffer size bytes, an error is generated.

- LPVOID is treated as binary data. In this case, on return from the external function, SQLBase assumes that the string length is unchanged and copies any data up to the original length back into its buffer space.

- If you want an external function to pass strings as binary data and include length information, specify the external data type for the parameter as BINARY or LPBINARY. Note that these data types are not standard Microsoft data types.

    BINARY data type is defined in SQL.H. LPBINARY is a pointer to BINARY. Its structure contains a 4-byte string pointer and a 4-byte string length. Also see the following section *Manipulating the Binary Data type* for available macros used to manipulate the BINARY and LBINARY data types.

- When a string is passed by reference with LPBINARY, the external function may allocate a string in its own memory and pass that string back. On return

from the function, SQLBase copies into its own buffer space, the data pointed by the string pointer up to a length defined by the string length field.

- HSTRING and LPHSTRING are data types used only by Centura Builder on 32-bit platforms to call SAL functions. Since SQLBase provides a script to create all function definitions for SAL functions, there is no need to create a function that uses HSTRING or LPHSTRING.

Manipulating the Binary Data Type

Four macros are provided in SQL.H to manipulate the BINARY datatype. They are:

- BINARY_GET_LENGTH (BINARY) - Get the length of the string
- BINARY_GET_BUFFER (BINARY) - Get the pointer to the string
- BINARY_SET_LENGTH (BINARY, LENGTH) - Put length into binary
- BINARY_SET_BUFFER (BINARY, STRING) - Put pointer to string into binary

Date/Time data types

Specify one of these external data types when you pass a date/time internal data type:

Note: By specifying an external data type that is prefixed by LP (such as, LPDATETIME, LPSTR, etc.) you indicate that the parameter for the data type is passed by reference.

External Datatypes (Passed By Value)	Corresponding C scalar data type	External Data types (Passed By Reference)	Corresponding C scalar data type
DATETIME	SQLBase internal date/time representation.	LPDATETIME	pointer to SQLBase internal date/time representation.
LPCSTR	char* (null terminated)	LPSTR	char* (null terminated)

The following rules apply when you specify external data types for each date/time data type.

- The external data type for Date/Time can also be a null terminated date string. In this case, SQLBase converts the data type to ASCII format.
- When Date/Time is passed by value with either the DATETIME or LPCSTR data types, any changes made to the string within the external function are not visible on return from the function.

- DATETIME and LPDATETIME are non-standard Microsoft data types and are SQLBase internal representations for date/time types. The definitions for them are provided in either SQL.H included with SQLBase or SWTYPE.H included with Centura.

- DATETIME consists of two fields, a 1-byte length field and a 12-byte character array containing the internal representation of the number.

- LPDATETIME is a pointer to the DATETIME data type

- DATETIME and LPDATETIME are used in only two cases:

  - When calling SAL functions within SQLBase that use these data types. Because a script is provided to create all external function definitions for SAL functions, you will never need to specify these data types.

  - When calling functions that in turn call SQL/API functions that use the internal numeric representation. See the *SQL/API Reference Manual* for those functions that use internal representation

Other external data types

This section contains information on SAL window and file function data types and non-Microsoft data types.

SAL Window and File function data types

The table below lists the external data types that have been added to support SAL Window and File functions. You can use these data types for other external functions that use window and file handles.

Internal data type	External data type	Corresponding C scalar data type
Window Handle	HWND	Microsoft data type HWND
	LPHWND	pointer to Microsoft data type HWND
File Handle	HFFILE	FILE
	LPHFFILE	FILE*

The following rules apply when you specify external data types for the window handle and file handle internal data types:

- The HWND and LPHWND external data types are used for storing window handles and support the SAL window manipulation function. If these data types are used in the parameter section of the procedure (that is, input/output),

you should bind to the variable using the program data type SQLPNUM. The same holds for set select buffer.

- Use the keyword hWndNull to check whether a window handle is null. This keyword is similar to STRING_NULL, NUMBER_NULL and DATETIME_NULL.

- The HFFILE and LPHFFILE external data types are used for storing file handles and support the SAL file manipulation function and C Run Time file manipulation functions. The file handle data type can only be used in the local variable section; that is, it cannot be used to pass input/output in a procedure.

- HWND, LPHWND, HFFILE, and LPHFFILE data types can only be used for storing window and file handles. These data types cannot be included in any arithmetic operations.

Non-Microsoft data types

The non-microsoft data types **number** and **datetime** are defined in SQL.H. You can use each data type within the external function. Please refer to SQL.H for the exact structure of these data types.

You can use the following macros with the data types **number** and **datetime**, which are defined in SQL.H.

- NUMBER_IS_NULL(number) - returns TRUE if number is null, FALSE otherwise

- DATETIME_IS NULL(datetime) - returns TRUE if datetime is null, FALSE otherwise

- NUMBER_SET_NULL(number) - sets a number type to null

- DATETIME_SET_NULL(datetime) - sets a datetime type to null

Note: These macros cannot be used with NUMBER and DATETIME data types.

Calling External Functions

This section provides a list of tasks you may need to perform before your external function is ready to be called from within a stored procedure. Review this list to see if you have met the basic requirements and any additional ones that may apply to your environment.

To set up SQLBase to call external functions, you need to:

1. Provide the CALL command for the external function within the stored procedure. Read *Specifying external functions within stored procedures* on page *8-19*.

2. Set up the Dynamic Linked Library (DLL) to store the external function.

   You must provide a fully qualified path name for the file, or else be sure the PATH environment variable is set to point to the location of the file in your operating system.

3. Optionally, specify the DLLs for loading at SQLBase server start up time.

   Note that this procedure is highly recommended and is mandatory if the DLL uses global variables that can be accessed from different functions or from multiple invocations of a function.

4. Define the external functions in the SQLBase server database using the CREATE EXTERNAL FUNCTION command. Read *How to declare external functions* on page *8-4*.

5. Run the SQLSAL32.SQL script if you are using SAL functions as external functions. Read *Calling SAL functions as external functions* on page *8-20*.

6. Set up user privileges to the functions. Read the *Database Administrator's Guide* for details on setting up security for external functions.

7. Set up synonyms. Read the *Database Administrator's Guide* for details on setting up synonyms and Chapter 3 of this manual for details on the CREATE SYNONYM command.

8. Make sure the function name is exported from the DLL. The exported name should be identical to either the external function name or the name in the external name clause.

   To export the function, you can use the EXPORTS keyword in the .DEF file, the /EXPORTS option when linking the DLL, or the keywords _declspec (dllexport) when declaring the function. Please read you compiler/linker documentation for more details.

Building a 16-bit DLL

If you are using the Microsoft Visual C++ version 1.52 to build a 16-bit DLL which contains the external functions, make sure to:

1. Select Project Compiler Options instead of using the Default Project Options.

2. Select LARGE memory model.

3. Set the segment setup option to:

    ```
    SS != DS, DS loaded on function entry
    ```

Note: This is the same option as /ALu when the 16-bit DLL is built from the command line.

4. In the Windows Prolog/Epilog section, choose None (that is, no prolog/epilog code optimization) option.

Note: This is the same as NOT giving the /GD option when the 16-bit DLL is built from the command line.

Pre-loading DLLs

By default, SQL loads the DLL at function call time by calling the Microsoft API function Load Library. Because of the enormous overhead involved in making this call, (especially in the case of large DLLs or DLLs that cause more DLLs to be loaded), SQLBase allows you to specify pre-loading DLLs at server startup time. This saves overhead and guarantees that the DLL is loaded as long as the server is running.

Note: If you have a DLL that uses global variables that can be accessed from different functions or from multiple invocations of a function, you must load the DLL at server start up.

To set up the DLLs for pre-loading on the server, add the EXTDLL=*dllname* keyword to the dbwservr or dbntsrv (whichever applies to your environment) server section of the SQL.INI file. For example, if you are loading DLLs for WINAPIs in a 32-bit platform, you would specify:

```
EXTDLL=USER32.DLL
```

If you want to pre-load more than one DLL, you can specify the parameter multiple times within the server section. If the DLL name is not qualified, the Operating System uses the path environment variable to locate the DLL.

At server startup time, the server screen displays the following message after each DLL is loaded:

```
Loaded External Library <dllname>
```
If there is an error loading the DLL, you will see the following message:

```
Load of External Library <dllname> failed with error <errornum>
```
The errornum is the error code returned by the Microsoft API call LoadLibrary. You must look up the error code in the Microsoft function reference.

DLLs and global variables

If the DLL uses global variables that can be accessed from different functions or from invocations of a function, be sure to load the DLL at server start up.

If you are using SAL functions, you are advised to preload the DLL in which the function resides (currently CDLLI10.DLL) since the size of this DLL is large and results in a costly load operation.

For example, if you want to use the function SalNumberRandom to return a different random number for each invocation, the CDLLI10.DLL must be pre-loaded. This is because the random number generator is initialized by calling SalNumberRandInit with a seed. This seed is maintained as a global variable and is used for each invocation of SalNumberRandom.

For details on using CDLLI10.DLL read the manual, *Developing with Centura Builder*.

Specifying external functions within stored procedures

You can directly invoke an external function using the CALL statement described in *Chapter 7, Procedures and Triggers*. For example:

```
CALL extfunc()
```
You can also embed external functions in SAL expressions. For example:

```
set n = m + extfun()
```
or,

```
if (extfun())
```

Function Names Used for Invocation

Calls to external functions in stored procedures are case sensitive. Any reference to an external function must be identical to the name of the external function or synonym. For details on naming external functions, read *Function name* on page *8-5*.

You cannot use qualified names to invoke functions. Hence, if a function creator grants execute privilege to another user, the creator must create a public or private synonym for the function.

Specifying external functions for export to the DLL

When you specify the external functions to be exported, each name must be identical to either the external name of the function if specified, or the function name (including case) if the external name is not specified.

If you are not specifying an external name and the exported name has lower case characters, you must enclose the function name in double quotes (" ") to be sure of the case sensitivity.

Calling SAL functions as external functions

If you are using a 32-bit platform, you can invoke SAL functions as external functions. Included in your SQLBase package is a script called SQLSAL32.SQL that creates external function definitions for the SAL functions. For details on the SAL functions included in the DLL, see the *Centura Function Reference* manual.

To call SAL functions in your stored procedure:

1. Run the SQLSAL32.SQL script. The script contains unqualified names to load into the DLLS.

2. Add the location of the SAL DLLs to the path environment.

3. If desired, pre-load the DLL for the SAL functions. Read *Pre-loading DLLs* on page *8-18* for details.

Note: To avoid high overhead since the SAL DLL is quite large, we recommend that you pre-load the SAL DLL.

Developing external functions

This section describes issues you need to consider when developing external functions on Windows 16-bit and 32-bit platforms and invoking the functions within SQLBase.

Choosing an Execution Mode for Win32

When you declare an external function and are using a 32-bit platform, you have the option of invoking the external function on a separate OS process or in the same server database thread as the invoking stored procedure. Read *How to declare external functions* on page *8-4* for syntax details.

You must choose the separate OS process if you are using external functions that require C/API calls

> **Note:** C/API calls cannot be invoked within external functions that execute in the same thread as the calling procedure.

Otherwise, you will want to consider what impact the execution mode has on the called function.

Once an external function is invoked within SQLBase, the server relinquishes its execution control over the code to the external function. If the external function is invoked on the same database thread as the SQLBase server, this action can have adverse impact on the server's ability to continue to carry through with the stored procedure.

For example, if the external function performs I/O's or is connected to another server, it may be locked out from performing its task, thereby blocking the SQLBase Server. In addition the external function would continue executing in the same process space as SQLBase and could corrupt server memory.

In using the same thread execution mode, you need to consider what task the function performs, its resources, and volume of activity. Small, self contained functions that do not perform I/O or C/API calls can execute in the same thread successfully. For example, SAL string manipulation functions (that are definable as external functions) are those that perform well under the same thread mode.

For functions that perform I/O and C/API calls, executing in a separate process is a way to prevent the server from blockage and memory corruption. When executing in this mode, C/API calls can be invoked from within the function and executed in SQLBase. When a function executes in a separate process there is no chance for the function to corrupt server memory. Also I/Os are performed in a separate process and cannot block the server.

Executing in separate process

By defining external functions to execute in a separate OS process mode within a stored procedure or application, all processes come under the control of the external function daemon (EFDaemon).

The SQLBase server process sends messages to the EFDaemon and informs the daemon when native OS shared memory is implemented. The EFDaemon communicates with each external function process known as EFHost through messages and the shared memory block created by SQLBase for each external function.

Legend:

    ⟷   Message passing

    ⟵------▶   Shared memory Access
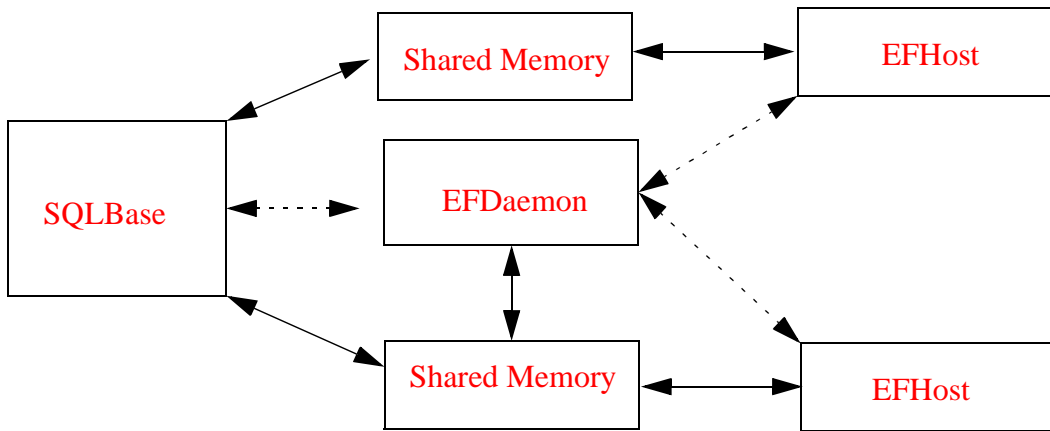
Developing External Functions for Concurrent Execution

When developing external functions for concurrent execution, note that the scope of execution of an external function is the duration of the stored procedure execution. This means that a EFHost process is assigned to a stored procedure on the first invocation of an external function that requires a separate process execution. From then on, all subsequent calls to external functions from the same stored procedure are routed to the same EFHost process. This behavior has the following implications that you need to keep in mind when designing an external application:

- Multiple external functions share the same Dynamic Link Library (DLL) and the DLL is loaded only once.
- Multiple external functions can share global variables.
- All DLLs are unloaded when the stored procedure is closed by SQLBase.
- Nested procedure(s) are executed in their own scope.

Checking external function processes

While a stored procedure or external application is running, you can check each external function process through the EFDaemon window which displays automatically upon function execution. On the window, a menu item displays the status of currently active EFHost processes. For each EFHost process, the following information is displayed:

Field	Description
Cursor Number	Unique number of SQLBase thread. If the stored procedure is called directly, the cursor number corresponds to the cursor number displayed in the Process Activity window in the main database server status window. If the stored procedure is called indirectly, for example, through a trigger, then the cursor number is different than the one shown in the server window.
External function host number	Serial number of the EFHost process.
EFHost	The possible values displayed under this heading are: Idle This indicates that the EFHost process is not executing any external function calls; that is, it is not communicating the EFDaemon. Busy This indicates that the EFHost process is busy executing an external function call. Waiting This indicates that the EFHost process is waiting in-between calls to external functions; that is, waiting to serve. Error This indicates the EFHost process experienced some error while executing the external function call.
Function Name	External function name. Not currently shown in window.
External function DLL name	DLL containing the external function. Not currently shown in window.

Testing and debugging external functions

If you are using a 32-bit platform, we recommend testing the external function using the separate process mode. Read *Choosing an Execution Mode for Win32* on page *8-20* for details. Once the function has been sufficiently tested without problems, you can change the execution mode to server thread using the ALTER FUNCTION command.

Note: You should always use the separate process model for external functions that perform blocking operations (such as file I/O), C/API calls, and any CPU memory intensive operations.

Before inserting the function into a DLL and defining the function to SQLBase, be sure to use standard debugging techniques to ensure the function is bug free. You may want to execute the function as a front end application and apply the debugging techniques of your choice to the application. In a test environment, you may also want to set up the compile of your function to display debugging information and then bring up SQLBase from within a symbolic debugging facility.

Modifying external function definitions

Once you have created an external function, you can alter its definition, or delete it.

Alter external function

You use the ALTER EXTERNAL FUNCTION command to alter those properties of an external function that do not invalidate dependent objects. Those properties are library name, external name, callstyle, and execution mode. You must have DBA authority to execute this command. For details, read the section on ALTER EXTERNAL FUNCTION in Chapter 3.

Drop external function

You use the DROP EXTERNAL FUNCTION command to delete the specified external function from the database. An external function can only be dropped by its creator or by a user with SYSADM or DBA authority.

The command presents three options that determine the behavior that occurs when an external function is dropped. You can:

- prevent the external function from being dropped if a stored procedure refers to the function.
- specify that all stored procedures that call the external function also be dropped.

• specify the external function be dropped and all stored procedures that refer to the function be invalidated.

A system catalog table, SYSDEPENDENCIES, maintains dependencies between dependent objects and determinant objects. The SYSDEPENDENCIES table contains one row for each dependency between a stored procedure and an external function.

For details, read the section on the DROP EXTERNAL FUNCTION command in Chapter 3. For details on the SYSDPENDENCIES tables, refer to *Appendix A*, *System Catalog Tables* of the *Database Administrator's Guide*.

Error Handling

By default, errors encountered when an external function is executed within a stored procedure terminates the procedure and returns an error code to you. For details on stored procedure error handling, read *Error Handling* in *Chapter 7, Procedures and Triggers*.

Errors specific to external functions are included in the ERROR.SQL file. The file includes the exact error message, the reason, and the remedy. You can identify errors specific to external functions in this file from the **EXF** identifier. For example:

```
12502 EXF GPA Cannot get address for external function <name>
      Reason: An attempt to get the address for an
externalfunction failed.
      Remedy: Check to make sure that the function exists and/or
its ordinal number is correct.
```

Exception Handling

If you are using a 32-bit platform, SQLBase identifies the following exceptions if they occur in the external function.

• Bad memory access

• Floating point underflow

• Floating point overflow

• Floating point divide by zero

• Integer overflow

• Integer divide by zero

In the 16-bit platform, such exceptions will result in server shut down.

System Catalog tables for external functions

SQLBase provides and maintains system catalogs, a set of tables owned by the SYSADM that contain information about objects in the database. Following are the tables that are specific to external functions. For details on each of these tables, refer to *Appendix A, System Catalog Tables* in the *Database Administrator's Guide*.

Table Name	Brief Description
SYSDEPENDENCIES	Lists each dependency between a stored procedure and an external function.
SYSEXTFUN	Lists all declared external functions.
SYSEXTPARAMS	Lists each parameter of an external function.
SYSOBAUTH	Lists each user who is granted execute privilege on an external function.
SYSOBJSYN	Lists each synonym created for an external function.

SQLBase-supplied scripts and DLLs

This section describes the external function scripts and DLLs that are supplied with SQLBase. You can use these scripts and DLLs to invoke SAL and C Run Time functions.

Scripts and DLLs for 32-bit systems

- SQLSAL32.SQL

  Script that contains the definitions for SAL functions. If you want to use SAL functions, be sure the Centura runtime DLLs are installed on your system.

- SQLCRT32.SQL

  Script that contains the definitions for the 32-bit C Run Time Library functions.

- SQLCRT32.DLL

  Because it is not possible for SQLBase to directly call the C Run Time Library function *malloc*(), this library contains a wrapper *malloc*() function that turns around and invokes the C Run Time version. The source, make, and project files are provided if you want to add any functions not already included with the C Run Time version. They are:

  - SQLCRT32.C (source file)

- SQLCRT32.MAK (make file)

- SQLCRT32.MDP (project file)

Note that the project was built using Microsoft Visual C++ 4.0.

- MSVCRT40.DLL

Contains the C Run Time Library functions and is a redistributable DLL provided by Microsoft for the 32-bit system.

Scripts and DLLs for 16-bit systems

- SQLCRT16.SQL

Script that contains the definitions for the 16-bit C Run Time Library functions.

- SQLCRT16.DLL

Library of C Run Time functions. The source, make, and def files are provided if you want to add any functions not already included with the C Run Time version. They are:

- SQLCRT16.C (source file)

- SQLCRT16.MAK (make file)

- SQLCRT16.DEF (def file)

Be sure to use the settings mentioned in *Building a 16-bit DLL* on page *8-18*.

External function example

Following is an example of the SAL external function *SalDateConstruct* invoked within stored procedure *dateconstruct*. The example also includes the CREATE EXTERNAL FUNCTION declaration for *SalDateConstruct*, and the SQL commands for creating a synonym for the function name, and privileges for executing the function. Read the end of this section for step by step explanations of this example.

```
-- create SAL external function definitions.

① create external function "SalDateConstruct"
   parameters (INT, INT, INT, INT, INT, INT)
   returns (DATETIME)
   library cdlli10.dll
   callstyle STDCALL
   execute in same thread;

② create public synonym "SalDateConstruct" for external
   function "SalDateConstruct";
```

```
③ grant execute on external function "SalDateConstruct" to
   public;

   store dateconstruct
④ procedure dateconstruct
   parameters
       number : nYear
       number : nMonth
       number : nDay
       number : nHour
       number : nMinute
       number : nSecond

       receive date/time : dtDate
   actions
⑤   set dtDate = SalDateConstruct
   (nYear,nMonth,nDay,nHour,nMinute,nSecond)
   ;

   execute p_extsal8
   \
   1994,12,26,9,15,0,,
   /
```

1. Creates the external function definition. This defines the parameters, return type, library, etc.

2. Creates a public synonym for *SalDateConstruct* so that all users may refer to the function as *SalDateConstruct*.

3. Grants all users execute privileges on *SalDateConstruct*.

4. Creates the procedure that invokes the external function *SalDateConstruct*. The procedure, *dateconstruct*, takes the individual components of a date and sends back a complete date.

5. Call to the external function *SalDateConstruct*.

Appendix A

SAL Functions

This appendix describes the SAL functions that you can invoke within SQLBase procedures.

SqlClearImmediate

Syntax

bOk = SqlClearImmediate ( )

Description

Disconnects the internal Sql Handle from a database.

You connect the internal handle to a database by calling SqlImmediate and it remains connected until the application terminates or you explicitly disconnect it with SqlClearImmediate.

SqlClearImmediate causes an implicit COMMIT if it is the last cursor you disconnect from the database.

Parameters

None.

Return value

bOk is TRUE if the function succeeds and FALSE if it fails.

SqlClose

Syntax

bOk = SqlClose (*hSql*)

Description

Invalidates a SQL command and/or frees the cursor name associated with the specified cursor, making the cursor name available for reuse.

If you create a named cursor by calling SqlOpen and then instead of closing it, call SqlOpen or SqlExecute again, you get an error that the name has already been used.

Parameters

hSql   Sql Handle. A handle that identifies a database connection.

Return value

bOk is TRUE if the function succeeds and FALSE if it fails.

SqlCommit

Syntax

bOk = SqlCommit (*hSql*)

Description

Commits *all* of the SQL transaction's cursors that are connected to the same database, including those outside the procedure.

Note: In stored procedures, if you have a SqlPrepare function called in an On Procedure Startup section and a SQLCommit function called in a subsequent On Procedure Execute section, the COMMIT will destroy the cursor of the SQLPrepare function. Subsequent executions will fail because the cursor's "preparation" is lost.

To prevent destroying a cursor's result set when a COMMIT is performed, turn on cursor context preservation by calling SqlSetParameter and setting the DBP_PRESERVE parameter to TRUE.

Parameters

hSql   Sql Handle. A handle that identifies a database connection.

Return value

bOk is TRUE if the function succeeds and FALSE if it fails.

Example

```
    ...
    Call SqlConnect ( hSql )
    Call SqlPrepare ( hSql, 'INSERT INTO TEST VALUES ( 1 )' )
    Call SqlExecute ( hSql )
    Call SqlCommit ( hSql )
...
```

SqlConnect

Syntax

bOk = SqlConnect (*hSql*)

Description

Connects to the currently active database. This means that SQLBase establishes a new connection to the same database that you were connected to when you executed the procedure.

For example, assume your SQLTalk session has two cursors outside the procedure, 1 and 2. These cursors are attached to databases DEMO1 and DEMO2, respectively. If you execute a procedure on cursor 1, you connect DEMO1; if you execute the procedure on cursor 2, you connect to DEMO2.

You cannot connect to multiple databases with SqlConnect.

Parameters

hSql   Receive Sql Handle. A handle that identifies a database
                         connection.

Return value

bOk is TRUE if the function succeeds and FALSE if it fails.

Example

Assume you are connected to the TEST database. When the procedure begins, it connects the *hSqlPrimary* Sql Handle to the TEST database. When the procedure ends, it disconnects the *hSqlPrimary* Sql Handle from the TEST database.

```
Actions
  On Procedure Startup
    Call SqlConnect ( hSqlPrimary )
...
  On Procedure Close
    Call SqlDisconnect ( hSqlPrimary )
...
```

SqlDisconnect

Syntax

bOk = SqlDisconnect (*hSql*)

Description

Disconnects from a database.

Parameters

hSql   Sql Handle. The handle that identifies the database
                          connection to disconnect.

Return value

bOk is TRUE if the function succeeds and FALSE if it fails.

Example

When the procedure begins, it connects the *hSqlPrimary* Sql Handle to the database
which is currently being accessed. When the procedure ends, it disconnects the
hSqlPrimary Sql Handle from the database.

```
    Actions
      On Procedure Startup
        Call SqlConnect ( hSqlPrimary )
  ...
      On Procedure Close
        Call SqlDisconnect ( hSqlPrimary )
...
```

SqlDropStoredCmd

Syntax

bOk = SqlDropStoredCmd (*hSql, strName*)

Description

Deletes a stored command/stored procedure from a database.

Parameters

hSql   Sql Handle. A handle that identifies a database connection.

strName   String. The name of the stored command/procedure to
                                   delete.

Return value

bOk is TRUE if the function succeeds and FALSE if it fails.

SqlError

Syntax

nError = SqlError (*hSql*)

Description

Returns the most recent error code for the specified Sql Handle.

SqlError is not useful after a call to SqlImmediate because SqlImmediate does not
return a handle that you can use as the parameter for SqlError.

Parameters

hSql   Sql Handle. A handle on which an error occurred.

Return value

nError is the error code returned. It is equal to zero (0) if no error occurred.

SqlExecute

Syntax

bOk = SqlExecute (*hSql*)

Description

Executes a SQL statement, procedure, or command that was prepared with
SqlPrepare, or a SQL statement, stored command, or stored procedure that was
retrieved with SqlRetrieve.

SqlExecute does not fetch data. To fetch data, call one of the fetch functions:
SqlFetchNext, SqlFetchPrevious, or SqlFetchRow.

Bind variables values are sent to the database when you call SqlExecute.

You can use SqlExecute just like SqlOpen, but you can never address rows in the
result set by a cursor name. That is, you cannot use the "CURRENT OF

<cursor_name>" and "ADJUSTING <cursor_name>" clauses to INSERT, UPDATE, or DELETE result set rows.

Parameters

hSql   Sql Handle. The handle associated with a SQL statement.

Return value

bOk is TRUE if the function succeeds and FALSE if it fails.

SqlExists

Syntax

bOk = SqlExists (*strSelect, bExists*)

Description

Determines whether a row or rows exist.

SqlExists connects to the currently active database and uses the internal Sql Handle to execute the specified query.

Parameters

strSelect   String. The SELECT statement that establishes the existence of a row.

bExists   Receive Boolean. TRUE if the row exists and FALSE if it does not.

Return value

bOk is TRUE if *strSelect* is correct and executable and FALSE otherwise.

SqlFetchNext

Syntax

bOk = SqlFetchNext (*hSql, nInd*)

Description

Fetches the next row in a result set. You must have first 1) prepared or retrieved the SELECT statement with SqlPrepare or SqlRetrieve, respectively, and then 2) either executed it with SqlExecute, or opened it with SqlOpen.

If you call the this function within the On Procedure Fetch section, it is recommended that you specify a Return statement. For example:

```
If NOT SqlFetchNext(hSqlCur1, nInd)
  Return 1
Else
  Return 0
```

Parameters

hSql   Sql Handle. The handle of a SELECT statement.

nInd   Receive Number. The fetch return code is one of the following fetch values:

Constant	Description
Fetch_Delete	Indicates failure. The row has been deleted since it was last fetched.
Fetch_EOF	Indicates failure. There are no more rows to fetch (end of fetch).
Fetch_Ok	Indicates success. The row was fetched.
Fetch_Update	Indicates failure. The row has been updated since it was last fetched.

Return value

bOk is TRUE if there is another row to fetch and FALSE otherwise.

SqlFetchPrevious

Syntax

bOk = SqlFetchPrevious (*hSql, nInd*)

Description

Fetches the previous row in a scrollable result set. You must have first 1) prepared or retrieved the SELECT statement with SqlPrepare or SqlRetrieve, respectively, and then 2) either executed it with SqlExecute, or opened it with SqlOpen.

If you call the this function within the On Procedure Fetch section, it is recommended that you specify a Return statement. For example:

```
If NOT SqlFetchPrevious(hSqlCur1, nInd)
  Return 1
```

```
Else
  Return 0
```

Note: To use this function, first ensure that result set mode is set to on. To turn it on, use SqlSetResultSet.

Parameters

hSql   Sql Handle. The handle of a SELECT statement.

nInd   Receive Number. The fetch return code is one of the following fetch values:

Constant	Description
Fetch_Delete	Indicates failure. The row has been deleted since it was last fetched.
Fetch_EOF	Indicates failure. There are no more rows to fetch (end of fetch).
Fetch_Ok	Indicates success. The row was fetched.
Fetch_Update	Indicates failure. The row has been updated since it was last fetched.

Return value

bOk is TRUE if there is another row to fetch and FALSE otherwise.

SqlFetchRow

Syntax

bOk = SqlFetchRow (*hSql, nRow, nInd*)

Description

Fetches a row according to an absolute row position. You must have first 1) prepared or retrieved the SELECT statement with SqlPrepare or SqlRetrieve, respectively, and then 2) either executed it with SqlExecute, or opened it with SqlOpen.

Parameters

hSql   Sql Handle. The handle of a SELECT statement.

nRow   Number. The row number of the row to fetch.

| *nInd* | Receive Number. The fetch return code is one of the following fetch values: |

Constant	Description
Fetch_Delete	Indicates failure. The row has been deleted since it was last fetched.
Fetch_EOF	Indicates failure. There are no more rows to fetch (end of fetch).
Fetch_Ok	Indicates success. The row was fetched.
Fetch_Update	Indicates failure. The row has been updated since it was last fetched.

Return value

bOk is TRUE if *nRow* could be fetched and FALSE otherwise.

SqlGetErrorPosition

Syntax

bOk = SqlGetErrorPosition (*hSql, nPos*)

Description

Returns the offset of the error position within a SQL statement. After a SqlPrepare, the error position points to the place in the SQL statement where a syntax error was detected. The first character position in the SQL statement is zero (0).

Parameters

| *hSql* | Sql Handle. The handle of a SELECT statement. |
| *nPos* | Receive Number. The position in the SQL statement where a syntax error occurred. |

Return value

bOk is TRUE if the function succeeds and FALSE if it fails.

SqlGetErrorText

Syntax

bOk = SqlGetErrorText (*nError, strText*)

Description

Gets the message text for a SQL error number from *error.sql*.

Parameters

nError	Number. The error number.
strText	Receive String. The error text.

Return value

bOk is TRUE if the function succeeds and FALSE if it fails.

SqlGetModifiedRows

Syntax

bOk = SqlGetModifiedRows (*hSql, nCount*)

Description

Returns the number of rows affected by the most recent INSERT, UPDATE, or DELETE statement.

Parameters

hSql	Sql Handle. The handle of a SQL statement.
nCount	Receive Number. The number of rows affected.

Return value

bOk is TRUE if the function succeeds and FALSE if it fails.

SqlGetParameter

Syntax

bOk = SqlGetParameter (*hSql, nParameter, nNumber, strString*)

Description

Gets the value of a database parameter. This function returns the parameter value in *nNumber* or *strString* as appropriate for the data type of the parameter.

Parameters

hSql	Sql Handle. A handle that identifies a database connection.
nParameter	Number. The database parameter. You can specify either one of the literal strings from the following table, or the number associated with the desired database parameter (for example, the When to return Describe information (SQLPDIS) parameter number is 3018). To find the number of the associated parameter, see the *sql.h* header file:

Constant	Description
DBP_AUTOCOMMIT	**Autocommit.** If autocommit is on (TRUE), the database commits changes automatically after each SQL command. If autocommit is off (FALSE), the database commits changes only when you issue a COMMIT command.
DBP_BRAND	**Database server brand.** Currently, only the SQLBase brand (DBV_BRAND_SQL) is supported.

Constant	Description
DBP_FetchTHROUGH	**Fetchthrough.** The fetchthrough feature enables you to retrieve rows directly from the database server instead of from the client's input message buffer, thereby ensuring that the user sees the most up-to-date data. If fetchthrough is on (TRUE), the application fetches data one row at a time from the backend. Using this feature increases response time because of the network traffic incurred, so you should only use it when the user needs the most current information. If fetchthrough is off (FALSE), the application fetches data from the client's input message buffer whenever possible. This is the default. Note that in a procedure, performance is enhanced. Each client side fetch request (by default) generates a buffer full of row(s), rather than one row for each fetch. If you want the On Procedure Fetch section to execute exactly once for every fetch call from the client (returning one row at a time), set fetchthrough mode on (TRUE) at the client.
DBP_LOCKWAITTIMEOUT	**Lock wait timeout.** This is the number of seconds an application should wait for the database server to acquire a lock before timing out. After the specified time has elapsed, SQLBase rolls back the transaction. The default lock timeout value is 300 seconds. Valid timeout values are 1 to 1800 (30 minutes), -1 (wait forever), and 0 (never wait).
DBP_NOPREBUILD	**Don't Prebuild.** SQLbase does not prebuild result sets when the application is in result set mode and is using the Release Locks isolation level. Pre-building a result set provides the advantage of being able to release shared locks and return control to the client. The disadvantage of pre-building a result set is that the application must wait while the result set is being built. If noprebuild is on (TRUE), result sets are not pre-built. A shared lock remains on the current page. This is the default. If noprebuild is off (FALSE), result sets are pre-built.

Constant	Description
DBP_PRESERVE	**Cursor context preservation.** If cursor context preservation is on (TRUE), a COMMIT does not destroy an active result set. This enables an application to maintain its position after a COMMIT, ROLLBACK, INSERT, or UPDATE. A user-initiated ROLLBACK preserves cursor context if both of the following are true: • The application is in Release Locks (RL) isolation level • A data definition language (DDL) operation was not performed Note that a system-initiated ROLLBACK such as a deadlock, timeout, etc., does not preserve cursor context even when cursor context preservation is on. If cursor context preservation is off (FALSE), a COMMIT does destroy an active result set. Cursor context preservation is lost.
DBP_ROLLBACKONTIMEOUT	**Roll back a transaction when a lock timeout occurs.** If TRUE, the entire transaction rolls back when a lock timeout occurs. If FALSE, only the current command rolls back on a lock timeout. The default is TRUE.
DBP_VERSION	Database server version.

nNumber	Receive number. The value (TRUE or FALSE) of the parameter.
	If *nParameter* is DBP_BRAND, *nNumber* is one of the DBV_BRAND_* values.
strString	Receive string. If you specify DBP_VERSION in *nParameter*, this is the version number.

Return value

bOk is TRUE if the function succeeds and FALSE if it fails.

Example

```
Actions
  On Procedure Startup
    Call SqlGetParameter ( hSql, DBP_LOCKWAITTIMEOUT,\
    nTimeout, strNull )
```

SqlGetParameterAll

Syntax

bOk = SqlGetParameterAll (*hSql, nParameter, nNumber, strString*,
   bNumber)

Description

Gets the value of a database parameter identified by a SQLP* constant value defined
in *sql.h*. This function returns the parameter value in *nNumber* or *strString* as
appropriate for the data type of the parameter.

Note: A set of the SQLP* constants in *sql.h* have the same values as the DBP_* constants, but
the values identify different parameters. Be sure to specify the correct number.

Parameters

hSql	Sql Handle. A handle that identifies a database connection.
nParameter	Number. The database parameter. You can specify either one of the literal strings from the table in the previous SqlGetParameter section, or the number associated with the desired database parameter (for example, the When to return Describe information (SQLPDIS) parameter number is 3018). To find the number of the associated parameter, see the *sql.h* header file.
nNumber	Receive number. The value (TRUE or FALSE) of the parameter. If *nParameter* is DBP_BRAND, *nNumber* is one of the DBV_BRAND_* values.
strString	Receive string. If you specify DBP_VERSION in *nParameter*, this is the version number.
bNumber	Boolean. If TRUE, the parameter value is returned in *nNumber*. If FALSE, the parameter value is returned in *strString*.

Return value

bOk is TRUE if the function succeeds and FALSE if it fails.

SqlGetResultSetCount

Syntax

bOk = SqlGetResultSetCount (*hSql, nCount*)

Description

Counts the rows in a scrollable result set by building the result set.

SQLBase fetches each row that has not already been fetched, returns a count of the rows, and positions the cursor back to its original position.

Warning: This can be time-consuming if the result set is large.

INSERTs into the result set increase the result set row count, but DELETEs do not decrease the row count. However, the deleted rows disappear on the next SELECT.

You must be in result set mode and you must call SqlExecute before SqlGetResultSetCount.

Parameters

hSql   Sql Handle. A handle associated with a result set.

nCount   Receive Number. The number of rows in the result set.

Return value

bOk is TRUE if the function succeeds and FALSE if it fails.

Example

```
      ...
      Call SqlPrepare ( hSql, strSqlStatement )
      Call SqlExecute ( hSql )
      Call SqlGetResultSetCount ( hSql, nRowCount )
   ...
```

SqlGetRollbackFlag

Syntax

bOk = SqlGetRollbackFlag (*hSql, bRollbackFlag*)

Description

Returns the database rollback flag. Use this function after an error to find out if a transaction rolled back.

SQLBase sets the rollback flag when a system-initiated rollback occurs as the result of a deadlock or system failure. It does not set the rollback flag on a user-initiated rollback.

Parameters

hSql	Sql Handle. The handle associated with the function call which got an error.
bRollbackFlag	Receive Boolean. TRUE if a rollback occurred and FALSE otherwise.

Return value

bOk is TRUE if the function succeeds and FALSE if it fails.

Example

```
...
Call SqlGetRollbackFlag ( hSqlError, bRollbackFlag )
If bRollbackFlag
! Execute code to handle rolled back transaction...
```

SqlImmediate

Syntax

bOk = SqlImmediate (*strSqlCommand*)

Description

Prepares and executes a SQL statement. SqlImmediate actually performs a SqlConnect, a SqlPrepare, a SqlExecute, and for SELECT statements, a SqlFetchNext. The first time you call SqlImmediate, the system performs all of these functions. On later calls, only those functions that need to be performed are performed. For example, if the handle is still connected to a database, the system does

not perform a SqlConnect. If the SQL statement to compile is the same statement as that used by the last SqlImmediate call, the system does not perform a SqlPrepare.

Use SqlImmediate with INSERT, UPDATE, DELETE, and other non-query SQL commands. You can use SqlImmediate with a SELECT statement if you expect that the statement only returns one row. SqlImmediate manages the internal handle.

Any command that you execute with SqlImmediate can also be executed with explicit calls to SqlConnect, SqlPrepare, SqlExecute or SqlOpen, and SqlFetchNext, for SELECTs.

When static procedures are executed, the compile phase of SqlImmediate is not reprocessed since all SQL statements within a static procedure are precompiled.

Note: Do not use SqlImmediate if you are implementing error handling with SqlError(), since SqlImmediate does not retain a database handle.

Parameters

strSqlCommand	String. The SQL statement to prepare and execute. This statement cannot have more than 255 bind variables and 255 INTO variables.

Return value

bOk is TRUE if the function succeeds and FALSE if it fails.

SqlOpen

Syntax

bOk = SqlOpen (*hSql, strCursorName*)

Description

Names a cursor and executes a SQL statement. Use this function to perform INSERTs, UPDATEs, and DELETEs on the current row.

Call SqlOpen after SqlPrepare and before any of the SqlFetch* commands.

Parameters

hSql	Sql Handle. The handle associated with the SqlPrepare.
strCursorName	String. A string containing the cursor name.

Specify this name in the 'CURRENT OF <cursor_name>' or 'ADJUSTING <cursor_name>' clause of an INSERT, UPDATE, or DELETE statement.

The value of this parameter is case insensitive. You can set it to null using the empty string (").

Return value

bOk is TRUE if the function succeeds and FALSE if it fails.

SqlPrepare

Syntax

bOk = SqlPrepare (*hSql, strSqlStatement*)

Description

Compiles a SQL statement (including non-stored procedures) for execution. Compiling includes:

- Checking the syntax of the SQL statement or procedure.

- Checking the system catalog.

- Processing a SELECT statement's INTO clause.

  An INTO clause names where data is placed when it is fetched. These variables are sometimes called INTO variables. You can specify up to 255 INTO variables per SQL statement.

- Identifying bind variables in the SQL statement. Bind variables contain input data for the statement. You can specify up to 255 bind variables per SQL statement.

Follow this function with a SqlOpen, SqlExecute, or fetches.

When static procedures are executed, the compile phase of SqlPrepare is not reprocessed since all SQL statements within a static procedure are precompiled.

Parameters

hSql	Sql Handle. A handle that identifies a database connection.
strSqlStatement	String. The SQL statement to compile.

Return value

bOk is TRUE if the function succeeds and FALSE if it fails.

SqlPrepareAndExecute

Syntax

bOk = SqlPrepareAndExecute (*hSql, strSqlStatement*)

Description

Compiles and executes a SQL statement (including non-stored procedures). Compiling includes:

- Checking the syntax of the SQL statement.

- Checking the system catalog.

- Processing a SELECT statement's INTO clause.

    An INTO clause names where data is placed when it is fetched. These variables are sometimes called INTO variables. You can specify up to 255 INTO variables per SQL statement.

- Identifying bind variables in the SQL statement. Bind variables contain input data for the statement. You can specify up to 255 bind variables per SQL statement.

SqlPrepareAndExecute does not fetch data. To fetch data, call one of the following fetch functions: SqlFetchNext, SqlFetchPrevious, or SqlFetchRow.

When static procedures are executed, the compile phase of SqlPrepareAndExecute is not reprocessed since all SQL statements within a static procedure are precompiled.

For dynamic procedures, it is recommended that you do not call SqlPrepareAndExecute if your procedure needs to be executed repeatedly, and only needs to be compiled once. An example is binding different variables at execution time. Calling SqlPrepareAndExecute in this situation would compile the SQL statement each time it is executed, resulting in unnecessary overhead.

Instead, prepare the SQL statement with SqlPrepare in the On Procedure Startup section, and then execute it in the On Procedure Execute section with SqlExecute.

Parameters

hSql   Sql Handle. A handle that identifies a database connection.

strSqlStatement   String. The SQL statement to compile and execute.

Return value

bOk is TRUE if the function succeeds and FALSE if it fails.

SqlRetrieve

Syntax

bOk = SqlRetrieve (*hSql, strName, strBindList, strIntoList*)

Description

Retrieves a SQLBase stored command or stored procedure.

To execute the command, you need only call SqlExecute. You do not need to compile the command with SqlPrepare because the command is compiled when it is stored with SqlStore.

Parameters

hSql	Sql Handle. A handle that identifies a database connection.
strName	String. The name of the compiled command.
strBindList	String. A comma-separated list of up to 255 bind variables. Each string must be preceded by a colon (**:**).This list has the same number of variables as the compiled command. This string can be null.
strIntoList	String. A comma-separated list of up to 255 INTO variables. Each string must be preceded by a colon (**:**).This list has the same (or less) number of INTO variables as named in the SELECT list of the compiled command.
	This string *can* be null ('' or strNULL).

Return value

bOk is TRUE if the function succeeds and FALSE if it fails.

SqlSetIsolationLevel

Syntax

bOk = SqlSetIsolationLevel (*hSql, strIsolation*)

Description

Sets the isolation level for all the application's cursors connected to the database.

The default isolation level for a procedure is Read Repeatability (RR). However, if the calling program is set at a different isolation level, the procedure isolation level automatically changes to that of the calling program.

Also, if the procedure makes a change to the isolation level using SqlSetIsolationLevel, the calling program inherits this new isolation level by default. As a result, all cursors connected to the same database both within and outside the procedure are committed.

Parameters

hSql   Sql Handle. A handle that identifies a database connection.

strIsolation   String. The isolation level to set. Specify one of these values:

CS	Cursor Stability
RL	Release Locks
RO	Read Only
RR	Read Repeatability (default)

Return value

bOk is TRUE if the function succeeds and FALSE if it fails.

SqlSetLockTimeout

Syntax

bOk = SqlSetLockTimeout (*hSql, nTimeout*)

Description

Specifies the maximum time to wait to acquire a lock. After the specified time elapses, a timeout occurs and the transaction rolls back.

Parameters

hSql   Sql Handle. A handle that identifies a database connection; the cursor on which you want to set a lock timeout value.

nTimeout   Number. The timeout period in seconds. Valid value include -1 (wait forever), 0 (never wait), and values up to and including 1800 (30 minutes). The default is 300.

Return value

bOk is TRUE if the function succeeds and FALSE if it fails.

SqlSetParameter

Syntax

bOk = SqlSetParameter (*hSql, nParameter, nNumber, strString*)

Description

Sets the value of a database parameter. Use the number (*nNumber*) and string (*strString*) arguments as appropriate for the data type of the parameter.

Parameters

hSql	Sql Handle. A handle that identifies a database connection.
nParameter	Number. The database parameter to set. Specify one of the DBP_* constants or the desired database parameter number (found in *sql.h*) listed for SqlGetParameter.
nNumber	Number. The value of *nParameter*. Specify TRUE or FALSE for all but DBP_LOCKWAITTIMEOUT, for which you must specify a value in seconds.
strString	String. The value of *nParameter*.

Return value

bOk is TRUE if the function succeeds and FALSE if it fails.

SqlSetParameterAll

Syntax

bOk = SqlSetParameterAll (*hSql, nParameter, nNumber, strString, bNumber*)

Description

Sets the value of a database parameter. Use the number (*nNumber*) and string (*strString*) arguments as appropriate for the data type of the parameter.

Parameters

hSql	Sql Handle. A handle that identifies a database connection.

nParameter	Number. The database parameter to set. Specify one of the DBP_* constants or the desired database parameter number (found in *sql.h*) listed for SqlGetParameter.
nNumber	Number. The value of *nParameter*. Specify TRUE or FALSE for all but DBP_LOCKWAITTIMEOUT, for which you must specify a value in seconds.
strString	String. The value of *nParameter*.
bNumber	Boolean. If TRUE, the parameter's value is in *nNumber*. If FALSE, the parameter's value is in *strString*.

Return value

bOk is TRUE if the function succeeds and FALSE if it fails.

SqlSetResultSet

Syntax

bOk = SqlSetResultSet (*hSql, bSet*)

Description

Turns result set mode on or off.

By default, result set mode is on.

Parameters

hSql	Sql Handle. A handle that identifies a database connection.
bSet	Boolean. Turns result set mode on (TRUE) or off (FALSE).

Return value

bOk is TRUE if the function succeeds and FALSE if it fails.

SqlStore

Syntax

bOk = SqlStore (*hSql, strName, strSqlCommand*)

Description

Stores and names a SQLBase compiled SQL statement (including procedures).

You do not need to call SqlPrepare before calling SqlStore. SqlStore compiles the SQL statement.

You can specify up to 255 bind variables. Use numeric bind variables in the SQL statement, not variable names. For example: "SELECT * FROM PRESIDENT WHERE LASTNAME = :1 AND AGE = :2;".

When you retrieve the stored command with SqlRetrieve, you specify the bind variable names in the INTO clause.

Parameters

hSql	Sql Handle. A handle that identifies a database connection.
strName	String. The name of the stored command.
strSqlCommand	String. The SQL statement to compile and store.

Return value

bOk is TRUE if the function succeeds and FALSE if it fails.

Glossary

access path—The path used to get the data specified in a SQL command. An access path can involve an index or a sequential search (table scan), or a combination of the two. Alternate paths are judged based on the efficiency of locating the data.

aggregate function—A SQL operation that produces a summary value from a set of values.

alias—An alternative name used to identify a database object.

API (application programming interface)—A set of functions that a program uses to access a database.

application—A program written by or for a user that applies to the user's work. A program or set of programs that perform a task. For example, a payroll system.

argument—A value entered in a command that defines the data to operate on or that controls execution. Also called parameter or operand.

arithmetic expression—An expression that contains operations and arguments that can be reduced to a single numeric value.

arithmetic operator—A symbol used to represent an arithmetic operation, such as the plus sign (+) or the minus sign (-).

attribute—A characteristic or property. For example, the data type or length of a row. Sometimes, attribute is used as a synonym for column or field.

audit file—A log file that records output from an audit operation.

audit message—A message string that you can include in an audit file

audit operation—A SQLBase operation that logs database activities and performance, writing output to an audit file. For example, you can monitor who logs on to a database and what tables they access, or record command execution time.

authorization—The right granted to a user to access a database.

authorization ID—A unique name that identifies a user. Associated to each authorization id is a password. Abbreviated auth id. Also called username.

back-end—See database server.

backup—To copy information onto a diskette, fixed disk, or tape for record keeping or recovery purposes.

base table—The permanent table on which a view is based. A base table is created with the CREATE TABLE command and does not depend on any other table. A base table has its description and its data physically stored in the database. Also called underlying table.

bindery—A NetWare 3.x database that contains information about network resources such as a SQLBase database server.

bind variable—A variable used to associate data to a SQL command. Bind variables can be used in the VALUES clause of an INSERT command, in a WHERE clause, or in the SET clause of an UPDATE command. Bind variables are the mechanism to transmit data between an application work area and SQLBase. Also called into variable or substitution variable.

browse—A mode where a user queries some of a database without necessarily making additions or changes. In a browsing application, a user needs to examine data before deciding what to do with it. A browsing application allows the user to scroll forward and backward through data.

buffer—A memory area used to hold data during input/output operations.

C/API—A language interface that lets a programmer develop a database application in the C programming language. The C/API has functions that a programmer calls to access a database using SQL commands.

cache—A temporary storage area in computer memory for database pages being accessed and changed by database users. A cache is used because it is faster to read and write to computer memory than to a disk file.

Cartesian product—In a join, all the possible combinations of the rows from each of the tables. The number of rows in the Cartesian product is equal to the number of rows in the first table times the number of rows in the second table, and so on. A Cartesian product is the first step in joining tables. Once the Cartesian product has been formed, the rows that do not satisfy the join conditions are eliminated.

cascade—A delete rule which specifies that changing a value in the parent table automatically affects any related rows in the dependent table.

case sensitive—A condition in which names must be entered in a specific lower-case, upper-case, or mixed-case format to be valid.

cast—The conversion between different data types that represent the same data.

CHAR—A column data type that stores character strings with a user-specified length. SQLBase stores CHAR columns as variable-length strings. Also called VARCHAR.

character—A letter, digit, or special character (such as a punctuation mark) that is used to represent data.

character string—A sequence of characters treated as a unit.

checkpoint—A point at which database changes older than the last checkpoint are flushed to disk. Checkpoints are needed to ensure crash recovery.

clause—A distinct part of a SQL command, such as the WHERE clause; usually followed by an argument.

client—A computer that accesses shared resources on other computers running as servers on the network. Also called front-end or requester.

column—A data value that describes one characteristic of an entity. The smallest unit of data that can be referred to in a row. A column contains one unit of data in a row of a table. A column has a name and a data type. Sometimes called field or attribute.

command—A user request to perform a task or operation. In SQLTalk, each command starts with a name, and has clauses and arguments that tailor the action that is performed. A command can include limits or specific terms for its execution, such as a query for names and addresses in a single zip code. Sometimes called statement.

commit—A process that causes data changed by an application to become part of the physical database. Locks are freed after a commit (except when cursor-context preservation is on). Before changes are stored, both the old and new data exist so that changes can be stored or the data can be restored to its prior state.

commit server—A database server participating in a distributed transaction, that has commit service enabled. It logs information about the distributed transaction and assists in recover after a network failure.

composite primary key—A primary key made up of more than one column in a table.

concatenated key—An index that is created on more than one column of a table. Can be used to guarantee that those columns are unique for every row in the table and to speed access to rows via those columns.

concatenation—Combining two or more character strings into a single string.

concurrency—The shared use of a database by multiple users or application programs at the same time. Multiple users can execute database transactions simultaneously without interfering with each other. The database software ensures that all users see correct data and that all changes are made in the proper order.

configure—To define the features and settings for a database server or its client applications.

connect—To provide a valid authorization-id and password to log on to a database.

connection handle—Used to create multiple, independent connections. An application must request a connection handle before it opens a cursor. Each connection handle represents a single transaction and can have multiple cursors. An application may request multiple connection handles if it is involved in a sequence of transactions.

consistency—A state that guarantees that all data encountered by a transaction does not change for the duration of a command. Consistency ensures that uncommitted updates are not seen by other users.

constant—Specifies an unchanging value. Also called literal.

control file—An ASCII file containing information to manage segmented load/unload files.

cooperative processing—Processing that is distributed between a client and a server in a such a way that each computer works on the parts of the application that it is best at handling.

coordinator—The application that initiates a distributed transaction.

correlated subquery—A subquery that is executed once for each row selected by the outer query. A subquery cannot be evaluated independently because it depends on the outer query for its results. Also called a repeating query. Also see subquery and outer query.

correlation name—A temporary name assigned to a table in an UPDATE, DELETE, or SELECT command. The correlation name and column name are combined to refer to a column from a specific table later in the same command. A correlation name is used when a reference to a column name could be ambiguous. Also called range variable.

crash recovery—The procedures that SQLBase uses automatically to bring a database to a consistent state after a failure.

CRC (Cyclic Redundancy Check)—A technique that makes unauthorized changes to a database page detectable. SQLBase appends an extra bit sequence to every database page called a Frame Check Sequence (FCS) that holds redundant information about the page. When SQLBase reads a database page, it checks the FCS to detect unauthorized changes.

current row—The latest row of the active result set which has been fetched by a cursor. Each subsequent fetch retrieves the next row of the active result set.

cursor—The term cursor refers to one of the following definitions:

- The position of a row within a result table. A cursor is used to retrieve rows from the result table. A named cursor can be used in the CURRENT OF clause or the ADJUSTING clause to make updates or deletions.

- A work space in memory that is used for gaining access to the database and processing a SQL command. This work space contains the return code, number of rows, error position, number of select list items, number of bind variables, rollback flag, and the command type of the current command.

- When the cursor belongs to an explicit connection handle that is created using the SQL/API function call *sqlcch* or the SQLTalk BEGIN CONNECTION command, it identifies a task or activity within a transaction. The task or activity can be compiled/executed independently within a single connection thread.

  Cursors can be associated with specific connection handles, allowing multiple transactions to the same database within a single application. When this is implemented, only one user is allowed per transaction.

- When a cursor belongs to an implicit connection handle created using the SQL/API function call *sqlcnc* or *sqlcnr*, or the SQLTalk CONNECT command, the cursor applies to an application in which you are connecting the cursor to a specific database that belongs to a single transaction.

cursor-context preservation—A feature of SQLBase where result sets are maintained after a COMMIT. A COMMIT does not destroy an active result set (cursor context). This enables an application to maintain its position after a COMMIT, INSERT, or UPDATE. For fetch operations, locks are kept on pages required to maintain the fetch position.

cursor handle—Identifies a task or activity within a transaction. When a connection handle is included in a function call to open a new cursor, the function call returns a cursor handle. The cursor handle can be used in subsequent SQL/API calls to identify the connection thread. A cursor handle is always part of a specific transaction and cannot be used in multiple transactions. However, a cursor handle can be associated with a specific connection handle. The ability to have multiple transactions to the same database within a single application is possible by associating cursor handles with connection handles.

Cursor Stability (CS)—The isolation level where a page acquires a shared lock on it only while it is being read (while the cursor is on it). A shared lock is dropped as the cursor leaves the page, but an exclusive lock (the type of lock used for an update) is retained until the transaction completes. This isolation level provides higher concurrency than Read Repeatability, but consistency is lower.

data dictionary—See system catalog.

data type—Any of the standard forms of data that SQLBase can store and manipulate. An attribute that specifies the representation for a column in a table. Examples of data types in SQLBase are CHAR (or VARCHAR), LONG VARCHAR (or

LONG), NUMBER, DECIMAL (or DEC), INTEGER (or INT), SMALLINT, DOUBLE PRECISION, FLOAT, REAL, DATETIME (or TIMESTAMP), DATE, TIME.

database—A collection of interrelated or independent pieces of information stored together without unnecessary redundancy. A database can be accessed and operated upon by client applications such as SQLTalk.

database administrator (DBA)—A person responsible for the design, planning, installation, configuration, control, management, maintenance, and operation of a DBMS and its supporting network. A DBA ensures successful use of the DBMS by users.

A DBA is authorized to grant and revoke other users' access to a database, modify database options that affect all users, and perform other administrative functions.

database area—A database area corresponds to a file. These areas can be spread across multiple disk volumes to take advantage of parallel disk input/output operations.

database management system (DBMS)—A software system that manages the creation, organization, and modification of a database and access to data stored within it. A DBMS provides centralized control, data independence, and complex physical structures for efficient access, integrity, recovery, concurrency, and security.

database object—A table, view, index, synonym or other object created and manipulated through SQL.

database server—A DBMS that a user interacts with through a client application on the same or a different computer. Also called back-end or engine.

DATE—A column data type in SQL that represents a date value as a three-part value (day, month, and year).

date/time value—A value of the data type DATE, TIME, or TIMESTAMP.

DCL (Data Control Language)—SQL commands that assign database access privileges and security such as GRANT and REVOKE.

DDL (Data Definition Language)—SQL commands that create and define database objects such as CREATE TABLE, ALTER TABLE, and DROP TABLE.

deadlock—A situation when two transactions, each having a lock on a database page, attempt to acquire a lock on the other's database page. One type of deadlock is where each transaction holds a shared lock on a page and each wishes to acquire an exclusive lock. Also called deadly embrace.

DECIMAL—A column data type that contains numeric data with a decimal point. Also called DEC.

default—An attribute, value, or setting that is assumed when none is explicitly specified.

delimited identifier—An identifier enclosed between two double quote characters (") because it contains reserved words, spaces, or special characters.

delimiter—A character that groups or separates items in a command.

dependent object—An object whose existence depends on another object.

For example, if a stored procedure calls an external function, the stored procedure is the dependent object of the external function, since its existence depends on the external function.

dependent table—The table containing the foreign key.

determinant object—An object that determines the existence of another object.

For example, if a stored procedure calls an external function, the external function is the determinant object, since it determines the existence of the stored procedure.

digital signature—A unique binary number generated by an algorithm that identifies the content of a larger block of bytes.

dirty page—A database page in cache that has been changed but has not been written back to disk.

distributed database—A database whose objects reside on more than one system in a network of systems and whose objects can be accessed from any system in the network.

distributed transaction—Coordinates SQL statements among multiple databases that are connected by a network.

DLL (Dynamic Link Library)—A program library written in C or assembler that contains related modules of compiled code. The functions in a DLL are not read until run-time (dynamic linking).

DML (Data Manipulation Language)—SQL commands that change data such as INSERT, DELETE, UPDATE, COMMIT, and ROLLBACK.

DOUBLE PRECISION—A column data type that stores a floating point number.

DQL (Data Query Language)—The SQL SELECT command, which lets a user request information from a database.

duplicates—An option used when creating an index for a table that specifies whether duplicate values are allowed for a key.

embedded SQL—SQL commands that are embedded within a program, and are prepared during precompilation and compilation before the program is executed. After a SQL command is prepared, the command itself does not change (although values of host variables specified within the command can change). Also called static SQL.

encryption—The transformation of data into a form unreadable by anyone without a decryption key or password. Encryption ensures privacy by keeping information hidden from anyone for whom it is not intended, even those who can see the encrypted data. Unencrypted data is called plain text; encrypted data is called cipher text.

engine—See database server.

entity—A person, place, or thing represented by a table. In a table, each row represents an entity.

equijoin—A join where columns are compared on the basis of equality, and all the columns in the tables being joined are included in the results.

Ethernet—A LAN with a bus topology (a single cable not connected at the ends). When a computer wants to transmit, it first checks to see if another computer is transmitting. After a computer transmits, it can detect if a collision has happened. Ethernet is a broadcast network and all computers on the network hear all transmissions. A computer selects only those transmissions addressed to it.

exclusive lock (X-lock)—An exclusive lock allows only one user to have a lock on a page at a time. An exclusive lock prevents another user from acquiring a lock until the exclusive lock is released. Exclusive locks are placed when a page is to be modified (such as for an UPDATE, INSERT, or DELETE).

An exclusive lock differs from a shared lock because it does not permit another user to place any type of lock on the same data.

expression—An item or a combination of items and operators that yield a single value. Examples are column names which yield the value of the column in successive rows, arithmetic expressions built with operators such as + or - that yield the result of performing the operation, and functions which yield the value of the function for its argument.

extent page—A database page used when a row is INSERTed that is longer than a page or when a row is UPDATEd and there is not enough space in the original page to hold the data.

external function—A user-defined function that resides in an "external" DLL (Dynamic Link Library) invoked within a SQLBase stored procedure.

field—See column.

file server—A computer that allows network users to store and share information.

FLOAT—A column data type that stores floating point numbers.

floating point—A number represented as a number followed by an exponent designator (such as 1.234E2, -5.678E2, or 1.234E-2). Also called E-notation or scientific notation.

foreign key—Foreign keys logically connect different tables. A foreign key is a column or combination of columns in one table whose values match a primary key in another table. A foreign key can also be used to match a primary key within the same table.

front-end—See client.

function—A predefined operation that returns a single value per row in the output result table.

grant—That act of a system administrator to permit a user to make specified use of a database. A user may be granted access to an entire database or specific portions, and have unlimited or strictly-limited power to display, change, add, or delete data.

GUI (Graphical User Interface)—A graphics-based user interface with windows, icons, pull-down menus, a pointer, and a mouse. Microsoft Windows and Presentation Manager are examples of graphical user interfaces.

history file—Contains previous versions of changed database pages. Used when read-only (RO) isolation level is enabled.

host language—A program written in a language that contains SQL commands.

identifier—The name of a database object.

index—A data structure associated with a table used to locate a row without scanning an entire table. An index has an entry for each value found in a table's indexed column or columns, and pointers to rows having that value. An index is logically ordered by the values of a key. Indexes can also enforce uniqueness on the rows in a table.

INTEGER—A column data type that stores a number without a decimal point. Also call INT.

isolation level—The extent to which operations performed by one user can be affected by (are isolated from) operations performed by another user. The isolation levels are Read Repeatability (RR), Cursor Stability (CS), Release Locks (RL), and Read Only (RO).

join—A query that retrieves data from two or more tables. Rows are selected when columns from one table match columns from another table. See also Cartesian product, self-join, equijoin, natural join, theta join, and outer join.

key—A column or a set of columns in an index used to identify a row. A key value can be used to locate a row.

keyword—One of the predefined words in a command language.

local area network (LAN)—A collection of connected computers that share data and resources, and access other networks or remote hosts. Usually, a LAN is geographically confined and microcomputer-based.

lock—To temporarily restrict other usersÕ access to data to maintain consistency. Locking prevents data from being modified by more than one user at a time and prevents data from being read while being updated. A lock serializes access to data and prevents simultaneous updates that might result in inconsistent data. See shared lock (S-lock) and exclusive lock (X-lock).

logical operator—A symbol for a logical operation that connects expressions in a WHERE or HAVING clause. Examples are AND, OR, and NOT. An expression formed with logical operators evaluates to either TRUE or FALSE. Logical operators define or limit the information sought. Also called Boolean operator.

LONG VARCHAR—In SQL, a column data type where the value can be longer than 254 bytes. The user does not specify a length. SQLBase stores LONG VARCHAR columns as variable-length strings. Also called LONG.

mathematical function—An operation such as finding the average, minimum, or maximum value of a set of values.

media recovery—Restoring data from backup after events such as a disk head crash, operating system crash, or a user accidentally dropping a database object.

message buffer—The input message buffer is allocated on both the client computer and the database server. The database server builds an input message in this buffer on the database server and sends it across the network to a buffer on the client. It is called an input message buffer because it is input from the client's point of view.

The out put message buffer is allocated on both the client computer and on the database server. The client builds an output message in this buffer and sends it to a buffer on the database server. It is called an output message buffer because it is output from the client's point of view.

modulo—An arithmetic operator that returns an integer remainder after a division operation on two integers.

multi-user—The ability of a computer system to provide its services to more than one user at a time.

natural join—An equijoin where the value of the columns being joined are compared on the basis of equality. All the columns in the tables are included in the results but only one of each pair of joined columns is included.

NDS (NetWare Directory Services)—A network-wide directory included with NetWare 4.x, that provides global access to all network resources, regardless of their physical location. The directory is accessible from multiple points by network users, services and applications.

nested query—See subquery.

NetWare—The networking components sold by Novell. NetWare is a collection of data link drivers, a transport protocol stack, client computer software, and the NetWare server operating system. NetWare runs on Token Ring, Ethernet, and ARCNET.

NetWare 386—A server operating system from Novell for computers that controls system resources on a network.

NLM (NetWare Loadable Module)—An NLM is a NetWare program that you can load into or unload from server memory while the server is running. When loaded, an NLM is part of the NetWare operating system. When unloaded, an NLM releases the memory and resources that were allocated for it.

null—A value that indicates the absence of data. Null is not considered equivalent to zero or to blank. A value of null is not considered to be greater than, less than, or equivalent to any other value, including another value of null.

NUMBER—A column data type that contains a number, with or without a decimal point and a sign.

numeric constant—A fixed value that is a number.

ODBC—The Microsoft Open DataBase Connectivity (ODBC) standard, which is an application programming interface (API) specification written by Microsoft. It calls for all client applications to write to the ODBC standard API and for all database vendors to provide support for it. It then relies on third-party database drivers or access tools that conform to the ODBC specification to translate the ODBC standard API calls generated by the client application into the database vendor's proprietary API calls.

operator—A symbol or word that represents an operation to be performed on the values on either side of it. Examples of operators are
arithmetic (+, -, *, /), relational (=, !=, >, <, >=, <=), and logical (AND, OR, NOT).

optimization—The determination of the most efficient access strategy for satisfying a database access.

outer join—A join in which both matching and non-matching rows are returned. Each preserved row is joined to an imaginary row in the other table in which all the fields are null.

outer query—When a query is nested within another query, the main query is called the outer query and the inner query is called the subquery. An outer query is executed once for each row selected by the subquery. A subquery cannot be evaluated independently but that depends on the outer query for its results. Also see subquery.

page—The physical unit of disk storage that SQLBase uses to allocate space to tables and indexes.

parent table—The table containing the primary key.

parse—To examine a command to make sure that it is properly formed and that all necessary information is supplied.

partitioning—A method of setting up separate user areas to maximize disk space. Databases can be stretched across several different network partitions.

password—A sequence of characters that must be entered to connect to a database. Associated to each password is an authorization-id.

picture—A string of characters used to format data for display.

precedence—The default order in which operations are performed in an expression.

precision—The maximum number of digits in a column.

precompilation—Processing of a program containing SQL commands or procedures that takes place before compilation. SQL commands are replaced with statements that are recognized by the host language compiler. Output from precompilation includes source code that can be submitted to the compiler.

predicate—An element in a search condition that expresses a comparison operation that states a set of criteria for the data to be returned by a query.

primary key—The columns or set of columns that are used to uniquely identify each row in a table. All values for a key are unique and non-null.

privilege—A capability given to a user to perform an action.

procedure—A named set of SAL or SQL statements that can contain flow control language. You compile a procedure for immediate and/or later execution.

query—A request for information from a database, optionally based on specific conditions. For example, a request to list all customers whose balance is greater than $1000. Queries are issued with the SELECT command.

Read Only (RO)—The isolation level where pages are not locked, and no user has to wait. This gives the user a snapshot view of the database at the instant that the transaction began. Data cannot be updated while in the read-only isolation level.

Read Repeatability (RR)—The isolation level where if data is read again during a transaction, it is guaranteed that those rows would not have changed. Rows referenced by the program cannot be changed by other programs until the program reaches a commit point. Subsequent queries return a consistent set of results (as though changes to the data were suspended until all the queries finished). Other users will not be able to update any pages that have been read by the transaction. All shared locks and all exclusive locks are retained on a page until the transaction completes. Read repeatability provides maximum protection from other active application programs. This ensures a high level of consistency, but lowers concurrency. SQLBase default isolation level.

REAL—A column data type that stores a single-precision number.

record—See row.

recovery—Rebuilding a database after a system failure.

referential cycle—Tables which are dependents of one another.

referential integrity—Guarantees that all references from one database table to another are valid and accurate. Referential integrity prevents problems that occur because of changes in one table which are not reflected in another.

relation—See table.

relational database—A database that is organized and accessed according to relationships between data items. A relational database is perceived by users as a collection of tables.

relational operator—A symbol (such as =, >, or <) used to compare two values. Also called comparison operator.

Release Locks (RL)—With the Cursor Stability isolation level, when a reader moves off a database page, the shared lock is dropped. However, if a row from the page is still in the message buffer, the page is still locked.

In contrast, the Release Lock (RL) isolation level increases concurrency. By the time control returns to the application, all shared locks have been released.

repeating query—See correlated subquery.

requester—See client.

restore—Copying a backup of a database or its log files to a database directory.

restriction mode—In restriction mode, the result set of one query is the basis for the next query. Each query further restricts the result set. This continues for each subsequent query.

result set mode—Normally, result table rows are displayed and scrolled off the screen. In result set mode, the rows of the result table are available for subsequent scrolling and retrieval.

result table—The set of rows retrieved from one or more tables or views during a query. A cursor allows the rows to be retrieved one by one.

revoke—The act of withdrawing a user's permission to access a database.

rollback—To restore a database to the condition it was in at its last COMMIT. A ROLLBACK cancels a transaction and undoes any changes that it made to the database. All locks are freed unless cursor-context preservation is on.

rollforward—Reapplying changes to a database. The transaction log contains the entries used for rollforward.

router—A client application talks to a SQLBase server through a router program. The router enables a logical connection between a client and the server. Once this connection is established on the LAN, the client application uses the router program to send SQL requests to the server and to receive the results.

row—A set of related columns that describe a specific entity. For example, a row could contain a name, address, telephone number. Sometimes called record or tuple.

ROWID—A hidden column associated with each row in a SQLBase table that is an internal identifier for the row. The ROWID can be retrieved like any other column.

ROWID validation—A programming technique that ensures that a given row that was SELECTed has not been changed or deleted by another user during a session. When a row is updated, the ROWID is changed.

SAP (Service Advertisement Protocol)—A NetWare protocol that resources (such as database servers) use to publicize their services and addresses on a network.

savepoint—An intermediate point within a transaction to which a user can later ROLLBACK to cancel any subsequent commands, or COMMIT to complete the commands.

scale—The number of digits to the right of the decimal point in a number.

search condition—A criterion for selecting rows from a table. A search condition appears in a WHERE clause and contains one or more predicates.

search—To scan one or more columns in a row to find rows that have a certain property.

self-join—A join of a table with itself. The user assigns the two different correlation names to the table that are used to qualify the column names in the rest of the query.

self-referencing table—A table that has foreign and primary keys with matching values within the same table.

server—A computer on a network that provides services and facilities to client applications.

SHA (Secure Hash Algorithm)—A hash algorithm published by the United States government that SQLBase uses to detect unauthorized changes to a database page. SHA produces a condensed representation of a database page called a message digest that is used to generate a digital signature. When SQLBase reads a page encrypted with SHA, it verifies the signature. Any unauthorized changes to the page results in a different message digest and the signature will fail to verify. It is extremely unlikely to find a page that corresponds to a given message digest, or to find two different pages which produce the same message digest.

shared cursor—A handle that is used by two or more Windows applications.

shared lock (S-lock)—A shared lock permits other users to read data, but not to change it. A shared lock lets users read data concurrently, but does not let a user acquire an exclusive lock on the data until all the users' shared locks have been released. A shared lock is placed on a page when the page is read (during a SELECT). At a given time, more than one user can have a shared lock placed on a page. The timing of the release of a shared lock depends on the isolation level.

A shared lock differs from an exclusive lock because it permits more than one user to place a lock on the same data.

single-user—A computer system that can only provide its services to one user at a time.

SMALLINT—A column data type that stores numbers without decimal points.

socket—An identifier that Novell's IPX (Internetwork Packet Exchange) uses to route packets to a specific program.

SPX (Sequenced Packet Exchange)—A Novell communication protocol that monitors network transmissions to ensure successful delivery. SPX runs on top of Novell's IPX (Internetwork Packet Exchange).

SQL (Structured Query Language)—A standard set of commands used to manage information stored in a database. These commands let users retrieve, add, update, or delete data. There are four types of SQL commands

Data Definition Language (DDL), Data Manipulation Language (DML), Data Query Language (DQL), and Data Control Language (DCL). SQL commands can be used interactively or they can be embedded within an application program. Pronounced ess-que-ell or sequel.

SQLBase—A relational DBMS that lets users access, create, and update data.

SQLTalk—SQLTalk is an interactive user interface for SQLBase that is used to manage a relational database. SQLTalk has a complete implementation of SQL and many extensions. SQLTalk is a client application.

static SQL—See embedded SQL.

statistics—Attributes about tables such as the number of rows or the number of pages. Statistics are used during optimization to determine the access path to a table.

storage group—A list of database areas. Storage groups provide a means to allow databases or tables to be stored on different volumes.

stored procedure—A precompiled procedure that is stored on the backend for future execution.

string delimiter—A symbol used to enclose a string constant. The symbol is the single quote (').

string—A sequence of characters treated as a unit of data.

subquery—A SELECT command nested within the WHERE or HAVING clause of another SQL command. A subquery can be used anywhere an expression is allowed if the subquery returns a single value. Sometimes called a nested query. Also called subselect. See also correlated subquery.

synonym—A name assigned to a table, view, external function that may be then used to refer to it. If you have access to another user's table, you may create a synonym for it and refer to it by the synonym alone without entering the user's name as a qualifier.

syntax—The rules governing the structure of a command.

system catalog—A set of tables SQLBase uses to store metadata. System catalog tables contain information about database objects, privileges, events, and users. Also called data dictionary.

system keywords—Keywords that can be used to retrieve system information in commands.

table—The basic data storage structure in a relational database. A table is a two-dimensional arrangement of columns and rows. Each row contains the same set of data items (columns). Sometimes called a relation.

table scan—A method of data retrieval where a DBMS directly searches all rows in a table sequentially instead of using an index.

theta join—A join that uses relational operators to specify the join condition.

TIME—A column data type in the form of a value that designates a time of day in hours, minutes, and possibly seconds (a two- or three-part value).

timeout—A time interval allotted for an operation to occur.

TIMESTAMP—A column data type with a seven-part value that designates a date and time. The seven parts are year, month, day, hour, minutes, seconds, and microseconds (optional). The format is

yyyy-mm-dd-hh.mm.ss.nnnnnn

token—A character string in a specific format that has some defined significance in a SQL command.

Token-Ring—A LAN with ring topology (cable connected at the ends). A special data packet called a token is passed from one computer to another. When a computer gets the token, it can attach data to it and transmit. Each computer passes on the data until it arrives at its destination. The receiver marks the message as being received and sends the message on to the next computer. The message continues around the ring until the sender receives it and frees the token.

tokenized error message—An error message formatted with tokens in order to provide users with more informational error messages. A tokenized error message contains one or more variables that SQLBase substitutes with object names (tokens) when it returns the error message to the user.

transaction—A logically-related sequence of SQL commands that accomplishes a particular result for an application. SQLBase ensures the consistency of data by verifying that either all the data changes made during a transaction are performed, or that none of them are performed. A transaction begins when the application starts or when a COMMIT or ROLLBACK is executed. The transaction ends when the next COMMIT or ROLLBACK is executed. Also called logical unit of work.

transaction log—A collection of information describing the sequence of events that occur while running SQLBase. The information is used for recovery if there is a system failure. A log includes records of changes made to a database. A transaction log in SQLBase contains the data needed to perform rollbacks, crash recovery, and media recovery.

trigger—Activates a stored procedure that SQLBase automatically executes when a user attempts to change the data in a table, such as on a DELETE or UPDATE command.

two-phase commit—The protocol that coordinates a distributed transaction commit process on all participating databases.

tuple—See row.

unique key—One or more columns that must be unique for each row of the table. An index that ensures that no identical key values are stored in a table.

username—See authorization-id.

value—Data assigned to a column, a constant, a variable, or an argument.

VARCHAR—See CHAR.

variable—A data item that can assume any of a given set of values.

view—A logical representation of data from one or more base tables. A view can include some or all of the columns in the table or tables on which it is defined. A view represents a portion of data generated by a query. A view is derived from a base table or base tables but has no storage of its own. Data for a view can be updated in the same manner as for a base table. Sometimes called a virtual table.

wildcard—Characters used in the LIKE predicate that can stand for any one character (the underscore _) or any number of characters (the percent sign %) in pattern-matching.

Windows—A graphical user interface from Microsoft that runs under DOS.

With Windows, commands are organized in lists called menus. Icons (small pictures) on the screen represent applications. A user selects a menu item or an icon by pointing to it with a mouse and clicking.

Applications run in windows that can be resized and relocated. A user can run two or more applications at the same time and can switch between them. A user can run multiple copies of the same application at the same time.

write-ahead log (WAL)—A transaction logging technique where transactions are recorded in a disk-based log before they are recorded in the physical database. This ensures that active transactions can be rolled back if there is a system crash.

Index

reserved word 5-4
multiply 7-29

N

NAME
  reserved word 5-4
name
  authorization ID 2-3
  bind variables 2-5
  column name 2-4
  command 2-5
  correlation 2-4
  cursor A-18
    free A-2
    SqlExecute A-6
  database 2-4
  examples 2-2
  index 2-5
  password 2-5
  procedure 2-5, 3-103
  requirements 2-6
  see also identifier
  SQL command A-25
  synonym 2-5
  table 2-5
  types 2-3
  user 2-3
  view 2-6
naming conventions
  variables 7-9
nest
  triggers 3-54
NEW
  reserved word 5-4
NEW AS
  trigger 3-61
next row
  fetch A-7
NEXTVAL 2-19, 2-20
  reserved word 5-4
non-equijoin 2-40, 2-43
NOT 7-29
  reserved word 5-4
NOT NULL
  ALTER TABLE 3-11
  CREATE TABLE 3-48
NOT NULL WITH DEFAULT
  ALTER TABLE 3-11

CREATE TABLE 3-48
NOT operator 2-25
NULL 2-18, 3-46
  ALTER TABLE 3-11
  reserved word 5-4
  with foreign key 3-17, 3-47, 6-8, 6-9, 6-10
null
  definition 2-7
  in expressions 2-23
  search conditions 2-26
NULL predicate 2-30
NUMBER 3-45
  reserved word 5-4
number 2-7, 7-9, 7-10
NUMBER data type 2-10
numeric constant 2-17
numeric data type 2-7, 2-9

O

ODBC Glossary-11
OF
  reserved word 5-4
OFF
  reserved word 5-4
offset
  error position A-10
OLD
  reserved word 5-4
OLD AS
  trigger 3-61
ON
  reserved word 5-4
On 7-15
  example 7-18
ON CLIENT
  LOAD 3-99
  UNLOAD 3-136
ON COLUMN
  LABEL 3-94
ON DELETE
  ALTER TABLE (referential integrity) 3-18
  CREATE TABLE 3-49
ON SERVER
  LOAD 3-99
  UNLOAD 3-136
On statement
  where to specify 7-16
ON TABLE