User's Guide Volume 2:

Reference

Sybase SQL Anywhere

A System 11 Server Product

Notice of Copyright

Disclaimer

Sybase (Sybase, Inc. and all of its subsidiaries) makes no representation or warranty with respect to the adequacy of this documentation or the programs which it describes for any particular purpose or with respect to its adequacy to produce any particular result. In no event shall Sybase, its employees, its contractors or the authors of this documentation be liable for special, direct, indirect or consequential damages, losses, costs, charges, claims, demands, or claim for lost profits, fees or expenses of any nature or kind.

ISBN 1-55094-110-0

Printed in the United States of America

Preface

Developed on personal computers for personal computers, SQL Anywhere is a full-featured transaction-processing SQL database management system which has excellent performance while requiring less resources (memory space, disk space, and CPU speed) than other database management systems.

SQL Anywhere can be used as a standalone database management system or as a network database server in a client/server environment. The SQL Anywhere standalone engine is available for DOS, Windows 3.x, OS/2, Windows 95, and Windows NT. The SQL Anywhere network server is available for all of these operating systems as well as QNX, and as a NetWare Loadable Module (NLM). Database files are compatible between versions and across all operating systems.

SQL Anywhere is a fast and efficient database for many environments, from notebook computers to servers supporting large numbers of concurrent users. It's a flexible and scalable solution for today's diverse needs.

With an efficient ODBC driver and full ODBC 2.1 level 2 support, as well as other interfaces, SQL Anywhere is also an ideal database for developers. Whether you are using C, PowerBuilder, or other application development tools, and whether you are developing single-user or client/server applications, SQL Anywhere provides you with all the capabilities you expect from a full SQL database server.

Trademarks

Borland C++, **dBASE** and **Turbo C** are trademarks of Borland International.

Excel, **Microsoft**, **Microsoft C**, **Visual C++**, **Visual Basic**, **Access**, **Microsoft Mail**, **Microsoft Word**, and **Windows** are trademarks of Microsoft Corp.

IBM, **Lotus**, **OS/2**, **C Set ++**, **REXX** and **Presentation Manager** are trademarks of International Business Machines Corp.

Microsoft, **Windows**, **Windows '95**, and **Windows NT** are trademarks of Microsoft Corp.

NetWare, **NetWare 386**, and **Novell** are trademarks of Novell Inc.

CompuServe is a trademark of CompuServe, Inc.

QNX is a trademark of QNX Software Systems Ltd.

Other company and product names used herein may be the trademarks or registered trademarks of their respective owners.

Sybase trademarks

The following are trademarks or registered trademarks of Sybase, Inc, and its subsidiaries:

InfoMaker, **Open Server**, **PowerBuilder**, **Replication Server**, **SQL Anywhere**, **SQL Server**, **Transact-SQL**, **Watcom VX-REXX**, **Watcom C**, **Watcom SQL**.

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Reference

Chapter 37

SQL Anywhere Components

About this chapter

This chapter presents reference information on the programs and database administration utilities that are part of SQL Anywhere. The utilities can be accessed from SQL Central, from ISQL, or as command-line programs.

For comprehensive documentation on SQL Central, see the SQL Central online help. For an introduction to the SQL Central dabatase administration tool, see the chapter "Managing Databases With SQL Central" on page 33. For information on the SQL Anywhere Service Manager, see the chapter "Running programs as Windows NT Services" on page 423. The commands are shown in upper case in the summaries. In QNX, you must type them in lower case. For all other systems, either case can be used.

The SQL Anywhere programs use a set of system environment variables. These variables are described in "Environment variables" on page 682.

The SQL Anywhere programs also use a standard set of return codes. These return codes are described in "Software component return codes" on page 685.

Contents

- "The database compression utility" on page 724.
- "The Stop utility" on page 727.
- "The log translation utility" on page 728.
- "The Unload utility" on page 731.
- "The Upgrade utility" on page 736.
- "The validation utility" on page 738.
- "The DBWATCH server monitoring facility" on page 740.
- "The write file utility" on page 741.
- "The SQL Remote database extraction utility" on page 744.
- "The REBUILD batch or command file" on page 748.
- "The C language SQL preprocessor" on page 534.
- "The DBWATCH server monitoring facility" on page 740.

37.1 Environment variables

The following is a list of the environment variables that are used by SQL Anywhere and a description of where they are used.

SQLANY

Syntax

```
SQLANY = path
```

Description

The SQLANY environment variable is used to contain the directory where SQL Anywhere is installed. The default installation directory is **c:\sqlany50**. The install procedure automatically adds the SQLANY environment variable to your startup environment. The SQLANY variable is used by the batch files or command files that build the Embedded SQL examples.

In QNX, SQL Anywhere is installed in a fixed directory and the SQLANY variable is not required.

SQLCONNECT

Syntax

```
SQLCONNECT = keyword=value ; ...

SQLCONNECT = keyword#value ; ...
```

Description

The SQLCONNECT environment variable specifies connection parameters that are used by several of the database tools to connect to a database engine or network server. This string is a list of parameter settings of the form KEYWORD=**value**, delimited by semicolons. The number sign "#" is an alternative to the equals sign, and should be used when setting the connection parameters string in the SQLCONNECT environment variable, as using "=" inside an environment variable setting is a syntax error. The keywords are from the following table.

Verbose keyword	Short form
Userid	UID
Password	PWD
ConnectionName	CON
EngineName	ENG
DatabaseName	DBN
DatabaseFile	DBF
DatabaseSwitches	DBS
AutoStop	AutoStop
Start	Start
Unconditional	UNC

Figure 16. *Connection management keywords*

For a description of the connection parameters, see "Database connection parameters" on page 128.

SQLPATH

Syntax

```
SQLPATH = path;...
PATH = path;...
```

Description

ISQL searches along SQLPATH for ISQL command files and help files before searching the system path.

SQLREMOTE

Syntax

```
SQLREMOTE = <path>
```

Description

Addresses for the FILE message link in SQL Remote replication are subdirectories of the SQLREMOTE environment variable. This variable should point to a shared directory.

SQLSTART

Syntax

```
SQLSTART = start-line
SQLSTARTW = start-line
```

Description

Historical. The SQLSTART environment variable information has been added to the SQLCONNECT environment variable as the **Start** parameter.

TMP

Syntax

```
TMP = directory
TMPDIR = directory
TEMP = directory
```

Description

The database engine creates temporary files for various operations such as sorting and performing unions. These temporary files will be placed in the directory specified by the TMP, TMPDIR, or TEMP environment variables. (The database engine takes the first one of the three that it finds.)

If none of the environment variables is defined, temporary files are placed in the current directory.

37.2 Software component return codes

All database components use the following executable return codes. The header file **sqldef.h** has constants defined for these codes.

Code	Explanation
0	Success
1	General failure
2	Invalid file format, and so on
3	File not found, unable to open, and so on
4	Out of memory
5	Terminated by user
6	Failed communications
7	Missing required database name
8	Client/server protocol mismatch
9	Unable to connect to database engine
10	Database engine not running
11	Database server not found
254	Reached stop time
255	Invalid parameters on command line

Figure 17. *Executable Return Codes*

37.3 The database engine

Syntax

	DBENG50	[engine-switches]
	. . .	[database-file [database-switches], . . .]
	RTDSK50	[engine-switches]
	. . .	[database-file [database-switches], . . .]
Windows 3.x:	DBENG50W	[engine-switches]
	. . .	[database-file [database-switches], . . .]
	RTDSK50W	[engine-switches]
	. . .	[database-file [database-switches], . . .]
	DBENG50S	[engine-switches]
	. . .	[database-file [database-switches], . . .]
	RTDSK50S	[engine-switches]
	. . .	[database-file [database-switches], . . .]
QNX:	dbsrv50	[engine-switches]
	. . .	[[database-file [database-switches]], . . .]

Engine switches:	@filename	Read in switches from configuration file
	@envvar	Read in switches from environment variable
	-b	Run in bulk operations mode
	-c cache-size	Set maximum cache size
	-d	Use DOS I/O instead of direct I/O (DOS, Windows 3.x) Disable asynchronous I/O (OS/2, Windows NT, Windows 95)
	-e	Enable packet encryption
	-ga	Automatically shut down after last database closed
	-gb level	Set database process priority class to *level*
	-gc num	Set checkpoint timeout period
	-gd level	Set database starting permission
	-ge size	Sets the stack size for threads that run external functions
	-gf	Disable firing of triggers
	-gn num	Set number of threads
	-gp size	Set maximum page size
	-gr num	Set maximum recovery time
	-gs size	Set thread stack size
	-n name	Name the database engine
	-q	Quiet mode—suppress output
	-v	Log old values of all columns on UPDATE, for all databases

Recovery switches:		
	-a log-file	Apply named transaction log file
	-f	Force database to start without transaction log

Database switches:		
	-n name	Name the database
	-v	Log old values of all columns on UPDATE

Description

There are several versions of the SQL Anywhere database engine, and each version has a different executable name. The database engine for platforms other than Windows 3.x is named DBENG50.EXE, the 32-bit and 16-bit Windows 3.x executables being DBENG50W.EXE and DBENG50S.EXE respectively. The runtime database engine is named RTDSK50.EXE, with the 32-bit and 16-bit Windows 3.x executables being named RTDSK50W.EXE and RTDSK50S.EXE respectively.

The SQL Anywhere standalone database engine is completely compatible with the SQL Anywhere network server (DBSRV50.EXE). Client applications developed on one run against the other without alteration.

The 32-bit versions of the Windows 3.x database engine have much better performance than the 16-bit versions. They require Windows 3.x running in enhanced mode.

The database engine can be started with a number of database files or no database files. Command-line switches specified before any databases apply to the engine and all databases. Switches specified after a particular database file apply only to that database.

Client applications can load additional databases dynamically after the database engine has started. However, only databases of the same or smaller page size can be loaded dynamically.

If a *database-file* is specified without a file extension, SQL Anywhere first looks for *database-file* with extension .WRT (a write file), followed by *database-file* with extension .DB.

The database engines are programs that run as a separate task. In DOS, the database engines are DOS **terminate and stay resident** programs (TSRs), meaning that they will load into the memory of your computer and then return control to DOS. In QNX, there is no standalone database engine. DBSTART is just a link to DBSERVER.

Runtime database engines are available with the SQL Anywhere Desktop Runtime System for DOS, Windows 3.x, OS/2 and Windows 95 or NT. The runtime engine does not allow ALTER, CREATE, COMMENT or DROP commands. GRANT and REVOKE will allow you to add new users and change passwords but changing permissions on tables is not allowed. Also, the runtime database engine does not employ a transaction log (although it is a fully transaction-processing database engine) and does not support stored procedures and triggers.

Supplying command-line switches in a configuration file

A set of command line switches can be stored in a configuration file or in an environment variable. The database engine can be instructed to use these switches using the `@` command-line switch.

Engine switches

@filename Read in command-line switches from the supplied file. The file may contain line breaks, and may contain any set of command line switches. For example, the following command file holds a set of command line switches for an engine that starts with a cache size of 4Mb, a name of **myserver**, and loads the sample database:

```
-c 4096
-n myserver
c:\sqlany50\sademo.db
```

If this configuration file is saved as **c:\config.txt**, it can be used in an command line as follows:

```
DBENG50 @c:\config.txt
```

@filename

Read in command-line switches from the supplied environment variable. The environment variable may contain any set of command line switches. For example, the first of the following pair of statements sets an environment variable holding a set of command line switches for a database engine that starts with a cache size of 4Mb and loads the sample database. The second statement starts the database engine:

```
set envvar=-c 4096 c:\sqlany50\sademo.db

DBENG50 @envvar
```

-b

Use bulk operation mode. This is useful when loading large quantities of data into a database (see "Tuning bulk operations" on page 307). The database engine allows only one connection by one application. It does not keep a rollback log or a transaction log, and the multi-user locking mechanism is turned off.

-c cache-size

Set the size of the cache. The database engine uses extra memory for caching database pages if it is set aside in the cache. Any cache size less than 10000 is assumed to be K-bytes (1K = 1024 bytes). Any cache size 10000 or greater is assumed to be in bytes. The cache size may also be specified as nK or nM (1M = 1024K). By default, the database engine usess 2 megabytes of memory for caching. The more cache that can be given the engine, the better will be its performance.

-d

Under DOS and Windows 3.x a database engine using this switch uses normal DOS input and output instead of direct input and output. By default the database engine for DOS or Windows 3.x reads and writes database pages with low-level direct calls rather than normal DOS calls. When your database file is small (0-3Mb) and you have disk caching software running, DOS calls can be faster than direct calls. Direct I/O becomes faster for larger databases (>10Mb).

Disk access under Windows 3.11

Direct I/O is automatically disabled when running under Windows 3.11 with 32-bit disk access enabled. In this case, using the `-d` option merely prevents a warning message being displayed in the message window. The advantages of Windows 3.11 32-bit disk access compensate for the lack of direct I/O by the database engine.

Under other operating systems this flag has the following meaning:

Use synchronous I/O rather than asynchronous I/O. Asynchronous I/O is generally the preferred option.

-ga Applications can cause databases to be started and stopped by the engine. Specifying this switch causes the engine to shutdown when the last database is stopped.

-gb level OS/2 and Windows NT only. Set the database process priority class to *level*. Level must be one of *idle*, *normal* (the default), *high*, or *maximum*. *idle* is provided for completeness, and *maximum* may interfere with the running of your computer. *Low* and *high* are the commonly used settings.

-gc num Set the maximum desired length of time (in minutes) that the database engine will run without doing a checkpoint. See the description of the CHECKPOINT_TIME option in "SET OPTION Statement" on page 989. When a database engine is running with multiple databases, the checkpoint time specified by the first database started will be used unless overridden by this switch.

-gd level Set the database starting permission to *level*. This is the permission level required by a user to cause a new database file to be loaded by the engine. The level can be one of the following:

dba	Only users with DBA authority can start new databases.
all	All users can start new databases.
none	Starting new databases is not allowed.

The default setting is *all*.

-ge size Sets the stack size for threads running external functions, in bytes. The default is 16384 (16K). This switch is used only for OS/2, WIndows NT, and NetWare.f

-gf Disable firing of triggers by the engine.

-gn num Sets the number of execution threads that will be used in the database engine while running with multiple users. See the

description of the THREAD_COUNT in "SET OPTION Statement" on page 989. When a database engine is running with multiple databases, the thread count specified by the first database started will be used unless overridden by this switch.

-gp size

Sets the maximum page size allowed, in bytes. The size specified must be one of: 512, 1024, 2048, or 4096. Databases can be created with different internal page sizes. When the engine is first started, it will use the largest page size of all of the databases specified. Any attempts to load a database file with a larger page size later will fail unless this option is first specified.

-gr num

Set the maximum desired length of time (in minutes) that the database engine will take to recover from system failure. See to the description of the RECOVERY_TIME option in "SET OPTION Statement" on page 989. When a database engine is running with multiple databases, the recovery time specified by the first database started will be used unless overridden by this switch.

-gs size

Sets the stack size of every thread in the engine. The value entered is multiplied by four to produce the stack size in bytes.

-n name

Set the name of the database engine. By default, the database engine receives the name of the database file with the path and extension removed. For example, if the engine is started on **c:\sqlany50\sademo.db** and no `-n` switch is specified, then the name of the engine will be **sademo**.

The engine name can be used on the connect statement (see the chapter "Connecting to a database") to specify to which engine you wish to connect. In all environments, there is always a default database engine that will be used if no engine name is specified provided at least one database engine or SQL Anywhere Client (DBCLIENT) is running on the computer.

-q

Operate quietly. Suppress all output.

Recovery switches

-a log-file

Apply the named transaction log. This is used to recover from media failure on the database file (see "Backup and Data Recovery" on page 331). When this option is specified, the database engine will apply the log and then terminate—it will not continue to run.

-f

This option is used for recovery: either to force the database engine to start after the transaction log has been lost (see "Backup and Data Recovery" on page 331), or to force the database engine to start using a transaction log it would otherwise not find.

If there is no transaction log, the database engine carries out a checkpoint recovery of the database and then terminates—it does not continue to run. You can then restart the database engine without the `-f` option for normal operation.

If there is a transaction log in the current directory, the database engine carries out a checkpoint recovery, and a recovery using the transaction log, and then terminates—it does not continue to run. You can then restart the database engine without the `-f` option for normal operation.

Database switches

-n name

Set the name of the database. Both database engines and databases can be named. Since a database engine can load several databases, the database name is used to distinguish the different databases.

By default, the database receives the name of the file with the path and extension removed. For example, if the engine is started on **c:\sqlany50\sademo.db** and no `-n` switch is specified, then the name of the database is **sademo**.

-v

Cause the engine to record the previous values in the transaction log of each of the columns whenever a row of the database is updated. By default, the engine will only record enough information to uniquely identify the row (primary key values or values from a not null unique index). This switch is useful for working on a copy of a database file (see "Transactions and portable computers" on page 213).

The backup, initialization, unload, and validation utilities automatically load the database engine if it has not been previously loaded. The SQL Anywhere Client (DBCLIENT) can also be started automatically in this way. See "Environment variables" on page 682 and the respective commands for more details. If the database engine (or client) is started automatically, it will be unloaded automatically when the software is finished executing.

37.4 The ISQL utility

Syntax

	ISQL	[switches] [isql-command]
	RTSQL	[switches] [isql-command]
Windows 3.X:	ISQLW	[switches] [isql-command]
	RTSQLW	[switches] [isql-command]

switches:

-b	Do not print banner
-c "keyword=value; ..."	Supply database connection parameters
-q	Quiet mode—no windows or messages
-s "cmd"	Specify command line to start database
-v	Verbose—output information on commands
-x	Syntax check only—no commands executed

See also

"The database engine" on page 685

Description

ISQL provides the user with an interactive environment for database browsing and for sending SQL statements to SQL Anywhere.

ISQL allows you to type SQL commands, or run ISQL command files. It also provides feedback about the number of rows affected, the time required for each command, the execution plan of queries, and any error messages.

If *isql-command* is specified, then ISQL executes the command. The most common form uses the READ statement to execute an ISQL command file in batch mode. If no *isql-command* is specified, then ISQL enters the interactive mode where you can type a command into a command window.

The Windows 3.x executable name is ISQLW. You can set up Program Manager icons to start ISQLW with the same command line syntax as specified above.

RTSQL and RTSQLW are runtime SQL command processors. These programs are included in the SQL Anywhere Runtime System for DOS, Windows, OS/2, or Windows NT. They are like ISQL except there is no interactive part; they can only run command files. By including RTSQL commands, SQL command files can be run from operating system batch or command files. RTSQL automatically loads the database engine if it is not already loaded. It unloads the database engine on exit. The command line switches are the same as those of ISQL.

Switches

-b Do not print the identification banner when ISQL starts up.

-c "keyword=value; ..."
Specify connection parameters. See "Database connection parameters" on page 128 for a description of the connection parameters. If this option is not specified, the environment variable SQLCONNECT is used. If required connection parameters are not specified, then you are presented with a dialog to enter the connection parameters.

-q Operate quietly. ISQL does not display any information on the screen. This is only useful if a command or command file is executed when starting ISQL.

-s "cmd" Historical. The start command was added to the connection parameters as the **Start** parameter.

-v Verbose output. Information about each command is displayed as it is executed (RTSQL only).

-x Scan commands but do not execute them. This is useful for checking long command files for syntax errors.

Commands available in ISQL

ISQL commands are broken into the following groups:

- Standard SQL statements that are part of the Watcom-SQL language. These commands are further broken into two groups, the data manipulation commands and the data definition commands.
- Commands that are particular to ISQL and manipulate the ISQL environment.
- Simple commands that manipulate the data window.

Detailed descriptions of SQL statements and ISQL commands can be found in the chapter "Watcom-SQL Language Reference" on page 751.

The ISQL commands are as follows:

CONFIGURE Statement
Activate the ISQL configuration window for displaying and changing ISQL options

CONNECT Statement
Reconnect to the database engine with a different user ID and password

DBTOOL Statement
Invoke one of the database tools

DISCONNECT Statement
Disconnect from the database engine

EXIT Statement
Leave ISQL

HELP Statement
Enter the on-line help facility

INPUT Statement
Do mass input into database tables

OUTPUT Statement
Output the current query results to a file

PARAMETERS Statement
Specify the parameters to a command file

QUIT Statement
Leave ISQL

READ Statement
Execute ISQL command files

SET CONNECTION Statement
Change the active database connection

SET OPTION Statement
Set options that control the ISQL environment

SYSTEM Statement
Executes an operating system command (not available in Windows 3.X).

The data window manipulation commands are as follows: These commands are not described in the command summary section.

CLEAR Statement
Clear the data window

DOWN [n] Move the current query results down **n** lines. The default is one line.

UP [n] Move the current query results up **n** lines. The default is one line.

37.4.2 Starting ISQL from SQL Central

You can start ISQL from SQL Central in the following ways:

- Right-click a database, and select Open ISQL from the popup menu.
- Right-click a table, and select View Data from the popup menu. ISQL opens with the data in the table displayed in the Data window.
- Right-click a stored procedure, and select Test from the popup menu. ISQL opens with a test script in the Command window.

37.5 The backup utility

With the backup utility, you can back up running databases, database files, transaction logs, and write files. For a description of suggested backup procedures, see the chapter "Backup and Data Recovery" on page 331.

You can access the backup utility in the following ways:

- From SQL Central, for interactive use under Windows 95 or NT.
- From the ISQL Database Tools window, for interactive use under Windows 3.x or OS/2.
- From the ISQL DBTOOL statement, for interactive use under any supported operating system.
- From the system command line, using the **dbbackup** utility. This is useful for incorporating backup procedures into batch or command files.

The backup utility makes a backup copy of all the files making up a single database. A simple database consists of two files: the main database file and the transaction log. More complicated databases can store tables in multiple files. Each file is a separate *dbspace*. All backup filenames are the same as the database filenames.

Running the backup utility on a running database is equivalent to copying the database files when the database is not running. It is provided to allow a database to be backed up while other applications or users are using the database.

37.5.1 Backing up a database from SQL Central

For full information on backing up a database from SQL Central, see the SQL Central online help.

You can back up a running database as follows:

1. Connect to the database.
2. Right-click the database and click Backup in the popup menu. The Backup Wizard is displayed.
3. Follow the instructions in the Wizard.

You can back up a database file or a running database as follows:

1. Open the Database Utilities folder in the left panel.
2. Double-click Backup Database in the right panel.
3. Follow the instructions in the wizard.

37.5.2 Backing up a database from the ISQL Tools window

To use the backup utility from the ISQL Tools window:

1. Select Database Tools from the Window menu.
2. Click Backup Database Files on the Tools list.
3. Enter a user ID and password to use when connecting to the database.
4. For a running database, enter a database name and server name (if more than one is running). For a database file, enter the file name (with path) and optionally a start line to specify command-line switches for the database engine or SQL Anywhere Client.
5. Click Backup, and select from the options displayed in the dialog.
6. Click OK to back up the database.

37.5.3 Backing up a database using the DBTOOL statement

The syntax to access the backup utility from the ISQL DBTOOL statement is as follows:

Syntax

```
DBTOOL BACKUP TO directory

...     | [ DBFILE ] [ WRITE FILE ] [ [ TRANSACTION ] LOG ]      |
        | [ ALL FILES ]                                         |

...     | [ RENAME [ TRANSACTION ] LOG ]                        |
        | [ TRUNCATE [ TRANSACTION ] LOG ]                      |

...     [ NOCONFIRM ] USING connection-string
```

Example

The following statement connects to and backs up the sample database, to directory **c:\temp**.

```
DBTOOL BACKUP TO 'c:\temp' DBFILE
USING 'dbf=c:\sqlany50\sademo.db;uid=dba;pwd=sql'
```

37.5.4 The DBBACKUP command-line utility

The DBBACKUP command-line switches are as follows:

Syntax

```
                DBBACKUP  [switches] directory

Windows 3.x:    DBBACKW   [switches] directory

switches:       -c "keyword=value; ..."
                                Supply database connection parameters
                -d              Only back up main database file
                -q              Quiet mode—do not print messages
                -r              Rename and restart transaction log
                -t              Only back up transaction log
                -w              Only back up write file
                -x              Delete and restart transaction log
                -y              Replace files without confirmation
```

If none of the switches `-d`, `-t`, or `-w` are used, all database files are backed up.

37.5.5 Backup utility options

Connection parameters

For a description of the connection parameters, see "Database connection parameters" on page 128. If the connection parameters are not specified, connection paremeters from the SQLCONNECT environment variable are used, if set. The **user ID** must have DBA authority.

For the DBBACKUP utility, this is the `-c` command-line switch. For example, the following statement backs up the **sademo** database running on the **sample_server** server, connecting as user ID DBA with password SQL:

```
dbbackup -c
"eng=sample_server;dbn=sademo;uid=dba;pwd=sql"
```

Backup main database only

Back up the main database files only, without backing up the transaction log file or a write file, if one exists. For the DBBACKUP utility, this is the `-d` command-line switch.

Operate quietly

Do not display messages on a window For the DBBACKUP utility, this is the `-q` command-line switch. This option is not available from other environments.

Rename transaction log, start new log

With this option, the existing transaction log is backed up and renamed, and a new transaction log is started. The renamed transaction log (on the server) has the same file name as the transaction log, but with an extension of **nnn**, where **nnn** is the first available number starting at 000. DBBACKUP displays the name of the renamed transaction log. This option causes the backup to wait for a point when all transactions from all connections are committed. For the DBBACKUP utility, this is the `-r` command-line switch.

Back up the transaction log file only

This can be used as an incremental backup since the transaction log can be applied to the most recently backed up copy of the database file(s). For the DBBACKUP utility, this is the `-t` command-line switch.

Back up the database write file only

See DBWRITE for a description of database write files. For the DBBACKUP utility, this is the `-w` command-line switch.

Delete and restart the transaction log

With this option, the existing transaction log is backed up, then the original is deleted and a new transaction log is started with the same name. This option causes the backup to wait for a point when all transactions from all connections are committed. For the DBBACKUP utility, this is the `-x` command-line switch.

Operate without confirming actions

Without this option, you are prompted to confirm the creation of the backup directory or the replacement of a previous backup file in the directory. For the DBBACKUP utility, this is the `-y` command-line switch.

37.6 The SQL Anywhere Client

In a client/server arrangement using the SQL Anywhere network server, the SQL Anywhere Client program runs on the client application computer. The client application connects to the SQL Anywhere Client, which manages the interface to the network server.

Syntax

```
                DBCLIENT [switches] server-name
Windows 3.x:    DBCLIENW [switches] server-name
```

See also

"The database engine" on page 685.

Description

See the *SQL Anywhere Network Guide* for a description of the SQL Anywhere Client.

The DOS and Windows 3.x SQL Anywhere Clients can be used to access an OS/2 or Windows 95 or NT database engine running on the same machine. See "DOS or Windows client applications on OS/2" on page 22 and "DOS or Windows 3.x client applications on Windows 95 or Windows NT" on page 22.

37.7 The collation utility

With the collation utility, you can extract a collation (sorting sequence) from the SYS.SYSCOLLATION system table of a database into a file suitable for creating a database using a custom collation.

The file produced by the collation utility can be modified and used with SQL Central or the `-z` option of DBINIT to create a new database with a custom collation.

Ensure that the label is changed on the line that looks like:

```
Collation label (name)
```

Otherwise, the collation cannot be used to create a database.

If you wish to create a custom collation but have not yet created a database, you should extract a collation from the sample database provided with SQL Anywhere. For a full description of custom collating sequences, see "Database Collations" on page 287.

You can access the collation utility in the following ways:

- From the ISQL Database Tools window, for interactive use under Windows 3.x or OS/2.
- From the ISQL DBTOOL statement, for interactive use under any supported operating system.
- From the system command line, using the **dbcollat** utility.

37.7.1 Extracting a collation in the ISQL Database Tools window

To use the collation utility from the ISQL Database Tools window:

1. Select Database Tools from the Window menu.
2. Click Extract Collation from Database on the Tools list.
3. For a running database, enter a database name and server name (if more than one is running). For a database file, enter the file name (with path) and optionally a start line to specify command-line switches for the database engine or SQL Anywhere Client.
4. Click Extract, and select from the options displayed in the dialog.
5. Click OK to extract the collation file from the database.

37.7.2 Extracting a collation using the DBTOOL statement

The syntax to access the collation utility from the ISQL DBTOOL statement is as follows:

Syntax

```
DBTOOL UNLOAD COLLATION [ name ] TO filename
    ...    USING connection-string
    ...    [ EMPTY MAPPINGS ] [ HEX | HEXADECIMAL ] [ NOCONFIRM ]
```

37.7.3 The DBCOLLAT command-line utility

The DBCOLLAT command-line switches are as follows:

Syntax

```
                DBCOLLAT  [switches] output-file

Windows 3.x:    DBCOLLW   [switches] output-file

switches:       -c "keyword=value; ..."
                                Supply database connection parameters
                -e              Include empty mappings
                -o filename     Output log messages to file
                -q              Quiet mode — do not print messages
                -x              Use hex for extended characters (7F-FF)
                -y              Replace file without confirmation
                -z col-seq      Specify collating sequence label
```

37.7.4 Collation utility options

Connection parameters

For a description of the connection parameters, see "Database connection parameters" on page 128. If the connection parameters are not specified, connection paremeters from the SQLCONNECT environment variable are used, if set. The **user ID** must have DBA authority.

For the DBCOLLAT utility, this is the `-c` command-line switch. For example, the following statement extracts a collation file from

the **sademo** database running on the **sample_server** server, connecting as user ID DBA with password SQL:

```
dbcollat -c
"eng=sample_server;dbn=sademo;uid=dba;pwd=sql"
c:\sample.col
```

Include empty mappings

Normally, collations don't specify the actual value that a character is to sort to. Instead, each line of the collation sorts to one position higher than the previous line. However, older collations have gaps between some sort positions. Normally, the collatino utility skips the gaps and writes the next line with an explicit sort-position. This option causes the collation utility to write empty mappings (consisting of just a colon(:)) for each line in the gap. For the DBCOLLAT utility, this is the `-e` command-line switch.

Output log messages to file

Redirect the log messages from the collation utility to a named file. For the DBCOLLAT utility, this is the `-o` command-line switch.

Operate quietly

Do not display messages on a window. For the DBCOLLAT utility, this is the `-q` command-line switch. This option is not available from other environments.

Use hexadecimal for extended characters (7F to FF)

Extended single-byte characters (whose value is greater than hex 7F) may or may not appear correctly on your screen, depending on whether or not the code page in use on your computer is the same as the code page being used in the collation you are extracting. This option causes the collation utility to write all characters at hex 7F or above as a two-digit hexadecimal number, in the form:

```
\xdd
```

(For example, \x80, \xFE). Normally, only characters from hex 00 to hex 1F, hex 7F and hex FF are written in hexadecimal form. For the DBCOLLAT utility, this is the `-x` command-line switch.

Operate without confirming actions.

Without this option, you are prompted to confirm replacing an existing collation file. For the DBCOLLAT utility, this is the `-y` command-line switch.

Specify collating sequence label

Specify the label of the collation to be extracted. The names of the collation sequences can be found in the **collation_label** column of the SYS.SYSCOLLATION table. If this option is not specified, then the collation utility extracts the collation being used by the database. For the DBCOLLAT utility, this is the `-z` command-line switch.

37.8 The Erase utility

With the Erase utility, you can erase a database file or write file and its associated transaction log, or you can erase a transaction log file or transaction log mirror file. All SQL Anywhere database files, write files, and transaction log files are marked **read-only** to prevent accidental damage to the database or accidental deletion of the database files. Deletion of a database file that references other dbspaces does not automatically delete the dbspace files.

If you erase a database file or write file, the associated transaction log and transaction log mirror are also deleted. If you erase a transaction log for a database that also maintains a transaction log mirror, the mirror is not deleted.

You can access the Erase utility in the following ways:

- From SQL Central, for interactive use under Windows 95 or NT.
- From the ISQL Database Tools window, for interactive use under Windows 3.x or OS/2.
- From the ISQL DBTOOL statement, for interactive use under any supported operating system.
- From the system command line, using the **dberase** utility. This is useful for incorporating into batch or command files.

37.8.1 Erasing a database from SQL Central

For full information on erasing a database from SQL Central, see the SQL Central online help. You can erase a database file as follows:

1. Open the Database Utilities folder in the left panel.
2. Double-click Erase Database in the right panel. The Erase a Database Wizard is displayed.
3. Follow the instructions in the Wizard.

37.8.2 Erasing a database from the ISQL Tools window

To use the erase utility from the ISQL Tools window:

1. Select Database Tools from the Window menu.
2. Click Erase Database or Write File on the Tools list.
3. Enter the file name (with path).
4. Click Erase, and select from the options displayed in the dialog.
5. Click OK to erase the selected files.

37.8.3 Erasing a database using the DBTOOL statement

The syntax for the erase utility from the ISQL DBTOOL statement is as follows:

Syntax

```
DROP DATABASE database-file [ NOCONFIRM ]
```

The *database-file* may be a database file, write file, or transaction log file. The full file name must be specified, including extension. If a database file or write file is specified, the associated transaction log file (and mirror, if one is maintained) is also erased.

Example

The following statement erases the database file **c:\temp.db** and its associated transaction log file, prompting to confirm deletion.

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

37.8.4 The DBERASE command-line utility

The DBERASE command-line switches are as follows:

Syntax

```
                DBERASE   [switches] database-file

Windows 3.x:    DBERASEW  [switches] database-file

switches:       -o        Output log messages to file
                -q        Operate quietly—do not print messages
                -y        Erase files without confirmation
```

The *database-file* may be a database file, write file, or transaction log file. The full file name must be specified, including extension. If a database file or write file is specified, the associated transaction log file (and mirror, if one is maintained) is also erased.

37.8.5 Erase utility options

Output log messages to file
: Redirect log messages to the named file. For the DBERASE utility, this is the `-o` option.

Operate quietly
: Do not display messages on a window For the DBERASE utility, this is the `-q` command-line switch. This option is not available from other environments.

Operate without confirming actions
: Without this option, you are prompted to confirm the deletion of each file. For the DBERASE utility, this is the `-y` command-line switch.

37.9 The database uncompression utility

With the uncompression utility you can expand a compressed database file created by the compression utility. The uncompression utility reads the given compressed database file and creates an uncompressed database file.

The uncompression utility does not uncompress files other than the main database file (dbspace files).

You can access the uncompression utility in the following ways:

- From SQL Central, for interactive use under Windows 95 or NT.
- From the ISQL Database Tools window, for interactive use under Windows 3.x or OS/2.
- From the ISQL DBTOOL statement, for interactive use under any supported operating system.
- From the system command line, using the **dbexpand** utility. This is useful for incorporating into batch or command files.

37.9.1 Uncompressing a database in SQL Central

For full information on uncompressing a database in SQL Central, see the SQL Central online help.

You can uncompress a compressed database file as follows:

1. Open the Database Utilities folder in the left panel.
2. Double-click Uncompress Database in the right panel. The Uncompress Wizard is displayed.
3. Follow the instructions in the Wizard.

37.9.2 Uncompressing a database from the ISQL Tools window

To use the uncompression utility from the ISQL Tools window:

1. Select Database Tools from the Window menu.
2. Click Uncompress Database on the Tools list.
3. Enter the database file name (with path).
4. Click Uncompress, and enter a file name to use for the uncompressed database.
5. Click OK to uncompress the selected database file.

37.9.3 Uncompressing a database using the DBTOOL statement

The syntax for the uncompression utility from the ISQL DBTOOL statement is as follows:

Syntax

```
DBTOOL UNCOMPRESS DATABASE compressed-database-file
   ...    [ TO filename ] [ NOCONFIRM ]
```

Example

The following statement uncompresses the compressed database file **c:\temp.cdb**, creating the database file **c:\:temp.db**

```
DBTOOL UNCOMPRESS DATABASE 'c:\temp.cdb'
```

37.9.4 The DBEXPAND command-line utility

Syntax

DBEXPAND [switches] compressed-database-file [database-file]

Windows 3.x: DBEXPANW [switches] compressed-database-file [database-file]

switches:	-q	Operate quietly—do not print messages
	-y	Erase existing output file without confirmation

The input filename extension defaults to .cdb. The output filename (with extension) must not have the same name as the input filename (with extension).

37.9.5 Uncompression utility options

Output log messages to file
Redirect log messages to the named file. For the DBEXPAND utility, this is the `-o` option.

Operate quietly
Do not display messages on a window. For the DBEXPAND utility, this is the `-q` command-line switch. This option is not available from other environments.

Operate without confirming actions
Without this option, you are prompted to confirm the replacement of an existing database file. For the DBEXPAND utility, this is the `-y` command-line switch.

37.10 The database information utility

With the information utility you can display information about a database file or write file. The database should not be running when you run the information utility. The information includes the options specified when the database was created, the name of any transaction log file or log mirror that is maintained, and other information.

You can access the information utility in the following ways:

- From the ISQL Database Tools window, for interactive use under Windows 3.x or OS/2.

- From the ISQL DBTOOL statement, for interactive use under any supported operating system.
- From the system command line, using the **dbinfo** utility.

37.10.1 Obtaining database information in the ISQL Tools window

To use the information utility from the ISQL Tools window:

1. Select Database Tools from the Window menu.
2. Click Database Information on the Tools list.
3. Enter the database file name (with path), user ID and password.
4. Click Display to show information about the database.
5. Click Page Usage to show information on database pages.

37.10.2 Obtaining database information using the DBTOOL statement

The syntax for the information utility from the ISQL DBTOOL statement is as follows:

Syntax

```
DBTOOL DBINFO DATABASE database-file TO output-file
   ...    [ [ WITH ] PAGE USAGE ]
   ...    USING connection-string
```

A database or write file name, with path, must be supplied. The database should not be running when you run the information utility.

Example

The following statement gets information about the sample database and puts it in the file **c:\temp.txt**.

```
DBTOOL   DBINFO DATABASE 'c:\sqlany50\sademo.db'
TO       'c:\temp.txt'
USING    'uid=dba;pwd=sql'
```

37.10.3 The DBINFO command-line utility

Syntax

DBINFO [switches] filename

Windows 3.x: DBINFOW [switches] filename

switches:

-c "keyword=value; ..."	Supply database connection parameters
-o filename	Output messages to file
-q	Suppress any messages
-u	Output page usage statistics

37.10.4 Information utility options

Database file name

A database or write file name, with path, must be supplied. The database should not be running when you run the information utility.

Connection parameters

Connection parameters are required only if page usage statistics are requested. For a description of the connection parameters, see "Database connection parameters" on page 128. If the connection parameters are not specified, connection paremeters from the SQLCONNECT environment variable are used, if it is set. The user ID specified need only have SELECT permissions on the system tables (granted to PUBLIC by default). For the DBINFO utility, this is the `-c` command-line switch.

Output log messages to file

Redirect log messages to the named file. For the DBINFO utility, this is the `-o` command-line switch.

Operate quietly

Do not display the information. The return code may still provide useful information (see "Software component return codes" on page 685). For the DBINFO utility, this is the `-q` command-line switch. This option is not available from other environments.

Page usage statistics

The information utility must establish a connection with the database in order to get system table information for the page

usage statistics. DBINFO starts the database engine, get the system table information, and then stops the database engine. The connection parameters are used only if page usage statistics are requested.

37.11 The database initialization utility

With the initialization utility you can initialize (create) a database. A number of database attributes are specified at initialization and cannot be changed later except by unloading, reinitializing, and rebuilding the entire database:

- Case sensitivity or insensitivity.
- Storage as an encrypted file.
- Treatment of trailing blanks in comparisons.
- The page size.
- The collation sequence used.

In addition, the choice of whether to use a transaction log and a transaction log mirror is made at initialization. This choice can be changed later using the transaction log utility.

You can access the initialization utility in the following ways:

- From SQL Central, for interactive use under Windows 95 or NT.
- From the ISQL Database Tools window, for interactive use under Windows 3.x or OS/2.
- From the ISQL DBTOOL statement, for interactive use under any supported operating system.
- From the system command line, using the DBINIT utility. This is useful for incorporating into batch or command files.

37.11.1 Creating a database in SQL Central

For full information on creating a database in SQL Central, see the SQL Central online help.

You can create a database as follows:

1. Open the Database Utilities folder in the left panel.
2. Double-click Create Database in the right panel. The Database Creation Wizard is displayed.
3. Follow the instructions in the Wizard.

37.11.2 Creating a database from the ISQL Tools window

To use the initialization utility from the ISQL Tools window:

1. Select Database Tools from the Window menu.
2. Click Create Database on the Tools list.
3. Enter the database file name (with path).
4. Click Create, and and select from the options displayed in the dialog.
5. Click OK to create the database.

37.11.3 Creating a database using the DBTOOL statement

The syntax for the initialization utility from the ISQL DBTOOL statement is as follows:

Syntax

```
DBTOOL CREATE DATABASE filename

...    | [ NO [ TRANSACTION ] LOG ]                    |
       | [ [ TRANSACTION ] LOG TO filename ]           |

...    | [ IGNORE CASE ]                               |
       | [ RESPECT CASE ]                              |

...    [ PAGE SIZE n ] [ COLLATION name ]

...    [ ENCRYPT ] [ TRAILING SPACES ]
```

37.11.4 The DBINIT command-line utility

Syntax

DBINIT [switches] new-database-file

Windows 3.x: DBINITW [switches] new-database-file

switches:		
	-b	Blank padding of strings for comparisons
	-c	Case sensitivity for all string comparisons
	-e	Encrypt database
	-g user	User name as replacement for DBO
	-l	List available collating sequences
	-m file-name	Use a transaction log mirror (default is no mirror).
	-n	No transaction log
	-o filename	Output messages to file
	-p page-size	Set page size
	-q	Quiet mode—do not print messages
	-t log-name	Transaction log filename (default is database name with .log)
	-z col-seq	Collation sequence used for comparisons

For example, the database **test.db** can be created with 512 byte pages as follows:

```
dbinit -p 512 test.db
```

37.11.5 Initialization utility options

Ignore trailing blanks

Trailing blanks are ignored for comparison purposes, and Embedded SQL programs pad strings fetched into character arrays. For example, the two strings

```
'Smith'
'Smith
```

would be treated as equal in a database created with trailing blanks ignored.

This option is provided for compatibility with the ISO/ANSI SQL standard, which is to ignore trailing blanks in comparisions. The default is that blanks are significant for comparisons, which was the only behavior supported in releases up to Watcom SQL Version 3.0. For the DBINIT utility, this is the `-b` command-line switch.

Case sensitivity for all string comparisons

For databases created with this option, all identifiers (table names, column names, and so on) and values are considered to be case sensitive in comparisons and string operations, so that **emp_id** is not the same as **emp_ID**.

This option is provided for compatibility with the ISO/ANSI SQL standard. The default is that all comparisons are case insensitive, which was the only behavior supported up to release Watcom SQL Version 3.0. For the DBINIT utility, this is the `-c` command-line switch.

User ID and password

All databases are created with one user ID: **DBA** with password **SQL**. If you specify create a database requiring case-sensitive comparisons, the **DBA** user ID and its password must be entered in upper case.

Encrypt database

Encryption makes it more difficult for someone to decipher the data in your database by using a disk utility to look at the file. File compaction utilities are not able to compress encrypted database files as much as unencrypted ones. For the DBINIT utility, this is the `-e` command-line switch.

User ID for DBO user

SQL Anywhere contains a set of system views that mimic the system tables of Sybase SQL Server. By default, the owner of these views is the user ID **DBO**, which is the same as the owner of the SQL Server system tables. If you already have a user ID named **DBO**, or you wish to use that user ID for other purposes, this option allows you to provide an alternative user ID for the owner of the SQL Server-like system views. For the DBINIT utility, this is the `-g` command-line switch.

Use a transaction log mirror

A transaction log mirror is an identical copy of a transaction log, usually maintained on a separate device, for greater protection of your data. By default, SQL Anywhere does not use a mirrored transaction log. If you do wish to use a transaction log mirror, this option allows you to provide a file name. For the DBINIT utility, this is the `-m` command-line switch.

Do not use a transaction log

Creating a database without a transaction log is more economical on disk space. The transaction log is required for data replication

and provides extra security for database information in case of media failure or system failure. For the DBINIT utility, this is the `-n` command-line switch.

Page size
The *page size* for a SQL Anywhere database can be 512, 1024, 2048 or 4096 bytes, with 1024 being the default. Other values for the size will be changed to the next larger size. Large databases usually benefit from a larger page size. For the DBINIT utility, this is the `-p` command-line switch.

Output log messages to file
Redirect log messages to the named file. For the DBINIT utility, this is the `-o` command-line switch.

Operate quietly
Do not display messages on a window. For the DBINIT utility, this is the `-q` command-line switch. This option is not available from other environments.

Set the transaction log filename
The transaction log is a file where the database engine logs all changes made by all users no matter what application system is being used. The transaction log plays a key role in backup and recovery (see "The transaction log" on page 334), and in data replication. If the filename has no path, it is placed in the same directory as the database file. For the DBINIT utility, this is the `-t` command-line switch. If you run DBINIT without specifying `-t` or `-n`, a transaction log is created with the same filename as the database file, but with extension .log.

Collating sequence or collation filename
The collation sequence is used for all string comparisons in the database.

The database is created with all collations listed in SYS.SYSCOLLATION.

If you want to use a custom collation, then you should use the collation utility to extract the closest collation from the database, then modify the collation file and use the initialization utility to specify the new collation. It is important to change the collation label in the custom collation file, otherwise the initialization utility prevents the insertion of the new collation, since it will be a duplicate label. For a full description of custom collating sequences, see "Database Collations" on page 287.

In order to change the collation that an existing database is using, it is necessary to unload the database, create a new database using the appropriate collation, then reload the database.

For the DBINIT utility, this is the `-z` command-line switch. If `-z` is specified and names one of the collations then that collation will be used by the database. If `-z` is specified but does not name one of the collations, then it is assumed that the name is a filename, and the file will be opened and the collation will be parsed from the file. The specified collation will then be used by the database. If `-z` is not specified, then the default collation is used. Normal ASCII (binary) ordering is used for the lower 128 characters. For the upper 128 characters (also called the extended characters), characters which are accented forms of a letter in the lower 128 are sorted to the same position as the unaccented form. The determination of whether or not an extended character is a letter is based upon code page 850 (multilingual code page).

The following table identifies the available collating sequence labels. All collating sequences except EBCDIC do not affect the normal ASCII character set.

Label	Explanation
437EBCDIC	(Code Page 437, EBCDIC)
437LATIN1	(Code Page 437, Latin 1)
437ESP	(Code Page 437, Spanish)
437SVE	(Code Page 437, Swedish/Finnish)
850CYR	(Code Page 850, Cyrillic)
850DAN	(Code Page 850, Danish)
850ELL	(Code Page 850, Greek)
850ESP	(Code Page 850, Spanish)
850ISL	(Code Page 850, Icelandic)
850LATIN1	(Code Page 850, Latin 1)
850LATIN2	(Code Page 850, Latin 2)
850NOR	(Code Page 850, Norwegian)
850RUS	(Code Page 850, Russian)
850SVE	(Code Page 850, Swedish/Finnish)
850TRK	(Code Page 850, Turkish)
852LATIN2	(Code Page 852, Latin 2)
852CYR	(Code Page 852, Cyrillic)
855CYR	(Code Page 855, Cyrillic)
856HEB	(Code Page 856, Hebrew)
857TRK	(Code Page 857, Turkish)
860LATIN1	(Code Page 860, Latin 1)
861ISL	(Code Page 861, Icelandic)
862HEB	(Code Page 862, Hebrew)
863LATIN1	(Code Page 863, Latin 1)
865NOR	(Code Page 865, Norwegian)
866RUS	(Code Page 866, Russian)
869ELL	(Code Page 869, Greek)
SJIS	(Japanese Shift-JIS Encoding)
EUC_JAPAN	(Japanese EUC JIS X 0208-1990 and JIS X 0212-1990 Encoding)
EUC_CHINA	(Chinese GB 2312-80 Encoding)
EUC_TAIWAN	(Taiwanese Big 5 Encoding)
EUC_KOREA	(Korean KS C 5601-1992 Encoding)
UTF8	(UCS-4 Transformation Format)

List the available collating sequences

Running the DBINIT utility with the `-l` (L) option displays a list of available collating sequences and then stops. No database is created. The list of available collating sequences is automatically presented in SQL Central and in the ISQL Database Tools window.

37.12 The transaction log utility

With the transaction log utility you can display or change the name of the transaction log or transaction log mirror associated with a database. You can also stop a database from maintaining a transaction log or mirror, or start maintaining a transaction log or mirror.

A transaction log mirror is a duplicate copy of a transaction log, maintained by a database in tandem.

The name of the transaction log is first set when the database is initialized. The transaction log utility works with database files or with write files. The database engine must not be running on that database when the transaction log filename is changed (an error message is displayed if it is).

You can access the transaction log utility in the following ways:

- From SQL Central, for interactive use under Windows 95 or NT.
- From the ISQL Database Tools window, for interactive use under Windows 3.x or OS/2.
- From the ISQL DBTOOL statement, for interactive use under any supported operating system.
- From the system command line, using the DBLOG utility.

37.12.1 Changing a log file name from SQL Central

For full information on changing a log file name from SQL Central, see the SQL Central online help.

You can change a transaction log file name as follows:

1. Open the Database Utilities folder in the left panel.
2. Double-click Change Log File in the right panel. The Transaction Log Wizard is displayed
3. Follow the instructions in the Wizard.

37.12.2 Changing a log file name from the ISQL Tools window

To use the transaction log utility from the ISQL Tools window:

1. Select Database Tools from the Window menu.
2. Click Change Transaction Log Name on the Tools list.
3. Enter the database file name (with path).
4. Click Change, and select from the options on the dialog.
5. Click OK to change the name of the transaction log, or to run without a transaction log.

37.12.3 Changing a log file name from the DBTOOL statement

The syntax for the transaction log utility from the ISQL DBTOOL statement is as follows:

Syntax

```
DBTOOL ALTER DATABASE name

...     | NO [ TRANSACTION ] LOG                          |
        | SET [ TRANSACTION ] LOG TO filename             |
```

You can rename the database file name as well as the transaction log name from the DBTOOL statement.

37.12.4 The DBLOG command-line utility

Syntax

```
                DBLOG   [switches] database-file

Windows 3.x:    DBLOGW  [switches] database-file

switches:       -m mirror-name  Set transaction log mirror name
                -n              No longer use a transaction log
                -o              Output messages to file
                -q              Quiet mode—do not print messages
                -r              No longer use a transaction log mirror
                -t log-name     Set transaction log name
```

37.12.5 Transaction log utility options

Set the name of the transaction log mirror file

This option sets a file name for a new transaction log mirror. If the database is not currently using a transaction log mirror, it starts using one. If the database is already using a transaction log mirror, it changes to using the new file name as its transaction log mirror. For the DBLOG utility, this is the `-m` command-line switch.

No longer use a transaction log

Stop using a transaction log. Without a transaction log, the database will no longer be able to participate in data replication or use the transaction log in data recovery. For the DBLOG utility, this is the `-n` command-line switch.

Output log messages to file

Redirect log messages to the named file. For the DBLOG utility, this is the `-o` command-line switch.

Operate quietly

Do not display messages on a window. For the DBLOG utility, this is the `-q` command-line switch. This option is not available from other environments.

No longer use a transaction log mirror

For databases maintaining a mirrored transaction log, this option changes their behavior to maintain only a single transaction log. For the DBLOG utility, this is the `-r` command-line switch.

Set the name of the transaction log file

This option sets a file name for a new transaction log. If the database is not currently using a transaction log, it starts using one. If the database is already using a transaction log, it changes to using the new file name as its transaction log. For the DBLOG utility, this is the `-t` command-line switch.

37.13 The Open Server Gateway

The SQL Anywhere Open Server enables Sybase **dblib** and **ctlib** applications to connect to SQL Anywhere databases.

Syntax

```
DBOS50  [switches] [open-server-name]

switches:   -d database-name   Name of database for connections
            -e engine-name     Name of engine or client for connections
            -o log_file        Output messages to file
            -p                 Run in non-preemptive mode
            -v                 Verbose operation (extended messages)
            -w                 React to Warnings as Errors
```

See also

The chapter "Using the Open Server Gateway" on page 519, "The DBOSSTOP utility" on page 722, "The DBOSINFO utility" on the next page,.

Description

The SQL Anywhere Open Server enables Sybase **dblib** and **ctlib** applications to connect to SQL Anywhere databases.

For SQL Anywhere databases participating in a Replication Server installation, SQL Anywhere Open Server is required at primary and replicate sites to enable Replication Server to connect to the database. At a SQL Anywhere replicate site, Replication Server must connect in order to apply changes to the database. At a SQL Anywhere primary site, Replication Server must connect for the materialization step and if asynchronous procedure calls are being carried out.

The default Open Server name is SQLAny.

Switches

-d database-name
: Sets the database for connections. Connections to SQL Server are made to the server, while connections to SQL Anywhere are made to an individual database. The `-d` switch allows Open Server Gateway to support SQL Server connection events. If no `-d` switch is provided, the Open Server uses the first active database on the specified engine or client for all connections.

-e server-name
: Sets the SQL Anywhere server name for connections. Connections to SQL Server are made to the server, while connections to SQL Anywhere are made to an individual database. The `-d` switch allows Open Server Gateway to support SQL Server connection events. If no `-e` switch is provided, the Open Server looks for a running engine or client and uses the first engine or client it finds for all connections.

-o logfile
: Name the log file to which messages are output. The default name for the log file is the server name with extension .WOS.

-p
: Run in non-preemptive mode. In non-preemptive mode the Open Server processes commands serially rather than concurrently.

-v
: With the `-v` switch set, the messages sent to the log file are also sent to the current command-line window.

-w With the -w switch set, the Open Server reacts to warnings as if they were errors, stopping execution. This may be useful during development, especially if the application is to run against other Open Servers also.

37.14 The DBOSINFO utility

Syntax

DBOSINFO [switches]

switches:	-S server	Open Server to return information on
	-U userid	Connection username
	-P password	Connection password
	-v	Verbose operation

See also

"The Open Server Gateway" on page 719, "The DBOSSTOP utility" on the next page.

Description

The DBOSINFO utility returns information about a Open Server Gateway. Currently, the only information returned is the TDS version.

Switches

-S server The Open Server to return information on. The Open Server name is declared on the DBOS50 command line, and corresponds to an entry in the list of Open Servers maintained by SQLEDIT .

-U userid A SQL Anywhere user ID on a database being supported by the Open Server.

-P password The SQL Anywhere connection password corresponding to the user ID provided in the -U command line.

-v Extra messages are sent to the window.

37.15 The DBOSSTOP utility

Syntax

DBOSSTOP [switches]

switches:	-S server	Open Server to stop
	-U userid	Connection username
	-P password	Connection password
	-x password	Stopping password
	-q	Operate quietly
	-v	Verbose operation

See also

"The Open Server Gateway" on page 719, "The DBOSINFO utility" on the previous page.

Description

The DBOSSTOP utility stops a Open Server Gateway.

Switches

-S server — The Open Server to stop. The Open Server name is declared on the DBOS50 command line, and corresponds to an entry in the list of Open Servers maintained by SQLEDIT .

-U userid — A SQL Anywhere user ID on a database being supported by the Open Server.

-P password — The SQL Anywhere connection password corresponding to the user ID provided in the -U command line.

-x password — An additional password for DBOSSTOP. This is currently set permanently to `thx1138`.

-q — Operate quietly. No messages are sent to the log file.

-v — Extra messages are sent to the window.

37.16 The SQL Remote Message Agent

Syntax

DBREMOTE [switches] [directory]

Windows 3.x: DBREMOTW [switches] [directory]

switches:	-a	Do not apply received transactions
	-b	Run in batch mode
	-c	"keyword=value; ..." Supply database connection parameters
	-k	Close window on completion
	-o file	Output messages to file
	-p	Do not purge messages
	-r	Receive only
	-s	Send only
	-v	Verbose operation

See also

Description

The messaging agent sends and applies messages for SQL Remote replication, and maintains the message tracking system to ensure message delivery.

The user ID in the Message Agent command line must have either REMOTE DBA or DBA authority. For information, see "The Message Agent and replication security" on page 410.

The optional *directory* parameter specifies a directory in which old transaction logs are held, so that the Message Agent has access to events from before the current log was started. For information on log management in SQL Remote, see "Transaction log and backup management for SQL Remote" on page 413.

Switches

-a Process the received messages (those in the inbox) without applying them to the database. Used together with `-v` (for verbose output) and `-p` (so the messages are not purged), this flag can help detect problems with incoming messages. Used without `-p`, this flag purges the inbox without applying the messages, which may be useful if a subscription is being restarted.

-b Run in batch mode. In this mode, the Message Agent processes incoming messages, scans the transaction log once and processes outgoing messages, and then stops.

-c "keyword=value; ..."

Specify connection parameters. See "Database connection parameters" on page 128 for a description of the connection parameters. If this option is not specified, the environment variable SQLCONNECT is used.

For example, the following statement runs **dbremote** on a database file named **c:\sqlany50\sademo.db**, connecting with user ID **dba** and password **sql**:

```
dbremote -c
"uid=dba;pwd=sql;dbf=c:\sqlany50\sademo.db"
```

The DBREMOTE utility must be run by a user with REMOTE DBA authority or DBA authority. For information on REMOTE DBA authority, see "The Message Agent and replication security" on page 410.

-k Close window on completion. This flag is valid for Windows and Windows NT only.

-o Append output to a log file. Default is to send output to the screen.

-p Process the messages without purging them.

-r Receive messages only.

-s Send messages only.

-v Verbose output. This switch displays the SQL statements contained in the messages to the screen and, if the `-o` switch is used, to a log file.

37.17 The database compression utility

With the compression utility you can compress a database file. The compression utility reads the given database file and create a compressed database file. Compressed database files are useful if disk space is limited. Compressed databases are usually 40 to 60 percent of their original size. The database engine cannot update compressed database files: they must be used in conjunction with write files.

The compression utility does not compress files other than the main database file.

You can access the compression utility in the following ways:

- From SQL Central, for interactive use under Windows 95 or NT.
- From the ISQL Database Tools window, for interactive use under Windows 3.x or OS/2.
- From the ISQL DBTOOL statement, for interactive use under any supported operating system.
- From the system command line, using the DBSHRINK utility. This is useful for incorporating into batch or command files.

37.17.1 Compressing a database in SQL Central

For full information on compressing a database in SQL Central, see the SQL Central online help.

You can compress a database file as follows:

1. Open the Database Utilities folder in the left panel.
2. Double-click Compress Database in the right panel. The Compress Database Wizard is displayed.
3. Follow the instructions in the Wizard.

37.17.2 Compressing a database from the ISQL Tools window

To use the compression utility from the ISQL Tools window:

1. Select Database Tools from the Window menu.
2. Click Compress Database on the Tools list.
3. Enter the database file name (with path).
4. Click Compress, and enter a file name to use for the uncompressed database.
5. Click OK to compress the selected database file.

37.17.3 Compressing a database using the DBTOOL statement

The syntax for the compression utility from the ISQL DBTOOL statement is as follows:

Syntax

```
DBTOOL COMPRESS DATABASE filename
    ...    [ TO filename ]
```

37.17.4 The DBSHRINK command-line utility

Syntax

```
                DBSHRINK  [switches] database-file [compressed-database-file]
Windows 3.x:    DBSHRINW  [switches] database-file [compressed-database-file]

switches:       -q        Quiet mode—do not print messages
                -y        Erase existing output file without confirmation
```

DBSHRINK reads the given *database* file and create a compressed database file. The compressed filename defaults to the same name as the first with an extension of .cdb. The output filename (with extension) must not have the same name as the input filename (with extension).

37.17.5 Compression utility options

Operate quietly

Do not display messages on a window. For the DBSHRINK utility, this is the `-q` command-line switch. This option is not available from other environments.

Operate without confirming actions

Without this option, you are prompted to confirm the replacement of an existing database file. For the DBSHRINK utility, this is the `-y` command-line switch.

37.18 The Stop utility

The Stop utility stops a SQL Anywhere standalone engine, network server, or SQL Anywhere Client.

All memory used by the database engine is returned to the system for use by other applications.

The Stop utility is a command-line utility only. In Windowing environments, you can stop a database engine or server by clicking Close on the engine window or choosing Exit from the File menu on the server window.

37.18.1 The DBSTOP command-line utility

Syntax

```
                DBSTOP  [switches] { name | CLIENT }

Windows 3.x:    DBSTOPW  [switches] { name | CLIENT }

switches:       -c          Do not stop if there are active connections
                -q          Quiet mode—do not print messages
  QNX switches: -p pswd     Password to unlock server
```

The *name* specifies the engine or server name.

In DOS, it is not necessary to specify a *name*, because only one database engine or client can be running.

In QNX, DBSTOP can shut down a server on any node on the network. The *name* is necessary to specify the name of the server you wish to stop.

The DBSTOP command is not necessary after the database engine has been started by ISQL. ISQL automatically unloads the database engine when it disconnects.

DBSTOP CLIENT stops the SQL Anywhere Client only.

Switches

-c Do not stop the engine if there are still active connections to the engine.

-q Operate quietly. Do not print a message if the database was not running.

QNX switch

-p pswd Specify a password to unlock the server. If the server's keyboard is locked and no password is specified or the specified password is incorrect, then the server will not shut down.

37.19 The log translation utility

With the log translation utility you can translate a transaction log into a SQL command file.

You can access the log translation utility in the following ways:

- From SQL Central, for interactive use under Windows 95 or NT.
- From the ISQL Database Tools window, for interactive use under Windows 3.x or OS/2.
- From the ISQL DBTOOL statement, for interactive use under any supported operating system.
- From the system command line, using the DBTAN utility. This is useful for incorporating into batch or command files.

37.19.1 Translating a transaction log in SQL Central

You can translate a transaction log into a command file as follows:

1. Open the Database Utilities folder in the left panel.
2. Double-click Translate Log in the right panel. The Translate Log Wizard is displayed
3. Follow the instructions in the Wizard.

37.19.2 Translating a transaction log from the ISQL Tools window

To use the log translation utility from the ISQL Tools window:

1. Select Database Tools from the Window menu.
2. Click Translate Transaction Log to SQL on the Tools list.
3. Enter the transaction log file name (with path).

4. Click Translate, and select from the options in the dialog.
5. Click OK to translate the selected transaction log file to a SQL command file.

37.19.3 Translating a transaction log using the DBTOOL statement

The syntax for the log translation utility from the ISQL DBTOOL statement is as follows:

Syntax

```
DBTOOL  TRANSLATE [ TRANSACTION ] LOG FROM logname

...     [ TO sqlfile ] [ WITH ROLLBACKS ]

...     | [ USERS u1, u2, ... , un ]                                  |
        | [ EXCLUDE USERS u1, u2, ... , un ]                          |

...     [ LAST CHECKPOINT ] [ ANSI ] [ NOCONFIRM ]
```

37.19.4 The DBTRAN command-line utility

Syntax

```
                DBTRAN   [switches] transaction-log [sql-log]

Windows 3.x:    DBTRANW  [switches] transaction-log [sql-log]
```

switches:	-a	Include rollback transactions in output
	-f	Output from most recent transaction
	-r	Remove uncommitted transactions (default)
	-s	Produce ANSI standard SQL UPDATE transactions
	-t	Include trigger-generated transactions in output
	-u userid,...	Translate transactions for listed users
	-x userid,...	Exclude transactions for listed users
	-y	Replace file without confirmation
	-z	Include trigger-generated transactions as comments

The *sql-log* filename defaults to the transaction log name with a .sql extension.

37.19.5 Log translation utility options

Include uncommitted transactions

The transaction log contains any changes made before the most recent COMMIT by ANY transaction. Changes made after the most recent commit are not present in the transaction log. For the DBTRAN utility, this is the `-a` command-line switch.

Translate from last checkpoint only

Only the transactions completed since the last checkpoint are translated. For the DBTRAN utility, this is the `-f` command-line switch.

Do not include uncommitted transactions

Remove any transactions that were not committed For the DBTRAN utility, this is the `-r` command-line switch, and the default operation.

Generate ANSI standard SQL UPDATE

For cases where there is no primary key or unique index on a table, the translation utility generates UPDATE statements with a non-standard FIRST keyword, in case of duplicate rows. The ANSI standard UPDATE flag does not output this keyword. For the DBTRAN utility, this is the `-s` command-line switch

Include transactions generated by triggers

By default, actions carried out by triggers are not included in the command file. If the matching trigger is in place in the database, then when the command file is run against the database, the trigger will carry out the actions automatically. Trigger actions should be included in the case that the matching trigger does not exist in the database against which the command file is to be run. For the DBTRAN utility, this is the `-t` command-line switch.

Translate transactions for listed users only

For the DBTRAN utility, this is the `-u` command-line switch.

Translate transactions except for listed users

For the DBTRAN utility, this is the `-x` command-line switch.

Operate without confirming actions.

Without this option, you are prompted to confirm the replacement of an existing command file. For the DBTRAN utility, this is the `-y` command-line switch.

Include transactions generated by triggers as comments only
Transactions generated by triggers will be included only as comments in the output file. For the DBTRAN utility, this is the `-z` command-line switch.

37.20 The Unload utility

With the Unload utility you can unload a database and put data files into a set of files in a named directory. The Unload utility creates the ISQL command file **reload.sql** to rebuild your database from scratch. It also unloads all of the data in each of your tables in comma delimited format into files in the specified directory. Binary data is properly represented with escape sequences.

You can access the Unload utility in the following ways:

- From SQL Central, for interactive use under Windows 95 or NT.
- From the ISQL Database Tools window, for interactive use under Windows 3.x or OS/2.
- From the ISQL DBTOOL statement, for interactive use under any supported operating system.
- From the system command line, using the DBUNLOAD utility. This is useful for incorporating into batch or command files.

The Unload utility should be run from a DBA user ID. This is the only way you can be sure of having privileges to unload all the data. Also, the **reload.sql** file should be run from the DBA user ID. (Usually it will be run on a new database where the only user ID is DBA with password SQL.)

37.20.1 Unloading a database from SQL Central

For full information on unloading a database from SQL Central, see the SQL Central online help.

You can unload a running database as follows:

1. Connect to the database.
2. Right-click the database and click Unload in the popup menu. The Unload Database Wizard is displayed.
3. Follow the instructions in the Wizard.

You can unload a database file or a running database as follows:

1. Open the Database Utilities folder in the left panel.

2. Double-click Unload Database in the right panel. The Unload Database Wizard is displayed.
3. Follow the instructions in the Wizard.

37.20.2 Unloading a database from the ISQL Tools window

To use the unload utility from the ISQL Tools window:

1. Select Database Tools from the Window menu.
2. Click Unload Database on the Tools list.
3. Enter a user ID and password to use when connecting to the database.
4. For a running database, enter a database name and server name (if more than one is running). For a database file, enter the file name (with path) and optionally a start line to specify command-line switches for the database engine or SQL Anywhere Client.
5. Click Unload, and enter a file name to use for the command file and a directory in which to hold the data. Select from the other options in the dialog.
6. Click OK to unload the database.

37.20.3 Unloading a database from the DBTOOL statement

The syntax for the unload utility from the ISQL DBTOOL statement is as follows:

Syntax

```
DBTOOL  UNLOAD TABLES TO directory

...     [ RELOAD FILE TO filename ]

...     | [ DATA ]                                                    |
        | [ SCHEMA ]                                                  |

...     [ UNORDERED ] [ VERBOSE ] USING connection-string
```

37.20.4 The DBUNLOAD command-line utility

Syntax

DBUNLOAD [switches] directory [table-name-list]

Windows 3.x: DBUNLOAW [switches] directory [table-name-list]

switches:

-c "keyword=value; ..."	Supply database connection parameters.
-d	Unload data only.
-e	No data output for listed tables
-g userid	Specify user name as replacement for DBO.
-i	Data output for listed tables only
-n	No data—schema definition only.
-q	Quiet mode—no windows or messages.
-r reload-file	Specify name of generated reload ISQL. command file (default RELOAD.SQL).
-u	Unordered data.
-v	Verbose messages.
-x	External unload (when client and server are on different machines).
-y	Replace command file without confirmation.

In the default mode the directory used by DBUNLOAD to hold the data is relative to the database engine or server, not to the current directory of the user. For details of how to supply a file name and path in this mode, see "UNLOAD TABLE Statement" on page 1025. If the `-x` switch is used, the directory is relative to the current directory of the user. The **reload.sql** command file is always relative to the current directory of the user, regardless of whether `-x` is used.

If no list of tables is supplied, the whole database is unloaded. If a list of tables is supplied, only those tables are unloaded.

37.20.5 Unload utility options

Connection parameters

For a description of the connection parameters, see "Database connection parameters" on page 128. If the connection parameters are not specified, connection paremeters from the SQLCONNECT environment variable are used, if set. The **user ID** should have DBA authority to ensure that the user has permissions on all the tables in the database.

For the DBUNLOAD utility, this is the `-c` command-line switch. For example, the following statement unloads the **sademo**

database running on the **sample_server** server, connecting as user ID DBA with password SQL. The data is unloaded into the **c:\unload** directory.

```
dbunload -c
"eng=sample_server;dbn=sademo;uid=dba;pwd=sql"
c:\unload
```

Unload data only

With this option, none of the database definition commands are generated (CREATE TABLE, CREATE INDEX, and so on); **reload.sql** contains statements to reload the data only. For the DBUNLOAD utility, this is the `-d` command-line switch.

No data output for listed tables

For the DBUNLOAD utility, this is the `-e` command-line switch. This is not accessible from other environments. By default, the optional table-list defines the tables to be unloaded. If you wish to unload almost all the tables in the database, the `-e` command-line switch unloads all tables except the specified tables.

User ID for DBO user

SQL Anywhere contains a set of system views that mimic the system tables of Sybase SQL Server. By default, the owner of these views is the user ID **DBO**, which is the same as the owner of the SQL Server system tables, but you may have configured this differently when creating or upgrading your database. If the SQL Server-compatibility system views in your database are owned by a user other than DBO, specify that user ID in this option. Views and procedures owned by the user ID you specify are not unloaded. For the DBUNLOAD utility, this is the `-g` command-line switch.

Data output for listed tables only

For the DBUNLOAD utility, this is the `-i` command-line switch. This is not accessible from other environments. This switch is the default option, and causes only the supplied table-list to be unloaded.

Unload schema definition only

With this option, none of the data in the database is unloaded; **reload.sql** contains SQL statements to build the structure of the database only. For the DBUNLOAD utility, this is the `-n` command-line switch.

Operate quietly

Display no messages except errors. For the DBUNLOAD utility, this is the `-q` command-line switch. This option is not available from other environments.

Specify reload file name

Modify the name and directory of the generated reload ISQL command file. The default is **reload.sql** in the current directory. For the DBUNLOAD utility, this is the `-r` command-line switch.

Output the data unordered

Normally the data in each table is ordered by the primary key. Use this option if you are unloading a database with a corrupt index so that the corrupt index is not used to order the data. For the DBUNLOAD utility, this is the `-u` command-line switch.

Enable verbose mode

The table name of the table currently being unloaded and how many rows have been unloaded is displayed. For the DBUNLOAD utility, this is the `-v` command-line switch. This option is not available from other environments.

Use external unloading

The unload utility uses the UNLOAD TABLE statement by default. If the external unloading optoin is specified, the ISQL OUTPUT statement is used instead. For the DBUNLOAD utility, this is the `-x` command-line switch. The default mode is faster than the external unloading mode. However, in the default mode the directory used by the unload utility to hold the data is relative to the database server, not to the current directory of the user. For details of how to supply a file name and path in this mode, see "UNLOAD TABLE Statement" on page 1025. The **reload.sql** command file is always relative to the current directory of the user, regardless of whether external unloading is used.

Operate without confirming actions

Without this option, you are prompted to confirm the replacement of an existing command file. For the DBUNLOAD utility, this is the `-y` command-line switch.

Rebuilding a database

To unload a database, start the database engine with your database, and run the unload utility with the DBA user ID and password.

To reload a database, you need to create a new database and then run the generated **reload.sql** command file through ISQL or RTSQL. In DOS, OS/2, Windows 95 and NT, and QNX, there is .do a file (**rebuild.bat**, **rebuild.cmd**, or **rebuild**) included with SQL Anywhere that automates the unload and reload process. In Windows 3.x, rebuilding can be done from the ISQL Database Tools window.

37.21 The Upgrade utility

The Upgrade utility upgrades a database created with Watcom SQL Version 3.2 or Watcom SQL Version 4.0 to the SQL Anywhere 5.0 format. While SQL Anywhere 5.0 does run against a database created with Watcom SQL Version 3.2 or Watcom SQL Version 4.0, some of the features introduced since the version that created the database are unavailable unless the database is upgraded.

For people switching from release 3.2 or 4.0 to the current release, the Upgrade utility upgrades their databases without having to unload and reload them.

If you wish to use replication on an upgraded database, you must also archive your transaction log and start a new one on the upgraded database.

You can access the Upgrade utility in the following ways:

- From SQL Central, for interactive use under Windows 95 or NT.
- From the system command-line, using the DBUPGRAD utility.

37.21.1 Upgrading a database from SQL Central

For full information on upgrading a database from SQL Central, see the SQL Central online help.

You can upgrade a database as follows:

1. Open the Database Utilities folder in the left panel.
2. Double-click Uprade Database in the right panel. The Upgrade Database Wizard is displayed.
3. Follow the instructions in the Wizard.

37.21.2 Upgrading databases too old for the Upgrade utility

If your database was created with a version of SQL Anywhere that is too old to be upgraded, you can upgrade it as follows:

1. Unload the database using the Unload utility.
2. Create a database with the name you wish to use for the upgraded version, using the database creation utility.
3. Connect to the new database from ISQL as the DBA user ID, and read the reaload.sql command file to build the new database.

37.21.3 The DBUPGRAD command-line utility

Syntax

DBUPGRAD [switches]

Windows 3.x: DBUPGRDW [switches]

switches:

-c "keyword=value; ..."	Supply database connection parameters
-g userid	User ID to use as replacement for DBO
-k	Close window on completion
-q	Quiet mode—no windows or messages

37.21.4 Upgrade utility options

Connection parameters

For a description of the connection parameters, see "Database connection parameters" on page 128. If the connection parameters are not specified, connection paremeters from the SQLCONNECT environment variable are used, if set. The **user ID** must have DBA authority.

For the DBUPGRAD utility, this is the `-c` command line switch. For example, the following command upgrades a database called sample40 to a 5.0 format, connecting as user *dba* with password *sql*:

```
dbupgrad -c "uid=dba;pwd=sql;dbf=c:\wsql40\sample40.db"
```

The DBUPGRAD utility must be run by a user with DBA authority.

User ID for DBO user

SQL Anywhere contains a set of system views that mimic the system tables of Sybase SQL Server. By default, the owner of these views is the user ID **DBO**, which is the same as the owner of the SQL Server system tables. If you already have a user ID named **DBO**, or you wish to use that user ID for other purposes, this option allows you to provide an alternative user ID for the owner of the SQL Server-like system views. For the DBUPGRAD utility, this is the `-g` command-line switch.

Close window on completion

Once the upgrade is completed, the message window is closed. For the DBUPGRAD utility, this is the `-k` command line switch. This option is not available from other environments.

Operate quietly

Do not display messages on a window For the DBUPGRAD utility, this is the `-q` command line switch. This option is not available from other environments.

37.22 The validation utility

With the validation utility you can validate all indexes and keys on some or all of the tables in the database. The validation utility scans the entire table, and looks up each record in every index and key defined on the table. See "VALIDATE TABLE Statement" on page 1032.

This utility can be used in combination with regular backups (see "Backup and Data Recovery" on page 331) to give you confidence in the security of the data in your database.

You can access the validation utility in the following ways:

- From SQL Central, for interactive use under Windows 95 or NT.
- From the ISQL Database Tools window, for interactive use under Windows 3.x or OS/2.
- From the ISQL DBTOOL statement, for interactive use under any supported operating system.
- From the system command line, using the DBVALID utility. This is useful for incorporating into batch or command files.

37.22.1 Validating a database from SQL Central

For full information on validating a database from SQL Central, see the SQL Central online help.

You can validate a running database as follows:

1. Connect to the database.
2. Right-click the database and click Validate in the popup menu.

You can validate an individual table as follows:

1. Connect to the database.
2. Locate the table you wish to validate.
3. Right-click the table and click Validate in the popup menu.

37.22.2 Validating a database from the ISQL Tools window

To use the validation utility from the ISQL Tools window:

1. Select Database Tools from the Window menu.
2. Click Check Database Integrity on the Tools list.
3. Enter a user ID and password to use when connecting to the database.
4. For a running database, enter a database name and server name (if more than one is running). For a database file, enter the file name (with path) and optionally a start line to specify command line switches for the database engine or SQL Anywhere Client.
5. Click Check.
6. Enter a list of tables to validate, if you do not wish to validate the entire database, and click OK.

37.22.3 Using the validation utility from the DBTOOL statement

The syntax for the validation utility from the ISQL DBTOOL statement is as follows:

Syntax

```
DBTOOL VALIDATE TABLES [ t1, t2, ..., tn ]
    ...    USING connection-string
```

37.22.4 Using the DBVALID command line utility

Syntax

DBVALID [switches] [table-name,. . .]

Windows 3.x: DBVALIDW [switches] [table-name,. . .]

switches: -c "keyword=value; ..."
Supply database connection parameters
-q Quiet mode—do not print messages

37.22.5 Validation utility options

Connection parameters

For a description of the connection parameters, see "Database connection parameters" on page 128. If the connection parameters are not specified, connection paremeters from the SQLCONNECT environment variable are used, if set. The **user ID** must have DBA authority.

For the DBVALID utility, this is the `-c` command line switch. For example, the following validates the sample database, connecting as user *dba* with password *sql*:

```
dbvalid -c "uid=dba;pwd=sql;dbf=c:\sademo\sademo.db"
```

Operate quietly

Do not display messages on a window. For the DBVALID utility, this is the `-q` command line switch. This option is not available from other environments.

37.23 The DBWATCH server monitoring facility

Syntax

DBWATCH [switches]

Windows 3.X: DBWATCHW [switches]

See also

DBCLIENT

Description

See the *SQL Anywhere Network Guide* for a description of the DBWATCH server monitoring facility .

37.24 The write file utility

The write file utility is used to manage database write files. A *write file* is a file attached to a particular database. All changes are written into the write file, leaving the database file unchanged. Write files can be used effectively for testing when you do not wish to modify the production database. They can also be used in network and student environments where read-only access to a database is desired, or when you distribute on CD-ROM a database you wish users to be able to modify. If you are using a compressed database, then you must use a write file; compressed database files cannot be modified directly. The write file name is then used in place of the database name when connecting to the database or when loading a database on the database engine or server command line.

You can access the write file utility in the following ways:

- From SQL Central, for interactive use under Windows 95 or NT.
- From the ISQL Database Tools window, for interactive use under Windows 3.x or OS/2.
- From the ISQL DBTOOL statement, for interactive use under any supported operating system.
- From the system command line, using the DBWRITE utility. This is useful for incorporating into batch or command files.

The write file utility runs against a database file. The database must not be running on an engine when you run the write file utility.

37.24.1 Creating a write file from SQL Central

For full information on creating a write file from SQL Central, see the SQL Central online help.

You can create a write file for a database as follows:

1. Open the Database Utilities folder in the left panel.
2. Double-click Create Write File in the right panel. The Write File Wizard is displayed.
3. Follow the instructions in the Wizard.

37.24.2 Creating a write file from the ISQL Tools window

To use the write file utility from the ISQL Tools window:

1. Select Database Tools from the Window menu.
2. Click Create Write File on the Tools list.
3. Enter the database file name (with path).
4. Click Create, and enter a file name to use for the write file and for the write file transaction log.
5. Click OK to create the write file.

37.24.3 Creating a write file using the DBTOOL statement

The syntax for the write file utility from the ISQL DBTOOL statement is as follows:

Syntax

```
DBTOOL CREATE WRITEFILE filename FOR DATABASE filename
   ...    [ [ TRANSACTION ] LOG TO logname ] [ NOCONFIRM ]
```

37.24.4 The DBWRITE command-line utility

Syntax

```
                DBWRITE   [switches] database-file [write-name]
Windows 3.x:    DBWRITEW  [switches] database-file [write-name]
```

switches:	-c	Create a new write file
	-d database-file	Point a write file to a different database
	-f database-file	Force write file to point at file
	-m mirror-name	Set transaction log mirror name
	-o file	Output messages to file
	-q	Quiet mode—do not print messages
	-s	Report write file status (default)
	-t log-name	Set transaction log name
	-y	Erase/replace old files without confirmation

If any changes are made to the original database (not using the write file), the write file will no longer be valid. This happens if you start the engine using the original database file and make a modification to it. It can be made valid again by the command:

```
dbwrite -c db-name write-name
```

However, this deletes all changes recorded in the write file.

The `log-name` and `mirror-name` parameters are only used when creating a new write file. The `write-name` parameter is only used with the `-c` and `-d` parameters. Note that the `db-name` parameter must be specified before the `write-name` parameter.

37.24.5 Write file utility options

Create a new write file.

If an existing write already exists, any information in the old write file will be lost. If no write filename is specified on the command line, the write filename will default to the database name with the extension .WRT. If no transaction log name is specified, the log filename will default to the database name with the extension .WLG. For the DBWRITE utility, this is the `-c` command-line switch.

Change the database file to which an existing write file points.

If a database file is moved to another directory, or renamed, this option allow you to maintain the link between the write file and the database file. For the DBWRITE utility, this is the `-d` command-line switch. The option is not available in other environments.

Force a write file to point to a file

For the DBWRITE utility, this is the `-f` command-line switch. The option is not available in other environments. This option is for use when a write file is being created and the database file is held on a Novell NetWare or other network path, for operating systems on which they cannot be entered directly. By providing the full Novell path name for the database file, (for example: **sys\\sademo.db**), dependencies on local mappings of the NetWare path can be avoided. Unlike the option to change the database file pointed to, no checking is done on the specified path.

Operate quietly

Do not display messages on a window. For the DBWRITE utility, this is the `-q` command-line switch. This option is not available from other environments.

Report write file status only

This displays the name of the database that the write file points to.

For the DBWRITE utility, this is the `-s` command-line switch. The option is not available in other environments.

Operate without confirming actions
Without this option, you are prompted to confirm the replacement of an existing database file. For the DBWRITE utility, this is the `-y` command-line switch.

37.25 The SQL Remote database extraction utility

You can access the remote database extraction utility in the following ways:

- From SQL Central, for interactive use under Windows 95 or NT.
- From the system command line, using the DBXTRACT utility. This is useful for incorporating into batch or command files.

By default, the extraction utility runs at isolation level zero. If you are extracting a database from a running server, you should run it at isolation level 3 (see "Extraction utility options" on the next page) to ensure that data in the extracted database is consistent with data on the server. Running at isolation level 3 may hamper others' turnaround time on the server because of the large number of locks required. It is recommended that you run the extraction utility when the server is not busy.

37.25.1 Extracting a remote database in SQL Central

For full information on extracting a remote database in SQL Central, see the SQL Central online help.

You can extract a remote database from a running database as follows:

1. Connect to the database.
2. Right-click the database and click Extract Database in the popup menu.
3. Follow the instructions in the wizard.

You can extract a remote database from a database file or a running database as follows:

1. Open the Database Utilities folder in the left panel.
2. Double-click Extract a Database in the right panel.
3. Follow the instructions in the wizard.

37.25.2 The DBXTRACT command-line utility

Syntax

DBXTRACT [switches] directory subscriber

Windows 3.x DBXTRACW [switches] directory subscriber

switches:

-b	Do not start subscriptions
-c "keyword=value; ..."	supply database connection parameters
-d	Unload data only
-g <user>	Specify user name as replacement for DBO
-I <level>	Perform all extraction operations at specified isolation level
-k	Close window on completion
-n	Extract schema definition only
-o <file>	Output messages to file
-q	Operate quietly: do not print messages or show windows
-r <file>	Specify name of generated reload ISQL command file (default "reload.sql")
-u	Unordered data
-v	Verbose messages
-xf	Exclude foreign keys
-xp	Exclude stored procedures
-xt	Exclude triggers
-xv	Exclude views
-y	Overwrite command file without confirmation

Description

Running the extraction utility from SQL Central carries out the following tasks related to creating and synchronizing SQL Remote subscriptions:

- Creates a command file to build a remote database containing a copy of the data in a specified publication.
- Creates the necessary SQL Remote objects, such as message types, publisher and remote user IDs, publication and subscription, for the remote database to receive messages from and send messages to the consolidated database.
- Starts the subscription at both the consolidated and remote databases.

37.25.3 Extraction utility options

Do not start subscriptions automatically

If this option is selected, subscriptions at the consolidated database (for the remote database) and at the remote database (for the consolidated database) must be started explicitly using the START SUBSCRIPTION statement for replication to begin. For the DBXTRACT command-line utility, this is the `-b` switch.

Connection parameters

For a description of the connection parameters, see "Database connection parameters" on page 128. If the connection parameters are not specified, connection paremeters from the SQLCONNECT environment variable are used, if set. The **user ID** should have DBA authority to ensure that the user has permissions on all the tables in the database.

For the DBXTRACT utility, this is the `-c` command-line switch. For example, the following statement extracts a database for remote user ID **joe_remote** from the **sademo** database running on the **sample_server** server, connecting as user ID DBA with password SQL. The data is unloaded into the **c:\unload** directory.

```
dbxtract -c "eng=sample_server;dbn=sademo;
    uid=dba;pwd=sql" c:\extract joe_remote
```

Unload the data only

If this option is selected, the schema definition is not unloaded, and publications and subscriptions are not created at the remote database. This option is for use when a remote database already exists with the proper schema, and needs only to be filled with data. For the DBXTRACT command-line utility, this is the `-d` switch.

Perform extraction at a specified isolation level

The default setting is an isolation level of zero. If you are extracting a database from a running server, you should run it at isolation level 3 (see "Extraction utility options" on the previous page) to ensure that data in the extracted database is consistent with data on the server. Increasing the isolation level may result in large numbers of locks being used by the extraction utility, and may restrict database use by other users. For the DBXTRACT command-line utility, this is the `-l` switch.

Unload the schema definition only

With this definition, none of the data is unloaded. The reload file contains SQL statements to build the database structure only. You can use the SYNCHRONIZE SUBSCRIPTION statement to load the data over the messaging system. Publications, subscriptions, PUBLISH and SUBSCRIBE permissions are part of the schema. For the DBXTRACT command-line utility, this is the `-n` switch.

Output messages to file

For the DBXTRACT command-line utility, this is the `-o` switch.

Operate quietly

Display no messages except errors. For the DBXTRACT utility, this is the `-q` command-line switch. This option is not available from other environments.

Reload file name

The default name for the reload command file is **reload.sql** in the current directory. For the DBXTRACT command-line utility, this is the `-r` switch.

Output the data unordered

By default the data in each table is ordered by primary key. Unloads are quicker with the `-u` switch, but loading the data into the remote database is slower. For the DBXTRACT command-line utility, this is the `-u` switch.

Verbose mode

The name of the table being unloaded and the number of rows unloaded are displayed. For the DBXTRACT command-line utility, this is the `-v` switch.

Exclude foreign key definitions

You can use this if the remote database contains a subset of the consolidated database schema, and some foreign key references are not present in the remote database. For the DBXTRACT command-line utility, this is the `-xf` switch.

Exclude stored procedure

Do not extract stored procedures from the database. For the DBXTRACT command-line utility, this is the `-xp` switch.

Exclude triggers

Do not extract triggers from the database. For the DBXTRACT command-line utility, this is the `-xt` switch.

Exclude views

Do not extract views from the database. For the DBXTRACT command-line utility, this is the `-xv` switch.

Operate without confirming actions

Without this option, you are prompted to confirm the replacement of an existing command file. For the DBXTRACT command-line utility, this is the `-y` switch.

37.26 The REBUILD batch or command file

Syntax

REBUILD old-database new-database [dba-password]

See also

DBINIT, DBUNLOAD, ISQL

Description

This DOS or Windows NT batch file, OS/2 command file, or QNX script uses DBUNLOAD to rebuild *old-database* into *new-database*. This is a simple script, but it helps document and automate the rebuilding process. Both database names should be specified without extensions. An extension of `.db` will automatically be added.

The *dba-password* must be specified if the password to the DBA userid in the *old-database* is not the initial password SQL.

REBUILD runs the DBUNLOAD, DBINIT and ISQL commands with the default command line options. If you need different options, you will need to run the three steps separately.

37.27 The SQL Preprocessor

The SQL preprocessor processes a C or C++ program with embedded SQL before the compiler is run.

Syntax

SQLPP [switches] sql-filename [output-filename]

switches:	-c	Favor code size
	-d	Favor data size
	-f	Put far keyword on generated static data
	-l userid,pswd	Logon identification
	-n	Line numbers
	-o operating-sys	Target operating system specification DOS, DOS32, DOS286, WINDOWS, WIN32 OS232, WINNT, NETWARE, QNX32 (default is DOS, OS232, or WINNT)
	-r	generate reentrant code
	-q	Quiet mode—do not print banner
	-s string-len	Maximum string constant length for compiler

See also

"The C language SQL preprocessor" on page 534

Description

The SQL preprocessor processes a C or C++ program with embedded SQL before the compiler is run. SQLPP translates the SQL statements in the *input-file* into C language source that is put into the *output-file*. The normal extension for source programs with embedded SQL is `.sqc`. The default output filename is the **sql-filename** with an extension of `.c`. If the **sql-filename** has a `.c` extension, then the output filename extension will be `.cc` by default.

Switches

-c Generate code that will favor code size and execution speed over data space size. Statically initialized data structures will be used as much as possible. This is the default.

-d Generate code that will reduce data space size. Data structures will be reused and initialized at execution time before use. This increases code size.

-f Put the *far* keyword in front of preprocessor generated data. This may be required in conjunction with the Borland C++ compiler for the large memory model. By default, all static data is put in the same segment. Adding the *far* keyword will force static data into different segments. (By default, WATCOM C and Microsoft C place data objects bigger than a threshold size in their own segment.)

-l userid,password
The named *userid* and *password* will be used for authorization of static SQL statements (see "Authorization" on page 542).

-n Generate line number information in the C file. This consists of *#line* directives in the appropriate places in the generated C code. If the compiler you are using supports the *#line* directive, this switch will make the compiler report errors on line numbers in the `.sqc` file (the one with the Embedded SQL) as opposed to reporting errors on line numbers in the `.c` file generated by the SQL preprocessor. Also, the *#line* directives will indirectly be used by the source level debugger so that you can debug while viewing the `.sqc` source file.

-o operating-sys Specify the target operating system. Note that this option must match the operating system you will be running the program

in. A reference to a special symbol will be generated in your program. This symbol is defined in the interface library. If you use the wrong operating system specification or the wrong library, an error will be detected by the linker. The supported operating systems are:

DOS	MS-DOS or DR-DOS
DOS32	32-bit DOS extended program
DOS286	286 DOS extended program
WINDOWS	Microsoft Windows 3.x
WIN32	32-bit Microsoft Windows
OS232	32-bit OS/2
WINNT	Microsoft Windows NT
NETWARE	Novell NetWare
QNX	16-bit QNX
QNX32	32-bit QNX

The default is DOS for the DOS version of SQLPP, OS232 for the OS/2 version, WINNT for the Windows NT version and QNX for the QNX version.

-r Generate reentrant code. This statement is only necessary when you are writing code that is reentrant (see "Multi-Threaded or Reentrant Code" on page 575).

-q Operate quietly. Do not print the banner.

-s string-len Set the maximum size string that the preprocessor will put into the C file. Strings longer than this value will be initialized using a list of characters (`'a','b','c'`, etc). Most C compilers have a limit on the size of string literal they can handle. This option is used to set that upper limit. The default value is 500.

Chapter 38

Watcom-SQL Language Reference

About this chapter

This chapter presents detailed descriptions of all SQL, ISQL and Embedded SQL commands, as well as descriptions of the elements that make up those commands.

The description of each command begins with a summary of the syntax, followed by the purpose, usage, authorization, side effects, other related commands and a detailed description.

Contents

38.1 Syntax conventions

The following conventions are used in the SQL syntax descriptions:

- All keywords are shown in upper case. This is often the way SQL statements are typed. However, SQL Anywhere allows all keywords to be in mixed case. Thus, SELECT is the same as Select which is the same as select.
- Items that the user must replace with appropriate identifiers or expressions are shown in lowercase.
- Options are listed vertically enclosed in vertical bars. In these cases, any one of the items in the vertical list is allowed.
- Lines beginning with . . . are a continuation of the commands from the previous line.
- Lists are shown with a list element followed by ",. . .". This means that one or more list elements are allowed and if more than one is specified, they must be separated by commas.
- Optional portions of a command are enclosed by square brackets. For example,

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

  indicates that the **savepoint-name** is optional. Alternative optional parts of a command are sometimes listed within the brackets separated by vertical bars. For example,

  ```
  [ ASC | DESC ]
  ```

  indicates that **ASC** or **DESC** are optional.
- When one of the options must be chosen, the alternatives are enclosed in curly braces. For example

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

  indicates that if the QUOTES option is chosen, one of ON or OFF must be provided. The braces should not be typed.

38.2 Watcom-SQL language elements

The following elements are found in the syntax of many SQL statements. Each of these elements is discussed in more detail later in this chapter.

column-name An *identifier* representing the name of a column.

condition An expression that evaluates to TRUE, FALSE, or UNKNOWN. See "Search conditions" on page 803.

connection-name
An *identifier* or a *string* representing the name of an active connection.

creator An *identifier* representing a user ID.

data-type A storage data type as described in "Data types" on page 755.

expression An expression, as described in "Expressions" on page 795.

filename A *string* containing a filename.

host-variable A C language variable declared as a host variable preceded by a colon.

identifier Any string of the characters A through Z, a through z, 0 through 9, underscore (_), at sign (@), number sign (#), or dollar sign ($). The first character must be a letter. Alternatively, any string of characters can be used as an identifier by enclosing it in quotation marks ("double quotes"). A quotation mark inside the identifier is represented by two quotation marks in a row. Identifiers are truncated to 128 characters. The following are all valid identifiers.

```
Surname
"Surname"
SomeBigName
some_big_name
"Client Number"
"With a quotation "" mark"
```

indicator-variable
A second host variable of type **short int** immediately following a normal host variable. It must also be preceded by a colon. Indicator variables are used to pass NULL values to and from the database. See "Indicator variables" on page 548.

number Any sequence of digits followed by an optional decimal part and preceded by an optional negative sign. Optionally, the number can be followed by an E and then an exponent. For example,

```
42
-4.038
.001
3.4e10
1e-10
```

role-name An *identifier* representing the role name of a foreign key.

search-condition
A condition that evaluates to TRUE, FALSE, or UNKNOWN. See "Search conditions" on page 803.

string Any sequence of characters enclosed in apostrophes ('single quotes'). An apostrophe is represented inside the string by two apostrophes in a row. A new line character is represented by a backslash followed by an n (\n). Hexadecimal escape sequences can be used for any character, printable or not. A hexadecimal escape sequence is a backslash followed by an x followed by two hexadecimal digits (for example, \x6d represents the letter m). A backslash character is represented by two backslashes in a row (\\). The following are valid strings:

```
'This is a string.'
'John''s database'
'\x00\x01\x02\x03'
```

savepoint-name
An *identifier* representing the name of a savepoint.

statement-label
An *identifier* representing the label of a loop or compound statement.

table-list A list of table names which may include correlation names and join operators. See "FROM Clause" on page 915.

table-name An *identifier* representing the name of a table.

userid An *identifier* representing a user name.

variable An *identifier* representing a variable name.

38.3 Data types

This section describes the data types supported by SQL Anywhere. The SQL Anywhere data types fall into the following categories:

Character data types For storing strings of letters, numbers and symbols.

Numeric data types For storing numerical data.

Date and time data types For storing dates and times.

Binary data types For storing binary data, including images and other information that is not interpreted by the database.

User-defined data types Aliases for built-in data types, including precision and scale values where applicable, and optionally including DEFAULT values and CHECK conditions.

Data type conversions are described in the section "Data type conversions" on page 764.

Some of the data types listed have Transact-SQL equivalents. These equivalents are not listed here. For a description of Transact-SQL data types supported by SQL Anywhere, see the chapter "Using Transact-SQL with SQL Anywhere".

38.3.1 Character data types

Syntax

Character data types

```
| CHAR    [ ( max-length ) ]                          |
| CHARACTER [ ( max-length ) ]                        |
| CHARACTER VARYING [ ( max-length ) ]                |
| LONG VARCHAR                                        |
| VARCHAR  [ ( max-length ) ]                         |
```

Description

CHAR [(size)]

Character data of maximum length **size**. If **size** is omitted, the default is 1. The maximum size allowed is 32,767. See the notes below on character data representation in the database, and on storage of long strings.

CHARACTER [(size)]
Same as CHAR[(size)].

VARCHAR [(size)]
Same as CHAR[(size)].

CHARACTER VARYING[(size)]
Same as CHAR[(size)].

LONG VARCHAR
Arbitrary length character data. The maximum size is limited by the maximum size of the database file (currently 2 gigabytes).

NOTE: In Watcom SQL Version 3.1, the maximum size for a LONG VARCHAR field was 32,767 characters.

Notes

Character data is placed in the database using the exact binary representation that is passed from the application. This usually means that character data is stored in the database with the binary representation of the current *code page*. The code page is the character set representation used by IBM-compatible personal computers. You can find documentation about code pages in the documentation for your operating system.

All code pages are the same for the first 128 characters. If you use special characters from the top half of the code page (accented international language characters), you must be careful with your databases. In particular, if you copy the database to a different machine using a different code page, those special characters will be retrieved from the database using the original code page representation. With the new code page, they will appear on the screen to be the wrong characters.

This problem also appears if you have two clients using the same multi-user server, but running with different code pages. Data inserted or updated by one client may appear incorrect to another.

This problem is quite complex. If any of your applications use the extended characters in the upper half of the code page, make sure that all clients and all machines using the database use the same or a compatible code page.

Long strings

SQL Anywhere treats CHAR, VARCHAR, and LONG VARCHAR columns all as the same type. Values up to 254 characters are stored as short strings, which are stored with a preceding length byte. Any values that are longer than 255

bytes are considered long strings. Characters after the 255th are stored separate from the row containing the long string value.

There are several functions (see "Functions" on page 765) that will ignore the part of any string past the 255th character. They are **soundex**, **similar**, and all of the date functions. Also, any arithmetic involving the conversion of a long string to a number will work on only the first 255 characters. It would be extremely unusual to run in to one of these limitations.

All other functions and all other operators will work with the full length of long strings.

38.3.2 Numeric data types

Syntax

Numeric data types

```
| DECIMAL [ ( precision [,scale] ) ]                |
| DOUBLE                                            |
| FLOAT                                             |
| INT                                               |
| INTEGER                                           |
| NUMERIC [ ( precision [,scale] ) ]                |
| REAL                                              |
| SMALLINT                                          |
```

Description

INT Signed integer of maximum value 2,147,483,647 requiring 4 bytes of storage.

INTEGER Same as INT.

SMALLINT Signed integer of maximum value 32,767 requiring 2 bytes of storage.

DECIMAL [(precision[,scale])]
A decimal number with **precision** total digits and with **scale** of the digits after the decimal point. The defaults are **scale = 6** and **precision = 30**. These defaults can be changed with the SET command (see "SET OPTION Statement" on page 989).

NOTE: The defaults for databases created with Watcom SQL Version 3.0 were 15 and 2 for precision and scale respectively.

The storage required for a decimal number can be computed as:

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

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

NUMERIC [(precision[,scale])]
Same as DECIMAL.

FLOAT Same as REAL.

DOUBLE A double precision floating-point number stored in 8 bytes. The range of values is 2.22507385850720160e-308 to 1.79769313486231560e+308. Values held as DOUBLE are accurate to 15 significant digits, but may be subject to round-off error beyond the fifteenth digit.

REAL A single precision floating-point number stored in 4 bytes. The range of values is 1.175494351e-38 to 3.402823466e+38. Values held as REAL are accurate to 6 significant digits, but may be subject to round-off error beyond the sixth digit.

Notes

The INTEGER, NUMERIC and DECIMAL data types are sometimes called **exact** numeric data types, in contrast to the **approximate** numeric data types FLOAT, DOUBLE, and REAL. The exact numeric data types are those for which precision and scale values can be specified, while approximate numeric data types are stored in a pre-defined manner. Columns holding exact numeric data are accurate after arithmetic operations to the least significant digit specified.

38.3.3 Date and time data types

Syntax

Date and time data types

```
| DATE                                                    |
| TIME                                                    |
| TIMESTAMP                                               |
```

Description

DATE A calendar date, such as a year, month and day. The year can be from the year 0001 to 9999. For historical reasons, a DATE column can also contain an hour and minute, but the TIMESTAMP data type is now recommended for anything with hours and minutes. A DATE value requires 4 bytes of storage.

NOTE: In Watcom SQL Version 3.1, dates before the year 1600 could not be stored.

TIMESTAMP A point in time, containing year, month, day, hour, minute, second and fraction of a second. The fraction is stored to 6 decimal places. A TIMESTAMP value requires 8 bytes of storage.

TIME A time of day, containing hour, minute, second and fraction of a second. The fraction is stored to 6 decimal places. A TIME value requires 8 bytes of storage. (ODBC standards restrict TIME data type to an accuracy of seconds. For this reason you should use Time data types in WHERE clause comparisons that rely on a higher accuracy than seconds.)

Sending dates and times to the database

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

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

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

```
21:35           -- 24 hour clock if no am or pm specified
10:00pm         -- pm specified, so interpreted as 12 hour clock
10:00           -- 10:00am in the absence of pm
10:23:32.234    -- seconds and fractions of a second included
```

When a date is sent to the database as a string, conversion to a date is automatic. The string can be supplied in one of two ways:

- As a string of format yyyy/mm/dd or yyyy-mm-dd, which is interpreted unambiguously by the database.
- As a string interpreted according to the DATE_ORDER database option.

Dates in the format yyyy/mm/dd or yyyy-mm-dd are always recognized unambiguously as dates regardless of the DATE_ORDER setting. Other characters can be used as separators instead of / or - for example, ?, a space character, or ,. You should use this format in any context where different users may be employing different DATE_ORDER settings. For example, in stored procedures, use of the unambiguous date format prevents misinterpretation of dates according to the user's DATE_ORDER setting. In other contexts, a more flexible date format can be used. SQL Anywhere can interpret a wide range of strings as formats. The interpretation depends on the setting of the database option DATE_ORDER. The DATE_ORDER database option can have the value 'MDY', 'YMD', or 'DMY' (see "SET OPTION Statement" on page 989). For example, the following statement sets the DATE_ORDER option to 'DMY':

```
SET OPTION DATE_ORDER    = 'DMY' ;
```

The default DATE_ORDER setting is 'YMD'. The SQL Anywhere ODBC driver sets the DATE_ORDER option to 'YMD' whenever a connection is made. The value can still be changed using the SET OPTION statement.

The database option DATE_ORDER determines whether the string '10/11/12' is interpreted by the database as Oct 11 1912, Nov 12 1910, or Nov 10 1912. The year, month, and day of a date string should be separated by some character (eg. /, -, or space) and appear in the order specified by the DATE_ORDER option. The year can be supplied as either 2 or 4 digits, with 2 digit years defaulting to the 20th century. The month can be the name or number of the month. The hours and minutes are separated by a colon, but can appear anywhere in the string.

With an appropriate setting of DATE_ORDER, the following strings are all valid dates:

```
92-05-23 21:35
92/5/23
1992/05/23
May 23 1992
23-May-1992
Tuesday May 23, 1992  10:00pm
```

If a string contains only a partial date specification, default values are used to fill out the date. The following defaults are used:

year This year.
month No default.
day 1 (useful for month fields; for example, 'May 1992' will be the date '1992-05-01 00:00')
hour, minute, second, fraction 0

Retrieving dates and times from the database

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

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

When a date or time is retrieved as a string, it is retrieved in the format specified by the database options DATE_FORMAT, TIME_FORMAT and TIMESTAMP_FORMAT. For descriptions of these options, see "SET OPTION Statement" on page 989.

For information on functions dealing with dates and times, see "Date and time functions" on page 774. The following arithmetic operators are allowed on dates:

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

timestamp - integer
: Subtract the specified number of days from a date or timestamp.

date - date
: Compute the number of days between two dates or timestamps.

date + time
: Create a timestamp combining the given date and time.

Date and time comparisons

The DATE data type also contains a time. (Watcom SQL Version 3.0 did not support the TIMESTAMP data type, so the DATE was used for both dates and dates with times.) If the time is not specified when a date is entered into the database, the time defaults to 0:00 or 12:00am (midnight). Any date comparisons always involve the times as well. A database date value of '1992-05-23 10:00' will not be equal to the constant '1992-05-23'. The DATEFORMAT function or one of the other date functions can be used to compare parts of a date and time field. For example:

```
DATEFORMAT(invoice_date,'yyyy/mm/dd') = '1992/05/23'
```

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

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

38.3.4 Binary data types

Syntax

Binary data types

```
| BINARY  [ ( max-length ) ]                    |
| LONG BINARY                                   |
```

Description

BINARY [(size)]

Binary data of maximum length **size** (in bytes). If **size** is omitted, the default is 1. The maximum size allowed is 32,767. The BINARY data type is identical to the CHAR data type except when used in comparisons. BINARY values will be compared exactly while CHAR values are compared without respect to upper/lower case (depending on the case-sensitivity of the database) or accented characters.

LONG BINARY

Arbitrary length binary data. The maximum size is limited by the maximum size of the database file (currently 2 gigabytes).

NOTE: In Watcom SQL Version 3.1, the maximum size for a LONG BINARY field was 32,767 bytes.

38.3.5 User-defined data types

User-defined data types allow columns throughout a database to be automatically defined on the same data type, with the same NULL or NOT NULL condition, with the same DEFAULT setting, and with the same CHECK condition. This encourages consistency throughout the database.

Simple user-defined data types

User-defined data types are created using the CREATE DATATYPE statement For full description of the syntax, see "CREATE DATATYPE Statement" on page 843.

The following statement creates a data type named **address**, which is a 35-character string.

```
CREATE DATATYPE street_address CHAR( 35 )
```

Resource authority is required to create data types. Once a data type is created, the user ID that executed the CREATE DATATYPE statement is the owner of that data type. Any user can use the data type, and unlike other database objects, the owner name is never used to prefix the data type name.

The **address** data type may be used in exactly the same way as any other data type when defining columns. For example, the following table with two columns has the second column as an **address** column:

```
CREATE TABLE twocol (
    id INT,
    street street_address
    )
```

User-defined data types can be dropped by their owner or by the DBA using the DROP DATATYPE statement:

```
DROP DATATYPE address
```

This statement can be carried out only if the data type is not used in any table in the database.

Constraints and defaults with user-defined data types

Many of the attributes associated with columns, such as allowing NULL values, having a DEFAULT value, and so on, can be built into a user-defined data type. Any column that is defined on the data type automatically inherits the NULL setting, CHECK condition, and DEFAULT values. This allows uniformity to be built into columns with a similar meaning throughout a database.

For example, many primary key columns in the sample database are integer columns holding ID numbers. The following statement creates a data type that may be useful for such columns:

```
CREATE DATATYPE id INT
NOT NULL
DEFAULT AUTOINCREMENT
CHECK( @col > 0 )
```

Any column created using the data type **id** is not allowed to hold NULLs, defaults to an autoincremented value, and must hold a positive number. Any identifier could be used instead of `col` in the `@col` variable.

The attributes of the data type can be overriden if needed by explicitly providing attributes for the column. A column created on data type **id** with NULL values explicitly allowed does allow NULLs, regardless of the setting in the **id** data type.

38.3.6 Data type conversions

Type conversions happen automatically, or they can be explicitly requested using the CAST or CONVERT function.

If a string is used in a numeric expression or as an argument to a function expecting a numeric argument, the string is converted to a number before use.

If a number is used in a string expression or as a string function argument, then the number is converted to a string before use.

All date constants are specified as strings. The string is automatically converted to a date before use.

There are certain cases where the automatic database conversions are not appropriate.

```
'12/31/90' + 5        -- SQL Anywhere tries to convert the string
                         to a number

'a' > 0               -- SQL Anywhere tries to convert 'a' to a
                         number
```

The CAST or CONVERT functions can be used to force type conversions. For information about the CAST and CONVERT functions, see "Data type conversion functions" on page 780.

The following functions can also be used to force type conversions (see "Functions" on the next page).

DATE(value) — Converts the expression into a date, and removes any hours, minutes or seconds. Conversion errors may be reported.

STRING(value) — Similar to CAST(value AS CHAR), except that string(NULL) is the empty string (''), while CAST(NULL AS CHAR) is the NULL value.

VALUE+0.0 — Equivalent to CAST(value AS DECIMAL).

38.4 Functions

This section describes the built-in functions supported by SQL Anywhere. Functions are used to return information from the database. They are allowed anywhere an expression is allowed.

SQL Anywhere's built-in functions fall into the following categories:

Aggregate functions.
Summarize data over a group of rows from the database.

Numeric functions.
Act on numerical data types or return numeric information.

String functions.
Act on strings or return information about strings.

Date and time functions.
Act on date and time data types or return date and time information.

Data type conversion functions.
Convert values from one to another.

System functions.
Report system information.

Miscellaneous functions.
Functions not in one of the above groups.

NULL value
Unless otherwise stated, any function that receives the NULL value as a parameter returns the NULL value.

Some of the functions listed have Transact-SQL equivalents that are not listed here. For a description of Transact-SQL functions supported by SQL Anywhere, see the chapter "Using Transact-SQL with SQL Anywhere".

38.4.1 Aggregate functions

Syntax

```
Aggregate function:

        | AVG ( aggregate-parm )                    |
        | COUNT ( * )                               |
        | COUNT ( aggregate-parm )                  |
        | LIST ( aggregate-parm )                   |
        | MAX ( aggregate-parm )                    |
        | MIN ( aggregate-parm )                    |
        | SUM ( aggregate-parm )                    |

aggregate-parm:
        | DISTINCT column-name                      |
        | expression                                |
```

Aggregate functions summarize data over a group of rows from the database. The groups are formed using the GROUP BY clause of the SELECT statement. Aggregate functions are only allowed in the select list and in the HAVING and ORDER BY clauses of a SELECT statement.

*COUNT(*)* Returns the number of rows in each group.

COUNT(expression)
Returns the number of rows in each group where the *expression* is not the NULL value.

COUNT(DISTINCT column-name)
Returns the number of different values in the column with name *column-name*. Rows where the value is the NULL value are not included in the count.

LIST(string-expr)
Returns a string containing a comma-separated list composed of all the values for *string-expr* in each group of rows. Rows where *string-expr* is the NULL value are not added to the list.

LIST(DISTINCT column-name)
Returns a string containing a comma-separated list composed of all the different values for *string-expr* in each group of rows. Rows where *string-expr* is the NULL value are not added to the list.

AVG(numeric-expr)
Computes the average of *numeric-expr* for each group of rows. This average does not include rows where the *expression* is the

NULL value. Returns the NULL value for a group containing no rows.

AVG(DISTINCT column-name)

Computes the average of the unique values for *numeric-expr* for each group of rows. This is of limited usefulness, but is included for completeness.

MAX(expression)

Returns the maximum *expression* value found in each group of rows. Rows where expression is the NULL value are ignored. Returns the NULL value for a group containing no rows.

MAX(DISTINCT column-name)

Returns the same as MAX(expression), and is included for completeness.

MIN(expression)

Returns the minimum *expression* value found in each group of rows. Rows where expression is the NULL value are ignored. Returns the NULL value for a group containing no rows.

MIN(DISTINCT column-name)

Returns the same as MIN(expression), and is included for completeness.

SUM(expression)

Returns the total of *expression* for each group of rows. Rows where the *expression* is the NULL value are not included. Returns NULL for a group containing no rows.

SUM(DISTINCT column-name)

Computes the sum of the unique values for *numeric-expr* for each group of rows. This is of limited usefulness, but is included for completeness.

38.4.2 Numeric functions

Syntax

Numeric function:

```
| ABS ( numeric-expr )                                    |
| ACOS ( numeric-expr )                                   |
| ASIN ( numeric-expr )                                   |
| ATAN ( numeric-expr )                                   |
| ATAN2 ( numeric-expr, numeric-expr )                    |
| CEILING ( numeric-expr )                                |
| COS ( numeric-expr )                                    |
| COT ( numeric-expr )                                    |
| DEGREES (numeric-expr )                                 |
| EXP ( numeric-expr )                                    |
| FLOOR ( numeric-expr )                                  |
| LOG ( numeric-expr )                                    |
| LOG10 ( numeric-expr )                                  |
| PI ( * )                                                |
| POWER ( numeric-expr, numeric-expr )                    |
| RADIANS ( numeric-expr )                                |
| RAND ( [integer-expr] )                                 |
| REMAINDER ( numeric-expr, numeric-expr )                |
| ROUND ( numeric-expr, integer-expr )                    |
| SIGN ( numeric-expr )                                   |
| SIN ( numeric-expr )                                    |
| SQRT ( numeric-expr )                                   |
| TAN ( numeric-expr )                                    |
| TRUNCATE ( numeric-expr, integer-expr )                 |
```

ABS(numeric-expr)
Returns the absolute value of *numeric-expr.*

ACOS(numeric-expr)
Returns the arc-cosine of *numeric-expr* in radians.

ASIN(numeric-expr)
Returns the arc-sine of *numeric-expr* in radians.

ATAN(numeric-expr)
Returns the arc-tangent of *numeric-expr* in radians.

ATAN2(numeric-expr1, numeric-expr2)
Returns the arc-tangent of *numeric-expr1/numeric-expr2* in radians.

CEILING(numeric-expr)
Returns the ceiling (smallest integer not less than) of *numeric-expr.*

COS(numeric-expr)
Returns the cosine of *numeric-expr*, expressed in radians.

COT(numeric-expr)
Returns the cotangent of *numeric-expr*, expressed in radians.

DEGREES(numeric-expr)
Converts *numeric-expr*, from radians to degrees.

EXP(numeric-expr)
Returns the exponential function of *numeric-expr*.

FLOOR(numeric-expr)
Returns the floor (largest integer not greater than) of *numeric-expr*.

LOG(numeric-expr)
Returns the logarithm of *numeric-expr*.

LOG10(numeric-expr)
Returns the logarithm base 10 of *numeric-expr*.

MOD(dividend, divisor)
Returns the remainder when *dividend* is divided by *divisor*. Division involving a negative *dividend* will give a negative or zero result. The sign of the *divisor* has no effect.

*PI(*)* Returns the numeric value PI.

POWER (numeric-expr1, numeric-expr2)
Raises *numeric-expr1* to the power *numeric-expr2*.

RADIAN (numeric-expr)
Converts *numeric-expr*, from degrees to radians.

RAND ([integer-expr])
Returns a random number in the interval 0 to 1, with *integer-expr* as an optional seed.

REMAINDER(dividend, divisor)
Same as the MOD function.

ROUND (numeric-expr, integer-expr)
Rounds *numeric-expr* to *integer-expr* places after the decimal point. A positive integer determines the number of significant

digits to the right of the decimal point; a negative integer, the number of significant digits to the left of the decimal point.

SIGN(numeric-expr)
Returns the sign of *numeric-expr*.

SIN(numeric-expr)
Returns the sine of *numeric-expr*, expressed in radians.

SQRT(numeric-expr)
Returns the square root of *numeric-expr*.

TAN(numeric-expr)
Returns the tangent of *numeric-expr*, expressed in radians.

TRUNCATE (numeric-expr, integer-expr)
Truncates *numeric-expr* at *integer-expr* places after the decimal point. A positive integer determines the number of significant digits to the right of the decimal point; a negative integer, the number of significant digits to the left of the decimal point.

38.4.3 String functions

String functions act on strings, or return information about strings. If you are working in a multi-byte character set you should check carefully whether the function you are using returns information concerning characters or bytes.

Syntax

String function:

```
| ASCII ( string-expr )                                              |
| BYTE_LENGTH ( string-expr )                                        |
| BYTE_SUBSTR ( string-expr, integer-expr [, integer-expr] )         |
| CHAR ( string-expr )                                               |
| DIFFERENCE (string-expr, string-expr )                             |
| INSERTSTR ( numeric-expr, string-expr, string-expr )               |
| LCASE ( string-expr )                                              |
| LEFT ( string-expr, numeric-expr )                                 |
| LENGTH ( string-expr )                                             |
| LOCATE ( string-expr, string-expr [, numeric-expr ] )              |
| LTRIM ( string-expr )                                              |
| PATINDEX ( '%pattern%', string_expr )                              |
| REPEAT ( string-expr, numeric-expr )                               |
| RIGHT ( string-expr, numeric-expr )                                |
| RTRIM ( string-expr )                                              |
| SIMILAR ( string-expr, string-expr )                               |
| SOUNDEX ( string-expr )                                            |
| SPACE ( integer-expr )                                             |
| STRING ( string-expr [, ...] )                                     |
| SUBSTR ( string-expr, integer-expr [, integer-expr] )              |
| TRIM ( string-expr )                                               |
| UCASE ( string-expr )                                              |
```

ASCII(string-expr)

Returns the integer ASCII value of the first byte in *string-expr*, or 0 for the empty string.

BYTE_LENGTH(string-expr)

Returns the number of bytes in the string *string-expr*.

BYTE_SUBSTR(string-expr, start [, length])

Returns the substring of *string-expr* starting at the given *start* position (origin 1), in bytes. If the *length* is specified, the substring is restricted to that number of bytes. Both *start* and *length* can be negative. A negative starting position specifies a number of bytes from the end of the string instead of the beginning. A positive *length* specifies that the substring ends *length* bytes to the right of the starting position, while a negative *length* specifies that the substring ends *length* bytes to the left of the starting position. Using appropriate combinations of negative and positive numbers, you can get a substring from either the beginning or end of the string.

CHAR(numeric-expr)

Returns the character with the ASCII value *numeric-expr*. The character in the current character set corresponding to the supplied

numeric expression modulo 256 is returned. If you are using multi-byte character sets, CHAR may not return a valid character.

DIFFERENCE(string-expr1, string-expr2)
Returns the difference in the soundex values of *string-expr1* and *string-expr2*.

INSERTSTR(numeric-expr, string-expr1, string-expr2)
Inserts *string-expr2* in *string-expr1* at character position *numeric-expr*

LCASE(string-expr)
Converts all characters in *string-expr* to lower case.

LEFT(string-expr, numeric-expr)
Returns the leftmost *numeric-expr* characters of *string-expr*.

LENGTH(string-expr)
Returns the number of characters in the string *string-expr*. If *string-expr* is of binary data type, the LENGTH function behaves as BYTE_LENGTH.

LOCATE(string-expr1, string-expr2 [, numeric-expr])
Returns the character offset (base 1) into the string *string-expr1* of the first occurrence of the string *string-expr2*. If *numeric-expr* is specified, the search will start at that offset into the string.

The first string can be a long string (longer than 255 bytes), but the second is limited to 255 bytes. If a long string is given as the second argument, the function returns a NULL value. If the string is not found, 0 is returned. Searching for a zero-length string will return 1. If any of the arguments are NULL, the result is NULL.

LTRIM(string-expr)
Returns *string-expr* with leading blanks removed.

PATINDEX('%pattern%', string-expr)
Returns an integer representing the starting position in characters of the first occurrence of *pattern* in the specified string expression, or a zero if *pattern* is not found. If the leading percent wild card is omitted, PATINDEX returns one (1) if *pattern* occurs at the beginning of the string, and zero if not. If the trailing percent wild card is omitted, PATINDEX returns one (1) if *pattern* occurs at the end of the string, and zero if not. If *pattern* starts with a percent wild card, the two leading percent wild cards are treated as one.

REPEAT(string-expr, integer-expr)
Returns a string comprised of *integer-expr* instances of *string-expr*, concatenated together.

RIGHT(string-expr, numeric-expr)
Returns the rightmost *numeric-expr* characters of *string-expr.*

RTRIM(string-expr)
Returns *string-expr* with trailing blanks removed.

SIMILAR(string-expr1, string-expr2)
Returns an integer between 0 and 100 representing the similarity between the two strings. The result can be interpreted as the percentage of characters matched between the two strings (100 percent match if the two strings are identical).

This function can be very useful for correcting a list of names (such as customers). Some customers may have been added to the list more than once with slightly different names. Join the table to itself and produce a report of all similarities greater than 90 percent but less than 100 percent.

SOUNDEX(string-expr)
Returns a number representing the sound of the *string-expr.* Although it is not perfect, soundex will normally return the same number for words which sound similar and start with the same letter. For example:

```
soundex( 'Smith') = soundex( 'Smythe' )
```

The soundex function value for a string is based on the first letter and the next three consonants other than H, Y, and W. Doubled letters are counted as one letter. For example,

```
soundex( 'apples' )
```

is based on the letters A, P, L and S. Multi-byte characters are ignored by the SOUNDEX function.

STRING(string1, [string2, .., string99])
Concatenates the strings into one large string. NULL values are treated as empty strings (''). Any numeric or date parameters are automatically converted to strings before concatenation. The STRING function can also be used to force any single expression to be a string by supplying that expression as the only parameter.

SUBSTR(string-expr, start [, length])

Returns the substring of *string-expr* starting at the given *start* position (origin 1). If the *length* is specified, the substring is restricted to that length. Both *start* and *length* can be negative. A negative starting position specifies a number of characters from the end of the string instead of the beginning. A positive *length* specifies that the substring ends *length* characters to the right of the starting position, while a negative *length* specifies that the substring ends *length* characters to the left of the starting position. Using appropriate combinations of negative and positive numbers, you can easily get a substring from either the beginning or end of the string. If *string-expr* is of binary data type, the SUBSTR function behaves as BYTE_SUBSTR.

TRIM(string-expr)

Returns *string-expr* with both leading and trailing blanks removed.

UCASE(string-expr)

Converts all characters in *string-expr* to uppercase.

38.4.4 Date and time functions

Syntax

Date and time function:

```
| DATE ( expression )                                    |
| DATEFORMAT ( datetime-expr, string-expr )              |
| DATENAME ( datepart, date-expr )                       |
| DATETIME ( expression )                                |
| DAY ( date-expr )                                      |
| DAYNAME( date-expr )                                   |
| DAYS ( date-expr )                                     |
| DAYS ( date-expr, date-expr )                          |
| DAYS ( date-expr, integer-expr )                       |
| DOW ( date-expr )                                      |
| HOUR ( datetime-expr )                                 |
| HOURS ( datetime-expr )                                |
| HOURS ( datetime-expr, datetime-expr )                 |
| HOURS ( datetime-expr, integer-expr )                  |
| MINUTE ( datetime-expr )                               |
| MINUTES ( datetime-expr )                              |
| MINUTES ( datetime-expr, datetime-expr )               |
| MINUTES ( datetime-expr, integer-expr )                |
| MOD ( numeric-expr, numeric-expr )                     |
```

```
| MONTH ( date-expr )                                    |
| MONTHNAME( date-expr )                                 |
| MONTHS ( date-expr )                                   |
| MONTHS ( date-expr, date-expr )                        |
| MONTHS ( date-expr, integer-expr )                     |
| NOW ( * )                                              |
| QUARTER( date-expr )                                   |
| SECOND ( expression )                                  |
| SECONDS ( datetime-expr )                              |
| SECONDS ( datetime-expr, datetime-expr )               |
| SECONDS ( datetime-expr, integer-expr )                |
| TODAY ( * )                                            |
| WEEKS ( date-expr )                                    |
| WEEKS ( date-expr, date-expr )                         |
| WEEKS ( date-expr, integer-expr )                      |
| YEAR ( date-expr )                                     |
| YEARS ( date-expr )                                    |
| YEARS ( date-expr, date-expr )                         |
| YEARS ( date-expr, integer-expr )                      |
| YMD ( integer-expr, integer-expr, integer-expr )       |
```

The date and time functions allow manipulation of time units. Most time units (such as MONTH) have four functions for time manipulation, although only two names are used (such as MONTH and MONTHS).

SQL Anywhere also supports several Transact-SQL date and time functions, allowing an alternative way of accessing and manipulating date and time functions. For information about the Transact-SQL date and time functions, see "Compatibility of date and time functions" on page 465.

Arguments to date functions should be converted to dates before being used, so that

```
days ( '1995-11-17', 2 )
```

is not correct, but

```
days ( date( '1995-11-17' ), 2 )
```

is correct.

DAY(date-expr)

Returns a number from 1 to 31 corresponding to the day of the given date.

DAYNAME(date-expr)

Returns the name of the day from the supplied date expression. For example, with the date_order option set to the default value of ymd:

```
SELECT DAYNAME ('1987/05/02')
```

returns the value `Saturday`.

DAYS(datetime-expr)
Return the number of days since an arbitrary starting date.

DAYS(date-expr, date-expr)
Returns the number of days from the first date to the second date. The number may be negative. Hours, minutes and seconds are ignored.

DAYS(date-expr, integer-expr)
Add *integer-expr* days to the given date. If the *integer-expr* is negative, the appropriate number of days are subtracted from the date. Hours, minutes and seconds are ignored.

DOW(date-expr)
Returns a number from 1 to 7 representing the day of the week of the given date, with Sunday=1, Monday=2, and so on.

HOUR(datetime-expr)
Returns a number from 0 to 23 corresponding to the hour component of the given date.

HOURS(datetime-expr)
Return the number of hours since an arbitrary starting date and time.

HOURS(datetime-expr, datetime-expr)
Returns the number of whole hours from the first date/time to the second date/time. The number may be negative.

HOURS(datetime-expr, integer-expr)
Add *integer-expr* hours to the given date/time. If the *integer-expr* is negative, the appropriate number of hours are subtracted from the date/time.

MINUTE(datetime-expr)
Returns a number from 0 to 59 corresponding to the minute component of the given date/time.

MINUTES(datetime-expr)
Return the number of minutes since an arbitrary starting date and time.

MINUTES(datetime-expr, datetime-expr)

Returns the number of whole minutes from the first date/time to the second date/time. The number may be negative.

MINUTES(datetime-expr, integer-expr)

Add *integer-expr* minutes to the given date/time. If the *integer-expr* is negative, the appropriate number of minutes are subtracted from the date/time.

MONTH(date-expr)

Returns a number from 1 to 12 corresponding to the month of the given date.

MONTHNAME(date-expr)

Returns the name of the month from the supplied date expression. For example, with the date_order option set to the default value of ymd:

```
SELECT MONTHNAME ('1987/05/02')
```

returns the value `May`.

MONTHS(datetime-expr)

Return the number of months since an arbitrary starting date. This number is often useful for determining if two date/time expressions are on the same month in the same year.

```
MONTHS( invoice_sent ) = MONTHS( payment_received )
```

Note that comparing the MONTH function would be wrong if a payment were made 12 months after the invoice was sent.

MONTHS(date-expr, date-expr)

Returns the number of whole months from the first date to the second date. The number may be negative. Hours, minutes and seconds are ignored.

MONTHS(date-expr, integer-expr)

Add *integer-expr* months to the given date. If the new date is past the end of the month (such as MONTHS('1992-01-31', 1)) the result is set to the last day of the month. If the *integer-expr* is negative, the appropriate number of months are subtracted from the date. Hours, minutes and seconds are ignored.

QUARTER(date-expr)

Returns the quarter from the supplied date expression. For

example, with the date_order option set to the default value of ymd:

```
SELECT QUARTER ('1987/05/02')
```

returns the value 2.

SECOND(datetime-expr)

Returns a number from 0 to 59 corresponding to the second component of the given date.

SECONDS(datetime-expr)

Return the number of seconds since an arbitrary starting date and time.

SECONDS(datetime-expr, datetime-expr)

Returns the number of whole seconds from the first date/time to the second date/time. The number may be negative.

SECONDS(datetime-expr, integer-expr)

Add *integer-expr* seconds to the given date/time. If the *integer-expr* is negative, the appropriate number of seconds are subtracted from the date/time.

WEEKS(datetime-expr)

Return the number of weeks since an arbitrary starting date. (Weeks are defined as going from Sunday to Saturday, as they do in a North American calendar.) This number is often useful for determining if two dates are in the same week.

```
WEEKS( invoice_sent ) = WEEKS( payment_received )
```

WEEKS(date-expr, date-expr)

Returns the number of whole weeks from the first date to the second date. The number may be negative. Hours, minutes and seconds are ignored.

WEEKS(date-expr, integer-expr)

Add *integer-expr* weeks to the given date. If the *integer-expr* is negative, the appropriate number of weeks are subtracted from the date. Hours, minutes and seconds are ignored.

YEAR(date-expr)

Returns a 4 digit number corresponding to the year of the given date.

YEARS(date-expr)
Same as the YEAR function.

YEARS(date-expr, date-expr)
Returns the number of whole years from the first date to the second date. The number may be negative. Hours, minutes and seconds are ignored. For example, age can be calculated by

```
YEARS( birthdate, CURRENT DATE )
```

YEARS(date-expr, integer-expr)
Add *integer-expr* years to the given date. If the new date is past the end of the month (such as YEARS('1992-02-29', 1)) the result is set to the last day of the month. If the *integer-expr* is negative, the appropriate number of years are subtracted from the date. Hours, minutes, and seconds are ignored.

DATE(expression)
Converts the expression into a date, and removes any hours, minutes or seconds. Conversion errors may be reported.

DATEFORMAT(date-expr, string-expr)
Returns a string representing the date *date-expr* in the format specified by *string-expr*. Any allowable date format can be used for *string-expr* (see the DATE_FORMAT option in "SET OPTION Statement" on page 989). For example,

```
DATEFORMAT('1989-01-01', 'Mmm Dd, yyyy')
```

is

```
'Jan 1, 1989'
```

DATETIME(expression)
Converts the expression into a timestamp. Conversion errors may be reported.

*NOW(*)* Returns the current date and time. This is the historical syntax for CURRENT TIMESTAMP.

*TODAY(*)* Returns today's date. This is the historical syntax for CURRENT DATE.

YMD(year-num, month-num, day-num)
Returns a date value corresponding to the given year, month, and day of the month. If the month is outside the range 1-12, the year

is adjusted accordingly. Similarly, the day is allowed to be any integer: the date is adjusted accordingly. For example,

```
YMD( 1992, 15, 1 ) = 'Mar 1 1993'
YMD( 1992, 15, 1-1 ) = 'Feb 28 1993'
YMD( 1992, 3, 1-1 ) = 'Feb 29 1992'
```

38.4.5 Data type conversion functions

Syntax

Data type conversion function:

```
| CAST ( expression AS datatype )                          |
| CONVERT (datatype, expression [, format-style] )         |
```

The DATE, DATETIME, and DATEFORMAT functions which convert expressions to dates, timestamps, or strings based on a date format are listed in "Date and time functions" on page 774. The STRING function, which converts expressions to a string, is discussed in "String functions" on page 770.

SQL Anywhere carries out many type conversions automatically. For example, if a string is supplied where a numerical expression is required, the string is automatically converted to a number. For more information on automatic data type conversions carried out by SQL Anywhere, see "Data type conversions" on page 764.

CAST(expression AS data-type)

Returns the value of *expression*, converted to the supplied *data-type*. If the length is omitted for character string types, SQL Anywhere chooses an appropriate length. If neither precision nor scale is specified for a DECIMAL conversion, SQL Anywhere selects appropriate values. For example:

```
CAST( '1992-10-31' AS DATE )    --ensure string is used
                                --as a DATE
CAST( 1 + 2 AS CHAR )           --SQL Anywhere chooses
                                --length
CAST( Surname AS CHAR(10) )     --useful for shortening
                                --strings
```

CONVERT(data-type, expression)

Returns the expression converted to data-type. The following is an example of this use of CONVERT:

```
SELECT CONVERT (VARCHAR(12), order_date )
FROM   sales_order ;
```

CONVERT(data-type, expression, format-style)
For converting functions to date data types, the *format-style* is a style code number describing the date format string to be used.

The values of the *format-style* argument have the following meanings:

(yy)	(yyyy)	Output
-	-0 or 100	mon dd yyyy hh:miAM (or PM)
1	101	mm/dd/yy
2	102	yy.mm.dd
3	103	dd/mm/yy
4	104	dd.mm.yy
5	105	dd-mm-yy
6	106	dd mon yy
7	107	mon dd, yy
8	108	hh:mm:ss
-	-9 or 109	mon dd yyyy hh:mi:ss:mmmAM (or PM)
10	110	mm-dd-yy
11	111	yy/mm/dd
12	112	yymmdd

38.4.6 System functions

Syntax

System function:

```
| connection_property ( { integer-expr | string-expr }                |
| . . .       [, integer-expr ] )                                     |
| datalength ( expression )                                           |
| db_id ( [string-expr] )                                             |
| db_name ( [integer-expr])                                           |
| db_property ( { integer-expr | string-expr }                        |
| . . . [,{ integer-expr | string-expr }] )                           |
| next_connection ( { NULL | string-expr } )                         |
| next_database ( { NULL | string-expr } )                           |
| property ( { integer-expr | string-expr } )                        |
| property_name ( integer-expr )                                      |
| property_number ( string-expr )                                     |
| property_description ( { integer-expr | string-expr } )             |
```

Databases currently running on a server or engine are identified by a database name and a database id number. The *db_id* and *db_name* functions provide information on these values.

A set of system functions provides information about properties of a currently running database, or of a connection, on the database engine. These system functions take the database name or id, or the connection name, as an optional argument to identify the database or connection for which the property is requested.

The available properties and their uses are described following the listing of the functions.

connection_property ({ property-id | property-name } [, connection-id])
Returns the value of the given property as a string. The current connection is used if the second argument is omitted.

datalength (expression)
Returns the length of the expression in bytes. The expression is usually a column name. If the expression is a string constant, it must be enclosed in quotes. The following query displays the longest string in the **company_name** column of the **customer** table:

```
SELECT  MAX( DATALENGTH( company_name ) )
FROM    customer
```

db_id ([database-name])
Returns the database ID number. The supplied *database_name* must be a string expression; if it is a constant expression, it must be enclosed in quotes. If no *database_name* is supplied, the ID number of the current database is returned.

db_name ([database-id])
Returns the database name. The supplied *database_id* must be a numeric expression. If no *database_id* is supplied, the name of the current database is returned.

db_property ({ property-id | property-name } [, { database-id | database-name }])
Returns the value of the given property as a string. The current database is used if the second argument is omitted.

next_connection ({ NULL | connection-id })
Returns the next connection number, or the first connection if parm is NULL.

next_database ({ NULL | database-id })
Returns the next database number, or the first connection if parm is NULL.

property ({ property-number | property-name })
Returns the value of the specified property as a string.

property_name (property-number)
Returns the name of the property with the supplied property-number.

property_number (property-name)
Returns the number of the property with the supplied property-name.

property_description ({ property-number | property-name })
Returns a description of the property with the supplied property-name or property-number.

The statistics and properties available are those in the following list. The list includes the name of each property, and a brief description. While each property does have a number as well as a name, the number is subject to change between releases of SQL Anywhere, and should not be used as a reliable identifier for a given property.

Properties available for each connection

Async2Read The number of rereads. A reread occurs when a read request for a page is received by the database IO subsystem while an asynchronous read IO operation has been posted to the operating system but has not completed.

AsyncRead The number of pages that have been read asynchronously from disk.

AsyncWrite The number of pages that have been written asynchronously to disk.

BlockedOn If the current connection is not blocked this is zero. If it is blocked, the connection number on which the connection is blocked due to a locking conflict.

CacheRead The number of database pages that have been looked up in the cache.

CacheReadIndInt The number of index internal-node pages that have been read from the cache.

CacheReadIndLeaf The number of index leaf pages that have been read from the cache.

CacheReadTable The number of table pages that have been read from the cache.

CacheWrite The number of pages (not necessarily distinct) in the cache that have been modified.

CommLink The communication link for the connection. This is one of the network protocols supported by SQL Anywhere, or is "local" for a connection without a SQL Anywhere Client.

Commit The number of Commit requests that have been handled.

ConnReq Current Requests is the number of engine threads that are currently handling a request.

Cursor Cursors is the number of declared cursors that are currently being maintained by the engine.

CursorOpen Open cursors is the number of open cursors that are currently being maintained by the engine.

DiskRead The number of pages that have been read from file.

DiskReadIndInt The number of index internal-node pages that have been read from disk.

DiskReadIndLeaf The number of index leaf pages that have been read from disk.

DiskReadTable The number of table pages that have been read from disk.

DiskSyncRead The number of pages that have been read synchronously from disk.

DiskSyncWrite The number of pages that have been written synchronously to disk. It is the sum of all the other "Disk SyncWrites" counters.

DiskWaitRead The number of times the engine has waited synchronously for the completion of a read IO operation which was originally issued as an asynchronous read. Waitreads often occur due to cache misses on systems that support asynchronous IO.

DiskWaitWrite The number of times engine has waited synchronously for the completion of a write IO operation which was originally issued as an asynchronous write.

DiskWrite The number of modified pages that have been written to disk.

FullCompare The number of comparisons beyond the hash value in an index that have been performed.

HintUsed The number of page-read operations that have been satisfied immediately from cache thanks to a earlier read hint.

IndAdd The number of entries that have been added to indexes.

IndLookup The number of entries that have been looked up in indexes.

LastReqTime The time at which the last request for the specified connection started.

LogFreeCommit The number of Redo Free Commits. A "Redo Free Commit" occurs when a commit of the transaction log is requested but the log has already been written (so the commit was done for "free").

LogRewrite The number of pages that were previously written to the transaction log (but were not full) that have been written to the transaction log again (but with more data added).

LogWrite The number of pages that have been written to the transaction log.

Number The id number of the connection.

PrepStmt The number of prepared statements that are currently being maintained by the engine.

ProcessTime The time since the start of the connection.

ReadHint The number of read hints. A read hint is an asynchronous read operation for a page that the database engine is likely to need soon.

Rlbk The number of Rollback requests that have been handled.

SyncWriteChkpt The number of pages that have been written synchronously to disk for a checkpoint.

SyncWriteExtend The number of pages that have been written synchronously to disk while extending a database file.

SyncWriteFreeCurr The number of pages that have been written synchronously to disk to free a page that cannot remain in the in-memory free list.

SyncWriteFreePush The number of pages that have been written synchronously to disk to free a page that can remain in the in-memory free list.

SyncWriteLog The number of pages that have been written synchronously to the transaction log.

SyncWriteRlbk The number of pages that have been written synchronously to the rollback log.

SyncWriteUnkn The number of pages that have been written synchronously to disk for a reason not covered by other "Disk SyncWrites" counters.

TaskSwitch The number of times the current engine thread has been changed.

TaskSwitchCheck The number of times the current engine thread has volunteered to give up the CPU to another engine thread.

UncommitOp The number of uncommitted operations for the connection.

Userid The user ID for the connection.

VoluntaryBlock The number of engine threads that have voluntarily blocked on pending disk IO.

WaitReadCmp The number of read requests associated with a full comparison (a comparison beyond the hash value in an index) that must be satisfied by a synchronous read operation.

WaitReadOpt The number of read requests posted by the optimizer that must be satisfied by a synchronous read operation.

WaitReadSys The number of read requests posted from the system connection that must be satisfied by a synchronous read operation. The system connection is a special connection used as the context before a connection is made and for operations performed outside of any client connection.

WaitReadTemp The number of read requests for a temporary table that must be satisfied by a synchronous read operation.

WaitReadUnkn The number of read requests from other sources that must be satisfied by a synchronous read operation.

Properties available for the engine

Async2Read The number of rereads. A reread occurs when a read request for a page is received by the database IO subsystem while an asynchronous read IO operation has been posted to the operating system but has not completed.

AsyncRead The number of pages that have been read asynchronously from disk.

AsyncWrite The number of pages that have been written asynchronously to disk.

CacheRead The number of database pages that have been looked up in the cache.

CacheReadIndInt The number of index internal-node pages that have been read from the cache.

CacheReadIndLeaf The number of index leaf pages that have been read from the cache.

CacheReadTable The number of table pages that have been read from the cache.

CacheWrite The number of pages (not necessarily distinct) in the cache that have been modified.

Chkpt The number of checkpoints that have been performed.

ChkptFlush The number of ranges of adjacent pages written out during a checkpoint.

ChkptPage The number of transaction log checkpoints.

CommitFile The number of times the engine has forced a flush of the disk cache. On NT and NetWare platforms, the disk cache does not need to be flushed when unbuffered (direct) IO is used.

ContReq The number of "CONTINUE" requests issued to the engine.

CurrIO The current number of file IOs issued by the engine which have not yet completed.

CurrRead The current number of file reads issued by the engine which have not yet completed.

CurrTaskSwitch The number of current request context switches.

CurrWrite The current number of file writes issued by the engine which have not yet completed.

DiskRead The number of pages that have been read from file.

DiskReadIndInt The number of index internal-node pages that have been read from disk.

DiskReadIndLeaf The number of index leaf pages that have been read from disk.

DiskReadTable The number of table pages that have been read from disk.

DiskSyncRead The number of pages that have been read synchronously from disk.

DiskSyncWrite The number of pages that have been written synchronously to disk. It is the sum of all the other "Disk SyncWrites" counters.

DiskWaitRead The number of times the engine has waited synchronously for the completion of a read IO operation which was originally issued as an asynchronous read. Waitreads often occur due to cache misses on systems that support asynchronous IO.

DiskWaitWrite The number of times engine has waited synchronously for the completion of a write IO operation which was originally issued as an asynchronous write.

DiskWrite The number of modified pages that have been written to disk.

ExtendDBWrite The number of pages by which the database file has been extended.

ExtendTempWrite The number of pages by which temporary files have been extended.

FreeWriteCurr The number of pages freed of those that cannot remain in the in-memory free list.

FreeWritePush The number of pages freed of those that can remain in the in-memory free list.

FullCompare The number of comparisons beyond the hash value in an index that have been performed.

HintUsed The number of page-read operations that have been satisfied immediately from cache thanks to a earlier read hint.

IdleCheck The number of times the engine's idle thread has become active to do idle writes, idle checkpoints, and so on.

IdleChkTime The number of 100'ths of a second spent checkpointing during idle I/O.

IdleChkpt The number of checkpoints completed by the engine's idle thread. An idle checkpoint occurs whenever the idle thread writes out the last dirty page in the cache.

IdleWrite The number of disk writes that have been issued by the engine's idle thread.

IndAdd The number of entries that have been added to indexes.

IndLookup The number of entries that have been looked up in indexes.

LastIdle The number of ticks between requests.

LogFreeCommit The number of Redo Free Commits. A "Redo Free Commit" occurs when a commit of the transaction log is requested but the log has already been written (so the commit was done for "free").

LogRewrite The number of pages that were previously written to the transaction log (but were not full) that have been written to the transaction log again (but with more data added).

LogWrite The number of pages that have been written to the transaction log.

MaxIO The maximum value that "Current IO" has reached.

MaxRead The maximum value that "Current Reads" has reached.

MaxWrite The maximum value that "Current Writes" has reached.

PendingReq The number of new requests detected by the engine.

Port An application-specific number for each client machine, identifying the connection port.

ReadHint The number of read hints. A read hint is an asynchronous read operation for a page that the database engine is likely to need soon.

Req The number of times the engine has been entered to allow it to handle a new request or continue processing an existing request.

ReqType A string for the type of the last request.

SyncWriteChkpt The number of pages that have been written synchronously to disk for a checkpoint.

SyncWriteExtend The number of pages that have been written synchronously to disk while extending a database file.

SyncWriteFreeCurr The number of pages that have been written synchronously to disk to free a page that cannot remain in the in-memory free list.

SyncWriteFreePush The number of pages that have been written synchronously to disk to free a page that can remain in the in-memory free list.

SyncWriteLog The number of pages that have been written synchronously to the transaction log.

SyncWriteRlbk The number of pages that have been written synchronously to the rollback log.

SyncWriteUnkn The number of pages that have been written synchronously to disk for a reason not covered by other "Disk SyncWrites" counters.

UnschReq The number of requests that are currently queued up waiting for an available engine thread.

VoluntaryBlock The number of engine threads that have voluntarily blocked on pending disk IO.

WaitReadCmp The number of read requests associated with a full comparison (a comparison beyond the hash value in an index) that must be satisfied by a synchronous read operation.

WaitReadOpt The number of read requests posted by the optimizer that must be satisfied by a synchronous read operation.

WaitReadSys The number of read requests posted from the system connection that must be satisfied by a synchronous read operation. The system connection is a special connection used as the context before a connection is made and for operations performed outside of any client connection.

WaitReadTemp The number of read requests for a temporary table that must be satisfied by a synchronous read operation.

WaitReadUnkn The number of read requests from other sources that must be satisfied by a synchronous read operation.

Properties available for each database

Alias The database name.

ConnCount The number of connections to the database.

DBNumber The id number of the database.

File The file name of the database root file.

Name The database name, or alias.

Number The id number of the database.

PageSize The page size of the database, in bytes.

38.4.7 Miscellaneous functions

Syntax

Miscellaneous function:

```
| ARGN ( integer-expr, expression [, ...] )                                    |
| COALESCE ( expression, expression [, ...] )                                  |
| ESTIMATE ( column-name [, number [, relation-string]] )                      |
| ESTIMATE_SOURCE ( column-name [, number [,relation-string]] )                |
| IFNULL ( expression, expression [, expression] )                             |
| INDEX_ESTIMATE( column-name, number [, relation-string])                     |
| EXPERIENCE_ESTIMATE( column-name, number [, relation-string])                |
| ISNULL ( expression, expression [, ...] )                                    |
| NUMBER ( * )                                                                 |
| PLAN ( string-expr )                                                         |
| TRACEBACK ( * )                                                              |
```

ARGN(integer-expr, expression [, ...])

Using the value of *integer-expr* as *n*, return the *n*'th argument (starting at 1) from the remaining list of arguments.

COALESCE(expression, expression [... , expression])

Returns the value of the first expression that is not NULL.

ESTIMATE(column-name [, number [, relation-string]])

The *relation-string* must be a comparison operator enclosed in single quotes; the default is `'='`. If *number* is specified, the function returns as a REAL the percentage estimate the query optimizer uses for the following condition:

```
column-name relation number
```

If *number* is not specified, the function returns the estimate used by the query-optimizer for the following condition:

```
column-name relation expression
```

The function returns NULL if the relation-string is not valid. For example, the following query returns the percentage estimate for employee id numbers being greater than 200:

```
SELECT  DISTINCT ESTIMATE( emp_id, 200, '>' )
FROM    employee
```

ESTIMATE_SOURCE(column-name [, number [, relation-string]])

This function is the same as the ESTIMATE function, except that it returns one of the strings **Column**, **Value**, or **Index**, where:

- **Column** means the estimate stored for the column.
- **Value** means the estimate for a particular value, stored in the frequency table.
- **Index** means the estimate derived from an index on the column.

EXPERIENCE_ESTIMATE(column-name [, number [, relation-string]])
This function is the same as the ESTIMATE function, except that it looks only in the frequency table.

IFNULL(expression1, expression2 [, expression3])
If the first expression is the NULL value, then the second expression is returned. Otherwise, the value of the third expression is returned if it was specified. If there was no third expression and the first expression is not NULL then the NULL value is returned.

INDEX_ESTIMATE(column-name [, number [, relation-string]])
This function is the same as the ESTIMATE function, except that it looks only in an index.

ISNULL(expression, expression [... , expression])
Same as the COALESCE function.

*NUMBER(*)*
Generates numbers starting at 1 for each successive row in the results of the query. Although the NUMBER(*) function is useful for generating primary keys when using the INSERT from SELECT statement (see "INSERT Statement" on page 942), the AUTOINCREMENT column is a preferred mechanism for generating sequential primary keys. For information on the AUTOINCREMENT default, see "CREATE TABLE Statement" on page 859.

You should not use the NUMBER(*) function anywhere but in a select-list. The NUMBER(*) values are constructed after all other processing is done, so that including the function in a WHERE clause or a HAVING clause will produce unpredictable results.

In Embedded SQL, care should be exercised when seeking a cursor that references a query containing a NUMBER(*) function. In particular, this function will return negative numbers when a database cursor is positioned relative to the end of the cursor (an absolute seek with a negative offset).

PLAN(string-expr)

Returns the optimization strategy of the SELECT statement *string-expr* as a string.

*TRACEBACK(*)*

Returns a string containing a traceback of the procedures and triggers that were executing when the most recent exception (error) occurred. This is useful for debugging procedures and triggers. To use the traceback function, enter the following after an error occurs while executing a procedure.

```
SELECT TRACEBACK (*)
```

38.5 Expressions

Syntax

```
expression:
        | constant                                          |
        | [correlation-name .] column-name                  |
        | variable-name                                     |
        | function-name ( expression, ... )                 |
        | - expression                                      |
        | expression + expression                           |
        | expression - expression                           |
        | expression * expression                           |
        | expression / expression                           |
        | expression + expression                           |
        | expression || expression                          |
        | ( expression )                                    |
        | ( subquery )                                      |
        | CAST ( expression AS data-type )                  |
        | if-expression                                     |

constant:

        | integer                                           |
        | number                                            |
        | 'string'                                          |
        | special-constant                                  |
        | host-variable                                     |

special-constant:

        | CURRENT DATE                                      |
        | CURRENT TIME                                      |
        | CURRENT TIMESTAMP                                 |
        | NULL                                              |
        | SQLCODE                                           |
        | SQLSTATE                                          |
        | USER                                              |

if-expression:

    IF condition THEN expression [ ELSE expression ] ENDIF
```

Purpose

To specify an arithmetic, string or date/time expression.

Usage

Anywhere.

Authorization

Must be connected to the database.

Side effects

None.

See also

Search conditions, Data types.

Description

Expressions are formed from the following elements:

- constants
- column names
- variables
- functions
- subqueries
- operators.

38.5.2 Constants in expressions

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

There are several special constants:

CURRENT DATE
: The current year, month and day represented in the DATE data type.

CURRENT TIME
: The current hour, minute, second and fraction of a second represented in the TIME data type. Although the fraction of a second is stored to 6 decimal places, the current time is limited by the accuracy of the system clock. Under DOS and Windows, the clock is only accurate to approximately 1/18th of a second rounded to two decimal places. Under QNX, the clock is accurate to the nearest microsecond.

CURRENT TIMESTAMP
: Combines CURRENT DATE and CURRENT TIME to form a TIMESTAMP value containing year, month, day, hour, minute, second and fraction of a second. Like CURRENT TIME, the accuracy of the fraction of a second is limited by the system clock.

NULL
: The NULL value (see "NULL value" on page 952).

SQLCODE Current SQLCODE value (see "SQL Anywhere Database Error Messages" on page 1035).

SQLSTATE Current SQLSTATE value (see "SQL Anywhere Database Error Messages" on page 1035).

CURRENT USER
A string containing the user ID of the current connection.

CURRENT PUBLISHER
A string containing publisher user ID of the database for SQL Remote replications.

In Embedded SQL, a host variable can also be used in an expression wherever a constant is allowed.

38.5.3 Column names in expressions

A column name is an identifier preceded by an optional correlation name. (A correlation name is usually a table name. See "FROM Clause" on page 915 for more information on correlation names.) If a column name has characters other than letters, digits and underscore, it must be surrounded by quotation marks (""). For example, the following are valid column names:

```
employee.name
address
"date hired"
"salary"."date paid"
```

See "Watcom-SQL language elements" on page 752 for a complete description of identifiers.

38.5.4 Watcom-SQL variables

SQL Anywhere supports three levels of variables:

- Local variables are defined inside a compound statement in a procedure or batch using the DECLARE statement. They exist only inside the compound statement.
- Connection-level variables are defined with a CREATE VARIABLE statement. They belong to the current connection, and disappear when you disconnect from the database or when you use the DROP VARIABLE statement.
- Global variables are SQL Anywhere-supplied variables that have system-supplied values.

Local variables

Local variables are declared using the DECLARE statement, which can be used only within a compound statement (that is, bracketed by the BEGIN and END keywords). The variable is initially set as NULL. The value of the variable can be set using the SET statement, or can be assigned using a SELECT statement with an INTO clause.

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

The following batch of SQL statements illustrates the use of local variables.

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

Running this batch from ISQL gives the message `local_var = 10` on the engine window.

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

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

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

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

Running this batch from ISQL gives the message `local_var = 10` on the engine window.

Connection-level variables

Connection-level variables are declared with the CREATE VARIABLE statement. The CREATE VARIABLE statement can be used anywhere except inside compound statements. Connection-level variables can be passed as parameters to procedures.

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

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

Running this batch from ISQL gives the message `local_var = 10` on the engine window.

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

```
DROP VARIABLE con_var
```

Global variables

Global variables are SQL Anywhere-supplied variables that have values set by the SQL Anywhere engine. For example, the global variable @@version has a value that is the current version number of the database engine.

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

The special constants available in SQL Anywhere, such as CURRENT DATE, CURRENT TIME, USER, SQLSTATE and so on are similar to, but not identical to, global variables. The special constants can be used as defaults for columns; global variables cannot.

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

```
SELECT @@version
```

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

```
CREATE PROCEDURE VersionProc ( OUT ver NUMERIC ( 5, 2 ) )
BEGIN
    SELECT @@version
    INTO   ver;
END
```

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

The following table lists the global variables available in SQL Anywhere

Variable name	Meaning
@@error	Commonly used to check the error status (succeeded or failed) of the most recently executed statement. It contains 0 if the previous transaction succeeded; otherwise, it contains the last error number generated by the system. A statement such as if @@error != 0 return causes an exit if an error occurs. Every SQL statement resets @error, so the status check must immediately follow the statement whose success is in question.
@@identity	Last value inserted into an IDENTITY column by an insert or select into statement. @@identity is reset each time a row is inserted into a table. If a statement inserts multiple rows, @@identity reflects the IDENTITY value for the last row inserted. If the affected table does not contain an IDENTITY column, @@identity is set to 0. The value of @@identity is not affected by the failure of an insert or select into statement, or the rollback of the transaction that contained it. @@identity retains the last value inserted into an IDENTITY column, even if the statement that inserted it fails to commit.
@@isolation	Current isolation level. @@isolation takes the value of the active level.
@@procid	Stored procedure ID of the currently executing procedure.
@@rowcount	Number of rows affected by the last command. @@rowcount is set to zero by any command which does not return rows, such as an if statement. With cursors, @@rowcount represents the cumulative number of rows returned from the cursor result set to the client, up to the last fetch request.
@@servername	Name of the current database server.
@@sqlstatus	Contains status information resulting from the last fetch statement.
@@version	Version of the current version of SQL Anywhere.

38.5.5 Functions in expressions

See "Functions" on page 765 for a description of the functions available in SQL Anywhere.

38.5.6 Subqueries in expressions

A subquery is a SELECT statement (see "SELECT Statement" on page 984 and "Introduction to Subqueries" on page 109) enclosed in parentheses. The SELECT statement must contain one and only one select list item. Usually, the subquery is allowed to return only one row. See "Search conditions" on page 803 for other uses of subqueries. A subquery can be used anywhere that a column name can be used. For example, a subquery can be used in the select list of another SELECT statement.

38.5.7 Watcom-SQL Operators

This section describes the arithmetic and string operators available in SQL Anywhere. For information on comparison operators, see the section "Search conditions" on page 803.

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

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

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

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

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

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

expression || *expression*

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

expression + *expression*

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

(expression)

Parentheses.

IF condition THEN expression1 [ELSE expression2] ENDIF

Evaluates to the value of **expression1** if the specified search condition is TRUE, the value of **expression2** if **condition** is FALSE, and the NULL value if **condition** is UNKNOWN. (See "NULL value" on page 952 and "Search conditions" on the next page for more information about TRUE, FALSE and UNKNOWN conditions.)

38.6 Search conditions

Syntax

search condition:

```
| expression compare expression                                    |
| expression compare ANY ( subquery )                              |
| expression compare ALL ( subquery )                              |
| expression IS [NOT] NULL                                         |
| expression [NOT] LIKE expression [ESCAPE expression]             |
| expression [NOT] BETWEEN expression AND expression               |
| expression [NOT] IN ( expression, ... )                          |
| expression [NOT] IN ( subquery )                                 |
| EXISTS ( subquery )                                              |
| NOT condition                                                    |
| condition AND condition                                          |
| condition OR condition                                           |
| ( condition )                                                    |
| ( condition , estimate )                                         |
| condition IS [NOT] TRUE                                          |
| condition IS [NOT] FALSE                                         |
| condition IS [NOT] UNKNOWN                                       |
```

compare: one of = > < >= <= <> != ~= !< !>

Purpose

To specify a search condition for a WHERE clause, a HAVING clause, a CHECK clause, a JOIN clause or an IF expression.

Usage

Anywhere.

Authorization

Must be connected to the database.

Side effects

None.

See also

Expressions.

Description

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

SQL conditions do not follow boolean logic, where conditions are either true or false. In SQL, every condition evaluates as one of TRUE, FALSE, or UNKNOWN. This is called three valued logic. The result of a comparison is

UNKNOWN if either value being compared is the NULL value. For tables showing how logical operators combine in three-valued logic, see the section "Three-valued logic" on page 810.

Rows satisfy a search condition if and only if the result of the condition is TRUE. Rows for which the condition is UNKNOWN do not satisfy the search condition. For more information about NULL, see the section "NULL value" on page 952.

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

The different types of search condition are as follows:

- Comparison conditions
- BETWEEN conditions
- LIKE conditions
- IN conditions
- ALL or ANY conditions
- EXISTS conditions
- IS NULL conditions
- Conditions with logical operators

38.6.2 Subqueries in search conditions

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

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

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

38.6.3 Comparison conditions

The syntax for comparison conditions is as follows:

```
. . . expression compare expression
```

where `compare` is a comparison operator. The following comparison operators are available in SQL Anywhere:

operator	description
=	Equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
<>	Not equal to
!=	Not equal to
!>	Not greater than
!<	Not less than

Comparisons are case insensitive

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

38.6.4 BETWEEN conditions

The syntax for BETWEEN conditions is as follows:

```
. . . expr [NOT] BETWEEN start-expr AND end-expr
```

The BETWEEN condition can evaluate as TRUE, FALSE, or UNKNOWN. Without the NOT keyword, the condition evaluates as TRUE if **expr** is between **start-expr** and **end-expr**. The NOT keyword reverses the meaning of the condition, leaving UNKNOWN unchanged.

The BETWEEN conditions is equivalent to a combination of two inequalities:

```
expr >= start-expr AND expr <= end-expr
```

38.6.5 LIKE conditions

The syntax for LIKE conditions is as follows:

```
. . . expression [NOT] LIKE pattern [ESCAPE escape-expr]
```

The LIKE condition can evaluate as TRUE, FALSE, or UNKNOWN.

Without the NOT keyword, the condition evaluates as TRUE if **expression** matches the **pattern**. If either **expression** or **pattern** is the NULL value, this condition is UNKNOWN. The NOT keyword reverses the meaning of the condition, leaving UNKNOWN unchanged.

The pattern may contain any number of wild cards. The wild cards are:

- **_ (underscore)** Matches any one character
- **% (percent)** Matches any string of zero or more characters.
- **[]** Matches any single character in the specified range or set.
- **[^]** Matches any single character not in the specified range or set.

All other characters must match exactly.

For example, the search condition

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

is TRUE for any row where **name** starts with the letter `a` and has the letter `b` as its second last character.

If an **escape-expr** is specified, it must evaluate to a single character. The character can precede a percent or an underscore in the **pattern** to prevent the special character from having its special meaning. When escaped in this manner, a percent will match a percent, and an underscore will match an underscore.

Any pattern of length 126 characters or less is matched properly. Patterns of length greater than 254 characters are not matched properly. Patterns of length between 127 and 254 characters may or may not be matched properly, depending on the contents of the pattern.

Using ranges and sets in patterns

Ranges and sets of characters can be given in LIKE search conditions using square brackets.

Searching for one of a set of characters

A set of characters to look for is specified by listing the characters inside the brackets. For example, the following condition finds the strings `'smith'` and `'smyth'`:

```
. . . LIKE 'sm[iy]th'
```

Searching for one of a range of characters

A range of characters to look for is specified by giving the ends of the range, separated by a hyphen. For example, the following condition finds the strings `'bough'` and `'rough'`, but not `'tough'`:

```
. . . LIKE '[a-r]ough'
```

The range of characters `[a-z]` is interpreted as "greater than or equal to `a`, and less than or equal to `z`", where the greater than and less than operations are carried out within the collation of the database. For information on ordering of characters within a collation, see the chapter "Database Collations" on page 287.

The lower end of the range must precede the higher end of the range. For example, any LIKE condition containing the expression `[z-a]` returns no rows, as no character matches the `[z-a]` range.

Unless the database was created as a case-insensitive database, the range of characters is case insensitive. For example, the following condition finds the strings `'Bough'`, `'rough'`, and `'TOUGH'`:

```
. . . LIKE '[a-z]ough'
```

If the database is created as a case-sensitive database, the search condition is case sensitive also.

Combining searches for ranges and sets

You can combine ranges and sets within a square bracket. For example, the following condition finds the strings `'bough'`, `'rough'`, and `'tough'`:

```
. . . LIKE '[a-rt]ough'
```

The bracket `[a-mpqs-z]` is interpreted as "exactly one character that is either in the range `a` to `m` inclusive, or is `p`, or is `q`, or is in the range `s` to `z` inclusive".

Searching for one character not in a range

The caret character (^) is used to specify a range of characters that is excluded from a search. For example, the following condition finds the string `'tough'`, but not the strings `'rough'`, or `'bough'`:

```
. . . LIKE '[^a-r]ough'
```

The caret negates the entire rest of the contents of the brackets. For example, the bracket `[^a-mpqs-z]` is interpreted as "exactly one character that is not in the range `a` to `m` inclusive, is not `p`, is not `q`, and is not in the range `s` to `z` inclusive".

Special cases of ranges and sets

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

Other special cases are as follows:

- The expression `[a-]` matches either of the characters `a` or `-`.
- The expression `[]` is never matched and always returns no rows.
- The expressions `[` or `[abp-q` are ill-formed expressions, and give syntax errors.
- You cannot use wild cards inside square brackets. The expression `[a%b]` finds one of `a`, `%`, or `b`.
- You cannot use the caret character to negate ranges except as the first character in the bracket. The expression `[a^b]` finds one of `a`, `^`, or `b`.

38.6.6 IN conditions

The syntax for IN conditions is as follows:

```
. . . expression [NOT] IN ( value-expr1 [, value-expr2 ] ... )
```

Without the NOT keyword, the IN conditions is TRUE if **expression** equals any of the listed values, UNKNOWN if **expression** is the NULL value, and FALSE otherwise. The NOT keyword reverses the meaning of the condition, leaving UNKNOWN unchanged.

38.6.7 ALL or ANY conditions

The syntax for ANY conditions is

```
. . . expression compare ANY ( subquery )
```

where `compare` is a comparison operator.

For example, an ANY condition with an equality operator:

```
. . . expression = ANY ( subquery )
```

is TRUE if **expression** is equal to any of the values in the result of the subquery, and FALSE is the expression is not NULL and does not equal any of the columns of the subquery. The ANY condition is UNKNOWN if **expression** is the NULL value unless the result of the subquery has no rows, in which case the condition is always FALSE.

The keyword SOME can be used instead of ANY.

38.6.8 EXISTS conditions

The syntax for EXISTS conditions is as follows:

```
. . . EXISTS( subquery )
```

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

38.6.9 IS NULL conditions

The syntax for IS NULL conditions is as follows:

```
expression IS [NOT] NULL
```

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

38.6.10 Conditions with logical operators

Search conditions can be combined using AND, OR and NOT.

Conditions are combined using AND as follows:

```
. . . condition1 AND condition2
```

The combined condition is TRUE if both conditions are TRUE, FALSE if either condition is FALSE, and UNKNOWN otherwise.

Conditions are combined using OR as follows:

```
. . . condition1 OR condition2
```

The combined condition is TRUE if either condition is TRUE, FALSE if both conditions are FALSE, and UNKNOWN otherwise. The result of a comparison is UNKNOWN if either value being compared is the NULL value. Rows satisfy a search condition if and only if the result of the condition is TRUE.

38.6.11 NOT conditions

The syntax for NOT conditions is as follows:

```
. . . NOT condition1
```

The NOT condition is TRUE if condition1 is FALSE, FALSE if condition1 is TRUE and UNKNOWN if condition1 is UNKNOWN.

38.6.12 Truth value conditions

The syntax for truth value conditions is as follows:

```
. . . IS [NOT] truth-value
```

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

38.6.13 Three-valued logic

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

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

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

NOT	TRUE	FALSE	UNKNOWN
	FALSE	TRUE	UNKNOWN

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

Figure 18. *Three Valued Logic*

38.7 Comments in Watcom-SQL

Comments are used to attach explanatory text to SQL statements or statement blocks. Comments are not executed by the database engine.

Several comment indicators are available in SQL Anywhere:

-- *(Double hyphen.)* Any remaining characters on the line are ignored by the database engine. This is the SQL/92 comment indicator.

% (Percent sign.) The percent sign has the same meaning as the double hyphen.

// (Double slash.) The double slash has the same meaning as the double hyphen.

/ . . . */ (Slash-asterisk.)* Any characters between the two comment markers are ignored. The two comment markers may be on the same or different lines.

Transact-SQL compatibility

The double-hyphen and the slash-asterisk comment styles are compatible with Transact-SQL.

The following examples illustrate the use of comments:

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

/*
    Lists the names and employee IDs of employees
    who work in the sales department.
*/
CREATE VIEW SalesEmployee AS
SELECT  emp_id, emp_lname, emp_fname
  FROM  "dba".employee
 WHERE  dept_id = 200
```

38.8 SQL Statement Syntax

This section includes an alphabetical listing of all SQL, Embedded SQL, and ISQL statements.

ALTER DBSPACE Statement

Syntax

```
ALTER DBSPACE { dbspace-name | TRANSLOG }

...     | ADD number                                    |
        | RENAME filename                                |
```

Purpose

To modify the characteristics of the main database file or an extra dbspace. To preallocate space for a database or for the transaction log.

Usage

Anywhere.

Authorization

Must have DBA authority.

Side effects

Automatic commit.

See also

CREATE DBSPACE Statement.

Description

Each database is held in one or more files. A dbspace is an internal name associated with each database file. ALTER DBSPACE modifies the main database file (also called the root file) or an extra dbspace. The dbspace names for a database are held in the SYSFILE system table. The default dbspace name for the root file of a database is SYSTEM.

An ALTER DBSPACE with the ADD clause is used to preallocate disk space to a dbspace. It extends the size of a dbspace by the number of pages given by **number**. The page size of a database is defined when the database is created.

The ALTER DBSPACE statement with the ADD clause allows database files to be extended in large amounts before the space is required, rather than the normal 32 pages at a time when the space is needed. This can improve performance for loading large amounts of data and also serves to keep the dbspace files more contiguous within the file system.

The ALTER DBSPACE statement with the special dbspace name TRANSLOG preallocates disk space to the transaction log. Preallocation improves performance if the transaction log is expected to grow quickly. You may want

to use this feature if, for example, you are handling large amounts of binary large objects (blobs), such as bitmaps.

The preallocation is carried out by altering the special dbspace name TRANSLOG, as follows:

```
ALTER DBSPACE TRANSLOG ADD number
```

This extends the size of the transaction log by the number of pages specified.

If you move a database file other than the root file to a different filename, directory, or device, use the ALTER DBSPACE command with RENAME to ensure that SQL Anywhere can find the file when the database is started.

When a multi-file database is started, the start line or ODBC data source description tells SQL Anywhere where to find the root database file. The root database file (which has the default dbspace name SYSTEM) holds the system tables, and SQL Anywhere looks in these system tables to find the location of the other dbspaces. SQL Anywhere then opens each of the other dbspaces.

Using ALTER DBSPACE with RENAME on a root file has no effect.

Examples

Increase the size of the SYSTEM dbspace by 200 pages.

```
ALTER DBSPACE system
ADD   200
```

Rename the file for dbspace SYSTEM_2 ro dbspace2.

```
ALTER DBSPACE system_2
RENAME 'e:\db\dbspace2.db'
```

ALTER PROCEDURE Statement

Syntax

```
ALTER PROCEDURE [creator.]procedure-name ( [ parameter , ... ] )

. . .   [ RESULT ( result-column , ... ) ]

. . .   [ ON EXCEPTION RESUME ]

. . .   compound-statement


parameter:

        | parameter_mode  parameter-name  data-type [ DEFAULT expr ]  |
        | SQLCODE                                                     |
        | SQLSTATE                                                    |


parameter_mode:

        | IN                                                          |
        | OUT                                                         |
        | INOUT                                                       |


result-column:

        column-name  data-type
```

Purpose

To replace a procedure with a modified version, without having to reassign permissions on the procedure. You must include the entire new procedure in the ALTER PROCEDURE statement.

Usage

Anywhere.

Authorization

Must be the owner of the procedure or be DBA.

Side effects

Automatic commit.

See also

CREATE PROCEDURE Statement.

Description

The ALTER PROCEDURE statement is identical in syntax to the CREATE PROCEDURE statement except for the first word.

The ALTER PROCEDURE statement replaces the entire contents of the CREATE PROCEDURE statement with the contents of the ALTER PROCEDURE statement. Existing permissions on the procedure are maintained, and do not have to be reassigned. If a DROP PROCEDURE and CREATE PROCEDURE were carried out, execute permissions would have to be reassigned.

ALTER PUBLICATION Statement

Syntax

```
ALTER PUBLICATION [creator.]publication-name

        | ADD TABLE article-description
...     | MODIFY TABLE article-description
        | DELETE TABLE [creator.]table-name
        | RENAME publication-name

article-description:

        table-name [ ( column-name, . . . ) ]

...     [ WHERE search-condition ]

...     [ SUBSCRIBE BY expression ]
```

Purpose

To alter the definition of a SQL Remote publication.

Usage

Anywhere.

Authorization

Must have DBA authority, or be owner of the publication.

Side effects

Automatic commit.

See also

CREATE PUBLICATION Statement, DROP PUBLICATION Statement.

Description

The ALTER PUBLICATION statement alters a SQL Remote publication in the database. The contribution to a publication from one table is called an *article*. Changes can be made to a publication by adding, modifying, or deleting articles, or by renaming the publication. If an article is modified, the entire specification of the modified article must be entered.

Example

The following statement adds the **customer** table to the **pub_contact** publication.

```
ALTER PUBLICATION pub_contact (
    ADD TABLE customer
    )
```

ALTER REMOTE MESSAGE TYPE Statement

Syntax

```
ALTER REMOTE MESSAGE TYPE message-system
    ...    ADDRESS address-string

message-system:
        |MAPI   |
        |FILE   |
        |other  |*
```

* A VIM link and an SMTP/POP link are expected to be available in the first quarter of 1996.

Purpose

To change the pulisher's address for a given message system, for a message type that has been created.

Usage

Anywhere.

Authorization

Must have DBA authority.

Side effects

Automatic commit.

See also

CREATE REMOTE MESSAGE TYPE Statement.

Description

The statement changes the publisher's address for a given message type.

The Message Agent sends outgoing messages from a database by one of the supported message links. The extraction utility uses this address when executing the GRANT CONSOLIDATE statement in the remote database.

The address is the publisher's address under the specified message system. If it is an e-mail system, the address string must be a valid e-mail address. If it is a file-sharing system, the address string is a subdirectory of the directory specified by the SQLREMOTE environment variable, or of the current directory if that is not set. You can override this setting on the GRANT CONSOLIDATE statement at the remote database.

TFor the FILE link, the ALTER REMOTE MESSAGE TYPE statement also causes the Message Agent to look for incoming messages in the address given for each message type.

Example

The following statement changes the publisher's address for the FILE message link to **new_addr**.

```
CREATE REMOTE MESSAGE TYPE file
ADDRESS 'new_addr'
```

ALTER TABLE Statement

Syntax

```
ALTER TABLE [creator.]table-name

            | ADD column-definition [column-constraint ...]                  |
            | ADD table-constraint                                           |
            | MODIFY column-definition                                       |
            | MODIFY column-name DEFAULT default-value                       |
            | MODIFY column-name [ NOT ] NULL                                |
            | MODIFY column-name CHECK NULL                                  |
            | MODIFY column-name CHECK ( condition )                         |
  . . .     | DELETE column-name                                             | , . . .
            | DELETE CHECK                                                   |
            | DELETE UNIQUE ( column-name, ... )                             |
            | DELETE PRIMARY KEY                                             |
            | DELETE FOREIGN KEY role-name                                   |
            | RENAME new-table-name                                          |
            | RENAME column-name TO new-column-name                          |

column-definition:

    column-name data-type [ NOT NULL ] [ DEFAULT default-value ]


column-constraint:

            | UNIQUE                                                         |
            | PRIMARY KEY                                                    |
            | REFERENCES table-name [( column-name )] [ actions ]|
            | CHECK ( condition )                                            |

default-value:

            | string                                                         |
            | number                                                         |
            | AUTOINCREMENT                                                  |
            | CURRENT DATE                                                   |
            | CURRENT TIME                                                   |
            | CURRENT TIMESTAMP                                              |
            | NULL                                                           |
            | USER                                                           |
```

```
table-constraint:

        | UNIQUE ( column-name, ... )                    |
        | PRIMARY KEY ( column-name, ... )               |
        | CHECK ( condition )                            |
        | foreign-key-constraint                         |

foreign-key-constraint:

   [NOT NULL] FOREIGN KEY [role-name] [(column-name, ... )]

   . . . REFERENCES table-name [(column-name, ... )]

   . . . [ actions ] [ CHECK ON COMMIT ]

actions:

        [ ON UPDATE action ] [ ON DELETE action ]

action:

        | CASCADE                                        |
        | SET NULL                                       |
        | SET DEFAULT                                    |
        | RESTRICT                                       |
```

Purpose

To modify a table definition.

Usage

Anywhere.

Authorization

Must be the creator of the table or have DBA authority.

Side effects

Automatic commit. The MODIFY and DELETE options close all cursors for the current connection. The ISQL data window is also cleared.

See also

CREATE TABLE Statement, DROP Statement, Data Types.

Description

The ALTER TABLE command changes table attributes (column definitions, constraints) in a table that was previously created. Note that the syntax allows a list of alter clauses; however, only one table-constraint or column-constraint can be added, modified or deleted in one ALTER TABLE statement.

ADD column-definition

Add a new column to the table. The table must be empty to specify NOT NULL.

NULL values

SQL Anywhere optimizes the creation of columns which are allowed to contain the NULL value. The first column that is allowed to contain the NULL value allocates room for eight such columns, and initializes all eight to be the NULL value. (This requires no extra storage.) Thus, the next seven columns added require no changes to the rows of the table. Adding one more column will then allocate room for another eight such columns and then modify each row of the table to allocate the extra space. Consequently, seven out of eight column additions run quickly.

ADD table-constraint

Add a constraint to the table. See "CREATE TABLE Statement" on page 859 for a full explanation of table constraints.

If PRIMARY KEY is specified, the table must not already have a primary key created by the CREATE TABLE command or another ALTER TABLE command.

MODIFY column-definition

Change the length or data type of an existing column in a table. If NOT NULL is specified, a NOT NULL constraint is added to the named column. Otherwise, the NOT NULL constraint for the column will not be changed. If necessary, the data in the modified column will be converted to the new data type. If a conversion error occurs, the operation will fail and the table will be left unchanged.

Deleting an index, constraint, or key

If the column is contained in a uniqueness constraint, a foreign key, or a primary key then the constraint or key must be deleted before the column can be modified. If a primary key is deleted, all foreign keys referencing the table will also be deleted.

NOTE

You cannot MODIFY a table or column constraint. To change a constraint, you must DELETE the old constraint and ADD the new constraint.

MODIFY column-name DEFAULT default-value

Change the default value of an existing column in a table. To remove a default value for a column, specify DEFAULT NULL.

MODIFY column-name [NOT] NULL

Change the NOT NULL constraint on the column to allow or disallow NULL values in the column.

MODIFY column-name CHECK NULL

Delete the check constraint for the column. This statement cannot be used on databases created before Release 5.0.

MODIFY column-name CHECK (condition)

Replace the existing CHECK condition for the column with the one specified. This statement cannot be used on databases created before Release 5.0.

DELETE column-name

Delete the column from the table. If the column is contained in any index, uniqueness constraint, foreign key, or primary key then the index, constraint or key must be deleted before the column can be deleted. This does not delete CHECK constraints that refer to the column.

DELETE CHECK

Delete all check constraints for the table. This includes both table check constraints and column check constraints.

DELETE UNIQUE (column-name,...)

Delete a uniqueness constraint for this table. Any foreign keys referencing this uniqueness constraint (rather than the primary key) will also be deleted.

DELETE PRIMARY KEY

Delete the primary key constraint for this table. All foreign keys referencing the primary key for this table will also be deleted.

DELETE FOREIGN KEY role-name

Delete the foreign key constraint for this table with the given role name.

RENAME new-table-name

Change the name of the table to the **new-table-name**. Note that any applications using the old table name will need to be modified. Also, any foreign keys which were automatically assigned the same name as the old table name will not change names.

RENAME column-name TO new-column-name

Change the name of the column to the **new-column-name**. Note that any applications using the old column name will need to be modified.

ALTER TABLE will be prevented whenever the command affects a table that is currently being used by another connection. ALTER TABLE can be time consuming and the server will not process requests referencing the same table while the command is being processed.

Before Release 5.0, all table and column constraints were held in a single table constraint. Consequently, for these databases individual constraints on columns cannot be deleted using the MODIFY column-name CHECK NULL clause or replaced using the MODIFY column-name CHECK (condition) clause. To use these statements, the entire table constraint should be deleted and the constraints added back using the MODIFY column-name CHECK (condition) clause.

Examples

Add a new column to the employees table showing which office they work in.

```
ALTER TABLE employee
  ADD office CHAR(20) DEFAULT 'Boston'
```

Drop the office column from the employees table.

```
ALTER  TABLE employee
DELETE office
```

The address column in the customer table can currently hold up to 35 characters. Allow it to hold up to 50 characters.

```
ALTER  TABLE customer
MODIFY address CHAR(50)
```

Add a column to the customer table assigning each customer a sales contact.

```
ALTER TABLE customer
  ADD sales_contact INTEGER
      REFERENCES employee (emp_id)
      ON UPDATE CASCADE
      ON DELETE SET NULL
```

This foreign key is constructed with a cascading updates and is set null on deletes. If an employee has their employee id changed, the column is updated to reflect this change. If an employee leaves the company and has their employee id deleted, the column is set to NULL.

CALL Statement

Syntax

[variable =] CALL procedure-name ([expression ,...])

[variable =] CALL procedure-name ([parameter-name = expression ,...])

Purpose

To invoke a procedure.

Usage

Anywhere.

Authorization

Must be the creator of the procedure, have EXECUTE permission for the procedure, or have DBA authority.

Side effects

None.

See also

The chapter "Using Procedures, Triggers, and Batches" on page 215, CREATE PROCEDURE Statement, GRANT Statement, DECLARE Statement.

Description

The CALL statement invokes a procedure that has been previously created with a CREATE PROCEDURE statement. When the procedure completes, any INOUT or OUT parameter values will be copied back.

The argument list can be specified by position or by using keyword format. By position, the arguments will match up with the corresponding parameter in the parameter list for the procedure. By keyword, the arguments are matched up with the named parameters.

All arguments are optional: procedure arguments can be assigned default values in the CREATE PROCEDURE statement, and missing parameters are assigned the default value or, if no default is set, the parameter is set to NULL.

Inside a procedure, a CALL statement can be used in a DECLARE statement when the procedure returns result sets (see "Returning results from procedures" on page 235).

Procedures can return a value (as a status indicator, say) using the RETURN statement. You can save this return value in a variable using using the quality sign as an assignment operator:

```
CREATE VARIABLE returnval INT ;
returnval = CALL proc_integer ( arg1 = val1, ... )
```

Examples

Call the sp_customer_list procedure. This procedure has no parameters, and returns a result set.

```
CALL sp_customer_list()
```

The following ISQL example creates a procedure to return the number of orders placed by the customer whose ID is supplied, creates a variable to hold the result, calls the procedure, and displays the result.

```
% Set the command delimiter to create the procedure
SET OPTION COMMAND_DELIMITER = ';;'

% Create the procedure
CREATE PROCEDURE OrderCount (IN customer_ID INT, OUT Orders INT)
    BEGIN
        SELECT COUNT("DBA".sales_order.id)
        INTO   Orders
        FROM   "DBA".customer
               KEY LEFT OUTER JOIN "DBA".sales_order
        WHERE "DBA".customer.id = customer_ID ;
    END ;;

% Reset the command delimiter to semicolon.
SET OPTION COMMAND_DELIMITER = ';'

% Create a variable to hold the result
CREATE VARIABLE Orders INT ;

% Call the procedure, FOR customer 101
%-----------------------------
CALL OrderCount ( 101, Orders) ;
%-----------------------------

% Display the result
SELECT Orders FROM DUMMY ;
```

CASE Statement

Syntax

```
CASE value-expression
   ...    WHEN [ constant | NULL ] THEN statement-list ...
   ...    [ WHEN [ constant | NULL ] THEN statement-list ] ...
   ...    ELSE statement-list
END CASE
```

Purpose

Select execution path based on multiple cases.

Usage

Procedures, triggers, and batches only.

Authorization

None.

Side effects

None.

See also

The chapter "Using Procedures, Triggers, and Batches" on page 215, Compound statements.

Description

The CASE statement is a control statement that allows you to choose a list of SQL statements to execute based on the value of an expression. If a WHEN clause exists for the value of *value-expression*, the *statement-list* in the WHEN clause is executed. If no appropriate WHEN clause exists, and an ELSE clause exists, the *statement-list* in the ELSE clause is executed. Execution resumes at the first statement after the END CASE.

Example

The following procedure using a case statement classifies the products listed in the product table of the sample database into one of shirt, hat, shorts, or unknown.

The following procedure uses a case statement to classify the results of a query.

```
CREATE PROCEDURE ProductType  (IN product_id INT, OUT type
CHAR(10))
        BEGIN
        DECLARE prod_name CHAR(20) ;
        SELECT name INTO prod_name FROM "DBA"."product"
        WHERE id = product_id;
        CASE prod_name
        WHEN 'Tee Shirt' THEN
                SET type = 'Shirt'
        WHEN 'Sweatshirt' THEN
                SET type = 'Shirt'
        WHEN 'Baseball Cap' THEN
                SET type = 'Hat'
        WHEN 'Visor' THEN
                SET type = 'Hat'
        WHEN 'Shorts' THEN
                SET type = 'Shorts'
        ELSE
                SET type = 'UNKNOWN'
        END CASE ;
        END
```

CHECKPOINT Statement

Syntax

```
CHECKPOINT
```

Purpose

To *checkpoint* the database.

Usage

Anywhere.

Authorization

Must have DBA authority to CHECKPOINT when you are running with multiple users over a LAN. For single-user, no authorization is required.

Side effects

None.

Description

The CHECKPOINT command will checkpoint the database. Checkpoints are also performed automatically by the database engine. It is not normally required for an application to ever issue the CHECKPOINT command. For a full description of checkpoints, see "Backup and Data Recovery" on page 331.

CLOSE Statement

Syntax

CLOSE cursor-name

cursor-name: identifier, or host-variable

Purpose

To close a cursor.

Usage

Embedded SQL, procedures, triggers, and batches. The **host-variable** format is for Embedded SQL only.

Authorization

The cursor must have been previously opened.

Side effects

None.

See also

OPEN Statement, DECLARE CURSOR Statement, PREPARE Statement.

Description

This statement closes the named cursor.

Examples

The following examples close cursors in Embedded SQL.

```
1.  EXEC SQL CLOSE employee_cursor;

2.  EXEC SQL CLOSE :cursor_var;
```

```
CREATE PROCEDURE TopCustomer (OUT TopCompany CHAR(35), OUT TopValue
INT)
BEGIN
    DECLARE err_notfound EXCEPTION FOR SQLSTATE '02000' ;
    DECLARE curThisCust CURSOR FOR
    SELECT company_name, CAST( sum(sales_order_items.quantity *
product.unit_price) AS INTEGER) VALUE
        FROM   customer
           LEFT OUTER JOIN sales_order
           LEFT OUTER JOIN sales_order_items
           LEFT OUTER JOIN product
        GROUP BY company_name ;
    DECLARE ThisValue INT ;
    DECLARE ThisCompany CHAR(35) ;
    SET TopValue = 0 ;
    OPEN curThisCust ;
    CustomerLoop:
        LOOP
            FETCH NEXT curThisCust INTO ThisCompany, ThisValue ;
            IF SQLSTATE = err_notfound THEN
                LEAVE CustomerLoop ;
            END IF ;
            IF ThisValue > TopValue THEN
                SET TopValue = ThisValue ;
                SET TopCompany = ThisCompany ;
            END IF ;
        END LOOP CustomerLoop ;
    CLOSE curThisCust ;
END
```

COMMENT Statement

Syntax

```
COMMENT ON

        | TABLE [creator.]table-name                          |
        | INDEX [creator.]index-name                         |
  ...   | COLUMN [creator.]table-name.column-name            | IS comment
        | FOREIGN KEY [creator.]table-name.role-name         |
        | USER userid                                        |
        | PROCEDURE [creator.]procedure-name                 |
        | TRIGGER [creator.]trigger-name                     |

comment:

        | string                                             |
        | NULL                                               |
```

Purpose

To store a comment in the system tables for a table, an index, a column, a foreign key, a userid, a procedure, or a trigger.

Usage

Anywhere.

Authorization

Must either be the creator of the database object being commented, or have DBA authority.

Side effects

Automatic commit.

Description

Several system tables have a column named Remarks that allows you to associate a comment with a database item (SYSUSERPERM, SYSTABLE, SYSCOLUMN, SYSINDEX, SYSFOREIGNKEY, SYSPROCEDURE, SYSTRIGGER). The COMMENT ON command allows you to set the Remarks column in these system tables. A comment can be removed by setting it to NULL.

When adding a comment to an index or trigger, the creator is the creator of the table on which the index or trigger is defined.

Examples

The following examples show how to add and remove a comment.

Add a comment to the employee table.

```
COMMENT ON TABLE employee IS "Employee information"
```

Remove the comment from the employee table.

```
COMMENT ON TABLE employee IS NULL
```

COMMIT Statement

Syntax

```
COMMIT [ WORK ]
```

Purpose

To make any changes to the database permanent.

Usage

Anywhere.

Authorization

Must be connected to the database.

Side effects

Closes all cursors that were not opened with the WITH HOLD option.

See also

ROLLBACK Statement, PREPARE TO COMMIT Statement, CONNECT Statement, SET CONNECTION Statement, DISCONNECT Statement.

Description

The COMMIT command ends a logical unit of work (transaction) and makes all changes made during this transaction permanent in the database. A transaction is defined as the database work done between successful COMMIT and ROLLBACK commands on a single database connection.

The COMMIT command is also used as the second phase of a two-phase commit operation. See "Coordinating transactions with multiple database engines" on page 212 and "PREPARE TO COMMIT Statement" on page 964 for more information.

The changes committed are those made by the data manipulation commands: INSERT, UPDATE, and DELETE, as well as the ISQL load command INPUT.

The data definition commands all do an automatic commit. They are:

ALTER
COMMENT
CREATE
DROP
GRANT
REVOKE
SET OPTION

The COMMIT command fails if SQL Anywhere detects any invalid foreign keys. This makes it impossible to end a transaction with any invalid foreign keys. Usually, foreign key integrity is checked on each data manipulation operation. However, if either the database option WAIT_FOR_COMMIT is set ON or a particular foreign key was defined with a CHECK ON COMMIT clause, the database engine will not check integrity until the COMMIT command is executed. For a two-phase commit operation, these errors will be reported on the first phase (PREPARE TO COMMIT), not on the second phase (COMMIT).

Compound statements

Syntax

```
[ statement-label : ]

... BEGIN [ [ NOT ] ATOMIC ]

...     [ local-declaration ; ... ]

...     statement-list

...     [ EXCEPTION [ exception-case ... ] ]

... END [ statement-label ]


local-declaration:

        | variable-declaration                                    |
        | cursor-declaration                                      |
        | exception-declaration                                   |
        | temporary-table-declaration                             |


variable-declaration:

        DECLARE variable-name data type


exception-declaration:
        DECLARE exception-name EXCEPTION FOR SQLSTATE
                [ VALUE ] string

exception-case:

        | WHEN exception-name [ ,... ] THEN statement-list        |
        | WHEN OTHERS THEN statement-list                         |
```

Purpose

To specify a statement that groups other statements together.

Usage

Procedures, triggers, and batches only.

Authorization

None.

Side effects

None.

See also

The chapter "Using Procedures, Triggers, and Batches" on page 215, DECLARE CURSOR Statement, DECLARE TEMPORARY TABLE Statement, LEAVE Statement, SIGNAL Statement, RESIGNAL Statement.

Description

The body of a procedure or trigger is a *compound statement*. Compound statements can also be used in control statements within a procedure or trigger.

A compound statement allows one or more SQL statements to be grouped together and treated as a unit. A compound statement starts with the keyword BEGIN and ends with the keyword END. Immediately following the BEGIN, a compound statement can have local declarations that only exist within the compound statement. A compound statement can have a local declaration for a variable, a cursor, a temporary table, or an exception. Local declarations can be referenced by any statement in that compound statement, or in any compound statement nested within it. Local declarations are not visible to other procedures that are called from within a compound statement.

If the ending *statement-label* is specified, it must match the beginning *statement-label*. The LEAVE statement can be used to resume execution at the first statement after the compound statement. The compound statement that is the body of a procedure or triggers has an implicit label that is the same as the name of the procedure or trigger.

See "Using Procedures, Triggers, and Batches" on page 215 for a complete description of compound statements and exception handling.

Example

The body of a procedure or trigger is a compound statement.

```
CREATE PROCEDURE TopCustomer (OUT TopCompany CHAR(35), OUT TopValue
INT)
BEGIN
    DECLARE err_notfound EXCEPTION FOR SQLSTATE '02000' ;
    DECLARE curThisCust CURSOR FOR
    SELECT company_name, CAST( sum(sales_order_items.quantity *
product.unit_price) AS INTEGER) VALUE
        FROM   customer
           LEFT OUTER JOIN sales_order
           LEFT OUTER JOIN sales_order_items
           LEFT OUTER JOIN product
        GROUP BY company_name ;
    DECLARE ThisValue INT ;
    DECLARE ThisCompany CHAR(35) ;
    SET TopValue = 0 ;
    OPEN curThisCust ;
    CustomerLoop:
        LOOP
            FETCH NEXT curThisCust INTO ThisCompany, ThisValue ;
            IF SQLSTATE = err_notfound THEN
                LEAVE CustomerLoop ;
            END IF ;
            IF ThisValue > TopValue THEN
                SET TopValue = ThisValue ;
                SET TopCompany = ThisCompany ;
            END IF ;
        END LOOP CustomerLoop ;
    CLOSE curThisCust ;
END
```

CONFIGURE Statement

Syntax

CONFIGURE

Purpose

To activate the ISQL configuration window.

Usage

ISQL.

Authorization

None.

Side effects

None.

See also

SET OPTION Statement.

Description

The CONFIGURE command activates the ISQL configuration window. This window displays the current settings of all ISQL options. It does not display or allow you to modify database options or options for other software that are stored in the database. The SET OPTIONS command (see "SET OPTION Statement" on page 989) allows you to set all options—ISQL, database and other options.

The **Tab** key can be used to change fields. If a hardware cursor appears in a field, then you are allowed to type a value for the option. If the hardware cursor does not appear, you can use the cursor up and down keys to select one of the allowed values. When you have changed the options, press **Enter** to save the changes and exit, or **Esc** to undo the changes and exit.

If you press **Enter** and you have selected "Save Options to Database" then the options will be written to the SYSOPTION table in the database and the database engine will perform an automatic COMMIT. If you do not select "Save Options to Database", then the options are set temporarily and remain in effect for the current database connection only.

CONNECT Statement

Syntax

Format 1

CONNECT [TO engine-name] [DATABASE database-name]

... [AS connection-name]

... [USER] userid IDENTIFIED BY password

Format 2

CONNECT USING connect-string

engine-name:	identifier, string or host-variable
database-name:	identifier, string or host-variable
connection-name:	identifier, string or host-variable
userid:	identifier, string or host-variable
password:	identifier, string or host-variable
connect-string:	a valid connection string

Purpose

To establish a connection to a database.

Usage

Embedded SQL, ISQL. Format 2 can be used only from ISQL.

Authorization

None.

Side effects

None.

See also

GRANT Statement, DISCONNECT Statement, SET CONNECTION Statement.

Description

The CONNECT command establishes a connection to the database identified by **database-name** running on the engine or server identified by **engine-name**.

If no **engine-name** is specified, the default local database engine will be assumed (the first database engine started). If a local database engine is not running and the SQL Anywhere Client (DBCLIENT) is running, the default server will be assumed (the server name specified when the client was started). If no **database-name** is specified, the first database on the given engine or server will be assumed.

No other database commands are allowed until a successful CONNECT command has been executed. (In Embedded SQL, WHENEVER, SET SQLCA and some DECLARE commands do not generate code and thus may appear before the CONNECT statement in the source file).

The userid and password are used for checking the permissions on SQL commands. If the password or the userid and password are not specified, the user will be prompted to type the missing information.

In Embedded SQL, the userid and password are used for permission checks on all dynamic SQL statements. Static SQL statements use the userid and password specified with the `-l` option on the SQLPP command line. If no `-l` option is given, then the userid and password of the CONNECT command are used for static SQL statements also.

If you are connected to a userid with DBA authority, you can connect to another userid without specifying a password. (The output of DBTRAN requires this capability.) For example, if you are connected to a database as DBA, you can connect without a password with the command:

```
connect other_user_id
```

In Embedded SQL, you can connect without a password by using a host variable for the password and setting the value of the host variable to be the null pointer.

A connection can optionally be named by specifying the AS clause. This allows multiple connections to the same database, or multiple connections to the same or different database servers, all simultaneously. Each connection has its own associated transaction. You may even get locking conflicts between your transactions if, for example, you try to modify the same record in the same database from two different connections.

Multiple connections are managed through the concept of a current connection. After a successful connect statement, the new connection becomes the current one. To switch to a different connection, use the SET CONNECTION statement. The DISCONNECT statement is used to drop connections. p. For Format 2, a connect string is a list of parameter settings of the form KEYWORD=*value*. For a description of valid settings for the connection string, see "Database connection parameters" on page 128. The connect string must be enclosed in single quotes. The ISQL utility has a different behavior from the default Embedded SQL behavior when a CONNECT statement is issued while already connected to a database. If no database or engine is specified in the CONNECT statement, ISQL connects to the current database, rather than to the default database. For example, in the following batch, the two tables are created in the same database.

```
CREATE TABLE t1( c1 int );
CONNECT DBA IDENTIFIED BY SQL;
CREATE TABLE t2 (c1 int );
```

This is a change in behavior for the ISQL utility from previous releases.

Examples

The following are examples of CONNECT usage within Embedded SQL.

```
1.  EXEC SQL CONNECT AS :conn_name
             USER :userid IDENTIFIED BY :password;

2.  EXEC SQL CONNECT USER "dba" IDENTIFIED BY "sql";
```

The following are examples of CONNECT usage from ISQL.

Connect to a database from ISQL. ISQL will prompt for a user ID and a password.

```
CONNECT
```

Connect to the default database as DBA, from ISQL. ISQL will prompt for a password.

```
CONNECT USER "DBA"
```

Connect to the sample database as the DBA, from ISQL.

```
CONNECT
TO          sademo
USER        "DBA"
IDENTIFIED  BY sql
```

Connect to the sample database using a connect string, from ISQL.

```
CONNECT
USING 'UID=DBA;PWD=sql;DBN=sademo'
```

CREATE DATATYPE Statement

Syntax

```
CREATE  DATATYPE user-type data-type
   ...     [ [ NOT ] NULL ]
   ...     [ DEFAULT default-value ]
   ...     [ CHECK ( condition ) ]
user-type:
        identifier
data-type:
        built-in data type, with precision and scale
```

Purpose

To create a user-defined data type in the database.

Usage

Anywhere.

Authorization

Must have RESOURCE authority.

Side effects

Automatic commit.

See also

The section "Data types" on page 755, DROP Statement

Description

User-defined data types are aliases for built-in data types, including precision and scale values where applicable. They improve convenience and encourage consistency in the database.

User-defined data types can have CHECK conditions and DEFAULT conditions associated with them, and you can indicate whether the data type permits NULL values or not. These conditions are inherited by any column defined on the data type. Any conditions explicitly specified on the column override the data type conditions.

The user who creates a data type is automatically made the owner of that data type. No owner can be specified in the CREATE DATATYPE statement. The

user-defined data type name must be unique, and all users can access the data type without using the owner as prefix.

User-defined data types are objects within the database. Their names must conform to the rules for identifiers. User-defined data type names are always case insensitive, as are built-in data type names.

By default, user-defined data types allow NULLs unless the **allow_nulls_by_default** option is set to OFF. In this case, new user-defined data types by default do not allow NULLs. Any columns created on a user-defined data type either allow or do not allow NULLs depending on the setting of the user-defined data type at the time it was created, not on the current setting of the **allow_nulls_by_default** option. Any explicit setting of NULL or NOT NULL in the column definition overrides the user-defined data type setting.

When creating a CHECK condition, you can use a variable name prefixed with the @ sign in the condition. When the data type is used in the definition of a column, such a variable is replaced by the column name. This allows CHECK conditions to be defined on data types and used by columns of any name.

To drop the data type from the database, use the DROP statement. You must be either the owner of the data type or have DBA authority in order to drop a user-defined data type.

Example

The following statement creates a data type named **address**, which holds a 35-character string, and which may be NULL.

```
CREATE DATATYPE address CHAR( 35 ) NULL
```

The following statement creates a data type named **id**, which does not allow NULLS, and which is autoincremented by default.

```
CREATE DATATYPE id INT
NOT NULL
DEFAULT AUTOINCREMENT
```

CREATE DBSPACE Statement

Syntax

CREATE DBSPACE dbspace-name AS filename

Purpose

To create a new database file. This file may be on a different device.

Usage

Anywhere.

Authorization

Must have DBA authority.

Side effects

Automatic commit.

See also

DROP Statement.

Description

The CREATE DBSPACE command creates a new database file. When a database is first initialized using DBINIT, it is composed of one file. All tables and indexes created are placed in that file. CREATE DBSPACE adds a new file to the database. This file can be on a different disk drive than the root file allowing the creation of databases larger than one physical device.

The dbspace-name is an internal name for the database file. The filename is the actual name of the database file, with a path where necessary.

A *filename* without an explicit directory is created in the same directory as the main database file. Any relative directory is relative to the main database file. When you are using a SQL Anywhere, the *filename* is a filename on the server machine. When you are using the SQL Anywhere database server for NetWare, the *filename* should use a volume name (not a drive letter) when an absolute directory is specified.

Each table, including its associated indexes, is contained entirely within one database file. The IN clause of the CREATE TABLE command the dbspace into which a table is placed. Tables are put into the root database file by default.

Example

Create a dbspace called library to hold the LibraryBooks table and its indices.

```
CREATE DBSPACE library
    AS 'e:\dbfiles\library.db' ;

CREATE TABLE LibraryBooks (
             title    char(100),
             author   char(50),
             isbn     char(30),
             ) IN library ;
```

CREATE FUNCTION Statement

Syntax

```
CREATE FUNCTION [creator.]function-name ( [ parameter , ... ] )
   ...    RETURNS data-type
   ...    { EXTERNAL NAME library-call |
   ...    [ ON EXCEPTION RESUME ]
   ...    compound-statement }

parameter:
          | parameter-name  data-type                                    |

library-call:
          '[operating-system:]function-name@library.dll; . . .'

operating-system:
          | OS2                                                          |
          | Windows3X                                                    |
          | Windows95                                                    |
          | WindowsNT                                                    |
          | NetWare                                                      |
```

Purpose

To create a new function in the database.

Usage

Anywhere.

Authorization

Must have RESOURCE authority.

Side effects

Automatic commit.

See also

The chapter "Using Procedures, Triggers, and Batches" on page 215, DROP Statement, Compound statements, GRANT Statement, CREATE PROCEDURE Statement, RETURN Statement.

Description

The CREATE FUNCTION statement creates (stores) a user-defined function in the database. A function can be created for another user by specifying a **creator name.** Subject to permissions, a user-defined function can be used in exactly the same way as other non-aggregate functions.

Parameter names must conform to the rules for other database identifiers such as column names. They must be one of the types supported by SQL Anywhere (see "Data types" on page 755), and must be prefixed by the keyword IN, signifying that the argument is an expression that provides a value to the procedure.

A function using the EXTERNAL NAME clause is a wrapper around a call to an external dynamic link library, and is called an external stored procedure. An external stored procedure can have no clauses other than the EXTERNAL NAME clause following the RETURNS clause. For a description of external procedures, see "Calling external libraries from stored procedures" on page 256.

Example

The following function concatenates a firstname string and a lastname string.

```
CREATE FUNCTION fullname (
firstname CHAR(30), lastname CHAR(30) ) RETURNS CHAR(61)
BEGIN
        DECLARE name CHAR(61) ;
        SET name = firstname || ' ' || lastname ;
        RETURN (name) ;
END
```

The following ISQL statements illustrate the use of the **fullname** function.

Return a full name from two supplied strings:

```
SELECT fullname ('joe','smith')
```

```
fullname('joe','smith')
joe smith
```

List the names of all employees:

```
SELECT fullname (emp_fname, emp_lname)
FROM   employee
```

```
fullname (emp_fname, emp_lname)
Fran Whitney
Matthew Cobb
Philip Chin
Julie Jordan
Robert Breault
  . . .
```

CREATE INDEX Statement

Syntax

```
CREATE [UNIQUE] INDEX index-name
... ON [creator.]table-name
... ( column-name [ ASC | DESC ], ... )
... [ IN dbspace-name ]
```

Purpose

To create an index on a specified table. Indexes are used to improve database performance.

Usage

Anywhere.

Authorization

Must be the creator of the table or have DBA authority.

Side effects

Automatic commit.

See also

DROP Statement.

Description

The CREATE INDEX statement creates a sorted index on the specified columns of the named table. Indexes are automatically used by SQL Anywhere to improve the performance of queries issued to the database as well as for sorting queries where an ORDER BY clause is specified. Once an index is created, it is never referenced again except to delete it using the DROP INDEX command.

There is no way of specifying the index owner in the CREATE INDEX statement. Indexes are automatically owned by the owner of the table on which they are defined. The index name must be unique for each owner.

The UNIQUE constraint ensures that there will not be two rows in the table with identical values in all the columns in the index.

Columns are sorted in ascending (increasing) order unless descending (DESC) is explicitly specified. An index will be used by SQL Anywhere for both an ascending and a descending ORDER BY, no matter whether the index was ascending or descending. However, if an ORDER BY is performed with mixed

ascending and descending attributes, an index will be used only if the index was created with the same ascending and descending attributes.

Indexes cannot be created for views.

By default, the index is placed in the same database file as its table. You can place the index in a separate database file by specifying a **dbspace** name in which to put the index. This feature is of use mainly for large databases, to circumvent the limit of 2 GB per table.

CREATE INDEX is prevented whenever the statement affects a table currently being used by another connection. CREATE INDEX can be time consuming and the server will not process requests referencing the same table while the statement is being processed.

SQL Anywhere automatically creates indexes for primary keys and for unique constraints. These automatically-created indexes are held in the same database file as the table.

Examples

Create a two-column index on the employee table.

```
CREATE INDEX employee_name_index
    ON employee
     ( emp_lname, emp_fname )
```

Create an index on the sales_order_items table for the product id column.

```
CREATE INDEX item_prod
    ON sales_order_items
     ( prod_id )
```

CREATE PROCEDURE Statement

Syntax

```
CREATE PROCEDURE [creator.]procedure-name ( [ parameter , ... ] )

   ...    { EXTERNAL NAME library-call |

   ...    [ RESULT ( result-column , ... ) ]

   ...    [ ON EXCEPTION RESUME ]

   ...    compound-statement }


parameter:

          | parameter_mode  parameter-name  data-type [ DEFAULT expr ]  |
          | SQLCODE                                                     |
          | SQLSTATE                                                    |


parameter_mode:

          | IN                                                          |
          | OUT                                                         |
          | INOUT                                                       |


result-column:

          column-name  data-type


library-call:

          '[operating-system:]function-name@library.dll; . . .'

operating-system:

          | OS2                                                         |
          | Windows3X                                                   |
          | Windows95                                                   |
          | WindowsNT                                                   |
          | NetWare                                                     |
```

Purpose

To create a new procedure in the database.

Usage

Anywhere.

Authorization

Must have RESOURCE authority.

Side effects

Automatic commit.

See also

"Using Procedures, Triggers, and Batches" on page 215, DROP Statement, CALL Statement, Compound statements, GRANT Statement, CREATE FUNCTION Statement, EXECUTE IMMEDIATE Statement

Description

The CREATE PROCEDURE command creates (stores) a procedure in the database. A procedure can be created for another user by specifying a **creator name.** A procedure is invoked with a CALL statement

Parameter names must conform to the rules for other database identifiers such as column names. They must be one of the types supported by SQL Anywhere (see "Data types" on page 755), and must be prefixed by one of the keywords IN, OUT or INOUT. The keywords have the following meanings:

- **IN** argument is an expression that provides a value to the procedure.
- **OUT** argument is a variable that could be given a value by the procedure.
- **INOUT** argument is a variable that provides a value to the procedure, and could be given a new value by the procedure.

When procedures are executed using the CALL Statement, not all parameters need to be specified. If a default value is provided in the CREATE PROCEDURE statement, missing parameters are assigned the default values. If no default value is supplied, the parameter is NULL.

SQLSTATE and **SQLCODE** are special parameters that output the SQLSTATE or SQLCODE value when the procedure ends (they are **OUT** parameters). Whether or not a SQLSTATE and SQLCODE parameter is specified, the SQLSTATE and SQLCODE special constants can always be checked immediately after a procedure call to test the return status of the procedure. However, the SQLSTATE and SQLCODE special constant values are modified by the next SQL statement. Providing SQLSTATE or SQLCODE as procedure arguments allows the return code to be stored in a variable.

A procedure that returns result sets ("Returning results from procedures" on page 235) can have a RESULT clause. The parenthesized list following the RESULT keyword defines the number of result columns and name and type. This information is returned by the Embedded SQL DESCRIBE or by ODBC **SQLDescribeCol** when a CALL statement is being described. Allowable data types are listed in "Data types" on page 755.

The body of a procedure consists of a compound statement. For information about compound statements, see "Compound statements" on page 836.

A procedure using the EXTERNAL NAME clause is a wrapper around a call to an external dynamic link library, and is called an external stored procedure. An external stored procedure can have no clauses other than the EXTERNAL NAME clause following the parameter list. For a description of external procedures, see "Calling external libraries from stored procedures" on page 256.

Example

The following procedure uses a case statement to classify the results of a query.

```
CREATE PROCEDURE ProductType  (IN product_id INT, OUT type
CHAR(10))
        BEGIN
        DECLARE prod_name CHAR(20) ;
        SELECT name INTO prod_name FROM "DBA"."product"
        WHERE id = product_id;
        CASE prod_name
        WHEN 'Tee Shirt' THEN
                SET type = 'Shirt'
        WHEN 'Sweatshirt' THEN
                SET type = 'Shirt'
        WHEN 'Baseball Cap' THEN
                SET type = 'Hat'
        WHEN 'Visor' THEN
                SET type = 'Hat'
        WHEN 'Shorts' THEN
                SET type = 'Shorts'
        ELSE
                SET type = 'UNKNOWN'
        END CASE ;
        END
```

```
CREATE PROCEDURE TopCustomer (OUT TopCompany CHAR(35), OUT TopValue
INT)
BEGIN
    DECLARE err_notfound EXCEPTION FOR SQLSTATE '02000' ;
    DECLARE curThisCust CURSOR FOR
    SELECT company_name, CAST( sum(sales_order_items.quantity *
product.unit_price) AS INTEGER) VALUE
        FROM   customer
           LEFT OUTER JOIN sales_order
           LEFT OUTER JOIN sales_order_items
           LEFT OUTER JOIN product
        GROUP BY company_name ;
    DECLARE ThisValue INT ;
    DECLARE ThisCompany CHAR(35) ;
    SET TopValue = 0 ;
    OPEN curThisCust ;
    CustomerLoop:
        LOOP
            FETCH NEXT curThisCust INTO ThisCompany, ThisValue ;
            IF SQLSTATE = err_notfound THEN
                LEAVE CustomerLoop ;
            END IF ;
            IF ThisValue > TopValue THEN
                SET TopValue = ThisValue ;
                SET TopCompany = ThisCompany ;
            END IF ;
        END LOOP CustomerLoop ;
    CLOSE curThisCust ;
END
```

CREATE PUBLICATION Statement

Syntax

```
CREATE  PUBLICATION [creator.]publication-name
   . . .     ( TABLE article-description,. . . )

article-description:
          table-name [ ( column-name, . . . ) ]
   . . .     [ WHERE search-condition ]
   . . .     [ SUBSCRIBE BY expression ]
```

Purpose

To create a publication for replication with SQL Remote.

Usage

Anywhere.

Authorization

Must have DBA authority.

Side effects

Automatic commit.

See also

Description

The CREATE PUBLICATION statement creates a SQL Remote publication in the database. A publication can be created for another user by specifying a **creator name.**

In SQL Remote, publishing is a two-way operation, as data can be entered at both consolidated and remote databases. In a SQL Remote installation, any consolidated database and all remote databases must have the same publication defined. Running the extraction utility from a consolidated database automatically executes the correct CREATE PUBLICATION statement in the remote database.

Example

```
CREATE PUBLICATION pub_contact (
    TABLE contact
    )
```

CREATE REMOTE MESSAGE TYPE Statement

Syntax

```
CREATE REMOTE MESSAGE TYPE message-system
   . . .    ADDRESS address-string

message-system:
        |MAPI   |
        |FILE   |
        |other  |*
```

* A VIM link and an SMTP/POP link are expected to be available in the first quarter of 1996.

Purpose

To identify a message-link and return address for outgoing messages from a database.

Usage

Anywhere.

Authorization

Must have DBA authority.

Side effects

Automatic commit.

See also

"Adding SQL Remote message types" on page 381, GRANT PUBLISH Statement, GRANT REMOTE Statement, GRANT CONSOLIDATE Statement.

Description

The Message Agent sends outgoing messages from a database using one of the supported message links. Return messages for users employing the specified link are sent to the specified address as long as the remote database is created by the extraction utility. The Message Agent starts links only if it has remote users for those links.

The address is the publisher's address under the specified message system. If it is an e-mail system, the address string must be a valid e-mail address. If it is a file-sharing system, the address string is a subdirectory of the directory set in the SQLREMOTE environment variable, or of the current directory if that is not set. You can override this setting on the GRANT CONSOLIDATE statement at the remote database.

For the FILE link, the CREATE REMOTE MESSAGE TYPE statement also causes the Message Agent to look for incoming messages in the address given for each message type.

The initialization utility creates message types automatically, without an address. Unlike other CREATE statements, the CREATE REMOTE MESSAGE TYPE statement does not give an error if the type exists; instead it alters the type.

Example

When remote databases are extracted using the extraction utility, the following statement sets all recipients of FILE messages to send messages back to the **company** subdirectory of the SQLREMOTE environment variable.

The statement also instructs DBREMOTE to look in the **company** subdirectory for incoming messages.

```
CREATE REMOTE MESSAGE TYPE file
ADDRESS 'company'
```

CREATE SUBSCRIPTION Statement

Syntax

CREATE SUBSCRIPTION TO publication-name [(string)]

. . . FOR userid

Purpose

To create a subscription for a user to a publication.

Usage

Anywhere.

Authorization

Must have DBA authority.

Side effects

Automatic commit.

See also

"Setting up subscriptions" on page 398, DROP SUBSCRIPTION Statement, GRANT REMOTE Statement, SYNCHRONIZE SUBSCRIPTION Statement, START SUBSCRIPTION Statement.

Description

In a SQL Remote installation, data is organized into **publications** for replication. In order to receive SQL Remote messages, a **subscription** must be created for a user ID with REMOTE permissions.

If a string is supplied in the subscription, it is matched against each SUBSCRIBE BY expression in the publication. The subscriber receives all rows for which the value of the expression is equal to the supplied string.

In SQL Remote, publications and subscriptions are two-way relationships. If you create a subscription for a remote user to a publication on a consolidated database, you should also create a subscription for the consolidated database on the remote database. The extraction utility carries this out automatically.

Example

```
CREATE SUBSCRIPTION TO pub_sales ( 'Eastern' )
FOR p_chin
```

CREATE TABLE Statement

Syntax

```
CREATE [ GLOBAL TEMPORARY ] TABLE [creator.]table-name

. . . (   | column-definition [ column-constraint ... ]                |, ... )
          | table-constraint                                            |

. . .     [ IN dbspace-name ]

. . . [   | ON COMMIT DELETE ROWS                                       |  ]
          | ON COMMIT PRESERVE ROWS                                     |
```

column-definition:

```
column-name data-type [ NOT NULL ] [ DEFAULT default-value ]
```

column-constraint:

```
          | UNIQUE                                                      |
          | PRIMARY KEY                                                 |
          | REFERENCES table-name [( column-name )] [ actions ]|
          | CHECK ( condition )                                         |
```

default-value:

```
          | string                                                      |
          | number                                                      |
          | AUTOINCREMENT                                               |
          | CURRENT DATE                                                |
          | CURRENT TIME                                                |
          | CURRENT TIMESTAMP                                           |
          | NULL                                                        |
          | USER                                                        |
```

```
table-constraint:

        | UNIQUE ( column-name, ... )                                    |
        | PRIMARY KEY ( column-name, ... )                               |
        | CHECK ( condition )                                            |
        | foreign-key-constraint                                         |

foreign-key-constraint:

   [NOT NULL] FOREIGN KEY [role-name] [(column-name, ... )]

   . . . REFERENCES table-name [(column-name, ... )]

   . . . [ actions ] [ CHECK ON COMMIT ]

actions:

        [ ON UPDATE action ] [ ON DELETE action ]

action:

        | CASCADE                                                        |
        | SET NULL                                                       |
        | SET DEFAULT                                                    |
        | RESTRICT                                                       |
```

Purpose

To create a new table in the database.

Usage

Anywhere.

Authorization

Must have RESOURCE authority. To create a table for another user, you must have DBA authority.

Side effects

Automatic commit.

See also

"Creating tables" on page 169, DROP Statement, ALTER TABLE Statement, CREATE DBSPACE Statement, Data Types.

Description

The CREATE TABLE command creates a new table. A table can be created for another user by specifying a **creator** name. If GLOBAL TEMPORARY is not specified, the table is referred to as a *base table*Otherwise, the table is a *temporary table*

The IN clause is allowed only for base tables, and is used to specify in which database file the base table will be created. See "CREATE DBSPACE Statement" on page 845 for more information.

A created temporary table is a table that exists in the database like a base table and remains in the database until it is explicitly removed by a DROP TABLE statement. The rows in a temporary table are only visible to the connection that inserted the rows. Multiple connections from the same or different applications can use the same temporary table at the same time and each connection will only see its own rows. The rows of a temporary table are deleted when the connection ends.

The ON COMMIT clause is only allowed for temporary tables. By default, the rows of a temporary table are deleted on COMMIT.

The parenthesized list following the CREATE TABLE command can contain the following clauses in any order:

column-name data-type [NOT NULL] [DEFAULT default-value]

Define a column in the table. Allowable data types are described in "Data types" on page 755. Two columns in the same table cannot have the same name.

If NOT NULL is specified, or if the column is in a UNIQUE or PRIMARY KEY constraint, the column cannot contain any NULL values. If a DEFAULT value is specified, it will be used as the value for the column in any INSERT statement which does not specify a value for the column. If no DEFAULT is specified, it is equivalent to DEFAULT NULL.

When using DEFAULT AUTOINCREMENT, the *data type* must be one of INTEGER, SMALLINT, FLOAT, or DOUBLE. On INSERTs into the table, if a value is not specified for the autoincrement column, a unique value will be generated. If a value is specified, it will be used. If the value is larger than the current maximum value for the column, that value will be used as a starting point for subsequent INSERTs.

Deleting rows will not decrement the autoincrement counter. Gaps created by deleting rows can only be filled by explicit assignment when using an insert. After doing an explicit insert of a row number less then the maximum, subsequent rows without explicit assignment will be autoincremented with a value of one greater than the previous maximum.

For performance reasons, it is highly recommended that DEFAULT AUTOINCREMENT only be used with columns defined as a PRIMARY KEY or with a UNIQUE constraint; or

columns that are the first column of an index. This will allow the maximum value determined at startup time to be found without scanning the entire table.

table-constraint

Table constraints help ensure the integrity of data in the database. There are four types of integrity constraints:

- A *Unique constraint* identifies one or more columns that uniquely identify each row in the table.
- A *primary key constraint* is the same as a unique constraint except that a table can have only one primary key constraint. The primary key usually identifies the best identifier for a row. For example, the customer number might be the primary key for the customer table.
- A *foreign key constraint* restricts the values for a set of columns to match the values in a primary key or uniqueness constraint of another table. For example, a foreign key constraint could be used to ensure that a customer number in an invoice table corresponds to a customer number in the customer table.
- A *check constraint* allows arbitrary conditions to be verified. For example, a check constraint could be used to ensure that a column called **Sex** only contains the values male or female.

If a statement would cause changes to the database that would violate an integrity constraint, the statement is effectively not executed and an error is reported. (*Effectively* means that any changes made by the statement before the error was detected are undone.)

column-constraint

Column constraints are abbreviations for the corresponding table constraints. For example, the following are equivalent:

```
1.  CREATE TABLE Product (
                product_num  integer UNIQUE
                )

2.  CREATE TABLE Product (
                product_num  integer,
                UNIQUE ( product_num )
                )
```

Column constraints are normally used unless the constraint references more than one column in the table. In these cases, a table constraint must be used.

Integrity constraints

column-definition UNIQUE or UNIQUE (column-name, ...)

No two rows in the table can have the same values in all the named column(s). A table may have more than one unique constraint.

Unique constraint versus unique index

There is a difference between a *unique constraint* and a *unique index*. Columns in a unique index are allowed to be NULL, while columns in a unique constraint are not. Also, the column referenced by a foreign key can be either a primary key or a column with a unique constraint. Unique indexes cannot be referenced, because they can include multiple NULLs.

column-definition PRIMARY KEY or PRIMARY KEY (column-name, ...)

The primary key for the table will consist of the listed column(s), and none of the named column(s) can contain any NULL values. SQL Anywhere ensures that each row in the table will have a unique primary key value. A table can have only one PRIMARY KEY.

When the second form is used (PRIMARY KEY followed by a list of columns), the primary key is created including the columns in the order in which they are defined, not the order in which they are listed.

column-definition REFERENCES primary-table-name [(primary-column-name)]

The column is a *foreign key* for the primary key or a unique constraint in the primary table. Normally, a foreign key would be for a primary key rather than a unique constraint. If a primary column name is specified, it must match a column in the primary table which is subject to a unique constraint or primary key constraint, and that constraint must consist of only that one column. Otherwise the foreign key references the primary key of the second table.

A temporary table cannot have a foreign key that references a base table and a base table cannot have a foreign key that references a temporary table.

FOREIGN KEY [role-name] [(...)] REFERENCES primary-table-name [(...)]

The table contains a *foreign key* for the primary key or a unique constraint in another table. Normally, a foreign key would be for a primary key rather than a unique constraint. (In this description, this other table will be called the *primary table*.)

If the primary table column names are not specified, then the primary table columns will be the columns in the table's primary key. If foreign key column names are not specified then the foreign key columns will have the same names as the columns in the primary table. If foreign key column names are specified, then the primary key column names must be specified, and the column names are paired according to position in the lists.

Any foreign key column not explicitly defined will automatically be created with the same data type as the corresponding column in the primary table. These automatically created columns cannot be part of the primary key of the foreign table. Thus, a column used in both a primary key and foreign key must be explicitly created.

The *role name* is the name of the foreign key. The main function of the role name is to distinguish two foreign keys to the same table. If no role name is specified, the role name will be set to the name of the primary table.

The referential integrity action defines the action to be taken to maintain foreign key relationships in the database. Whenever a primary key value is changed or deleted from a database table, there may be corresponding foreign key values in other tables that should be modified in some way. You can specify either an ON UPDATE clause, an ON DELETE clause, or both, followed by one of the following actions:

CASCADE When used with ON UPDATE, update the corresponding foreign keys to match the new primary key value. When used with ON DELETE, deletes the rows from the table that match the deleted primary key.

SET NULL Sets to NULL all the foreign key values that correspond to the updated or deleted primary key.

SET DEFAULT Sets to the value specified by the column(s) DEFAULT clause, all the foreign key values that match the updated or deleted primary key value.

RESTRICT Generates an error if an attempt is made to update or delete a primary key value while there are corresponding foreign keys elsewhere in the database. This was the only form of referential integrity prior to SQL

Anywhere Version 4.0 and is the default action if no action is specified.

The CHECK ON COMMIT clause causes the database to wait for a COMMIT before checking the integrity of this foreign key, overriding the setting of the WAIT_FOR_COMMIT database option. CHECK ON COMMIT can only be used with the RESTRICT action. If you use the short form of CHECK ON COMMIT then RESTRICT is implied.

A temporary table cannot have a foreign key that references a base table and a base table cannot have a foreign key that references a temporary table.

column-definition CHECK (condition) or CHECK (condition)

No row is allowed to fail the condition. If an INSERT or UPDATE statement would cause a row to fail the condition, the operation is not permitted and the effects of the statement are undone.

When is the change rejected?

The change is rejected only if the condition is FALSE; in particular, the change is allowed if the condition is UNKNOWN. (See "NULL value" on page 952 and "Search conditions" on page 803 for more information about TRUE, FALSE, and UNKNOWN conditions.)

Examples

The first two examples are for a library database.

Create a table for a library database to hold book information.

```
CREATE TABLE library_books (
       -- NOT NULL is assumed for primary key columns
       isbn            CHAR(20) PRIMARY KEY,
       copyright_date  DATE,
       title           CHAR(100),
       author          CHAR(50),
       -- column(s) corresponding to primary key of room
       -- will be created
       FOREIGN KEY location REFERENCES room
       )
```

Create a table for a library database to hold information on borrowed books.

```
CREATE TABLE borrowed_book (
       -- Default on insert is that book is borrowed today
       date_borrowed   DATE NOT NULL DEFAULT CURRENT DATE,
       -- date_returned will be NULL until the book is returned
       date_returned   DATE,
       book            CHAR(20) REFERENCES library_books (isbn),
       -- The check condition is UNKNOWN until
       -- the book is returned, which is allowed
       CHECK( date_returned >= date_borrowed )
       )
```

The following example is for a sales database.

Create tables for a sales database to hold order and order item information.

```
CREATE TABLE Orders (
       order_num       INTEGER NOT NULL PRIMARY KEY,
       date_ordered    DATE,
       name            CHAR(80)
       ) ;

CREATE TABLE Order_item (
       order_num       INTEGER NOT NULL,
       item_num        SMALLINT NOT NULL,
       PRIMARY KEY (order_num, item_num),
       -- When an order is deleted, delete all of its items.
       FOREIGN KEY (order_num)
               REFERENCES Orders (order_num)
               ON DELETE CASCADE
       )
```

CREATE TRIGGER Statement

Syntax

```
CREATE TRIGGER trigger-name trigger-time trigger-event
   ...    [ ORDER integer ] ON table-name
   ...    [ REFERENCING [ OLD AS old-name ]
                        [ NEW AS new-name ] ]
                        [ REMOTE AS remote-name ] ]
   ...    [ FOR EACH { ROW | STATEMENT } ]
   ...    [ WHEN ( search-condition ) ]
   ...    [ IF UPDATE (column-name)
   ...    [ {AND | OR} UPDATE (column-name) ] . . . ]
   ...      compound-statement
   ...    [ ELSE IF UPDATE (column-name)
   ...      [ {AND | OR} UPDATE (column-name) ] . . .
   ...      compound-statement
   ...     ENDIF ] ]
trigger-time:
            | BEFORE                                    |
            | AFTER                                     |
            | RESOLVE                                   |
trigger-event:
            | DELETE                                    |
            | INSERT                                    |
            | UPDATE                                    |
            | UPDATE OF column-list                     |
```

Purpose

To create a new trigger in the database.

Usage

Anywhere.

Authorization

Must have RESOURCE authority and have ALTER permissions on the table, or must have DBA authority.

Side effects

Automatic commit.

See also

The chapter "Using Procedures, Triggers, and Batches" on page 215, Compound statements, CREATE PROCEDURE Statement, DROP Statement.

Description

The CREATE TRIGGER command creates a trigger associated with a table in the database and stores the trigger in the database.

Triggers can be triggered by the following events:

- **INSERT** Invoked whenever a new row is inserted into the table associated with the trigger.
- **DELETE** Invoked whenever a row of the associated table is deleted.
- **UPDATE** Invoked whenever a row of the associated table is updated.
- **UPDATE OF column-list** Invoked whenever a row of the associated table is updated and a column in the *column-list* has been modified.

The trigger is declared as either a row-level trigger, in which case it executes before or after each row is modified, or as a statement-level trigger, in which case it executes after the entire triggering statement is completed.

Row-level triggers can be defined to execute BEFORE or AFTER the insert, update, or delete. Statement-level triggers execute AFTER the statement.The RESOLVE trigger time is for use with SQL Remote; it fires before row-level UPDATE or UPDATE OF column-lists only. The body of a trigger consists

To declare a trigger as a row-level trigger, use the FOR EACH ROW clause. To declare a trigger as a statement-level trigger, you can either use a FOR EACH STATEMENT clause or omit the FOR EACH clause. For clarity, it is recommended that you enter the FOR EACH STATEMENT clause if declaring a statement-level trigger.

Triggers of the same type (insert, update, or delete) that fire at the same time (before, after, or resolve) can use the ORDER clause to determine the order that the triggers are fired.

The REFERENCING OLD and REFERENCING NEW clauses allow you to refer to the deleted and inserted rows. For the purposes of this clause, an UPDATE is treated as a delete followed by an insert.

The REFERENCING REMOTE clause is for use with SQL Remote. It allows you to refer to the values in the VERIFY clause of an UPDATE statement. It should be used only with RESOLVE UPDATE or RESOLVE UPDATE OF column-list triggers.

The meaning of REFERENCING OLD and REFERENCING NEW differs, depending on whether the trigger is a row-level or a statement-level trigger. For row-level triggers, the REFERENCING OLD clause allows you to refer to the values in a row prior to an update or delete, and the REFERENCING NEW clause allows you to refer to the inserted or updated values. The OLD and NEW rows can be referenced in BEFORE and AFTER triggers. The REFERENCING NEW clause allows you to modify the new row in a BEFORE trigger before the insert or update operation takes place.

For statement-level triggers, the REFERENCING OLD and REFERENCING NEW clauses refer to declared temporary tables holding the old and new values of the rows. The default names for these tables are **deleted** and **inserted**.

The WHEN clause causes the trigger to fire only for rows where the search-condition evaluates to true.

Example

When a new department head is appointed, update the manager_id column for employees in that department.

```
CREATE TRIGGER
tr_manager BEFORE UPDATE OF dept_head_id ON department
REFERENCING OLD AS old_dept
            NEW AS new_dept
FOR EACH ROW
BEGIN
    UPDATE employee
    SET    employee.manager_id=new_dept.dept_head_id
    WHERE  employee.dept_id=old_dept.dept_id
END
```

CREATE VARIABLE Statement

Syntax

```
CREATE VARIABLE identifier data-type
```

Purpose

To create a SQL variable.

Usage

Anywhere.

Authorization

None.

Side effects

None.

See also

Data Types, DROP VARIABLE Statement, SET Statement, Compound statements.

Description

The CREATE VARIABLE command creates a new variable of the specified data type. The variable contains the NULL value until it is assigned a different value by the SET VARIABLE command.

A variable can be used in a SQL statement anywhere a column name is allowed. If there is no column name that matches the identifier, SQL Anywhere checks to see if there is a variable that matches and uses its value.

Variables belong to the current connection, and disappear when you disconnect from the database or when you use the DROP VARIABLE command. Variables are not visible to other connections. Variables are not affected by COMMIT or ROLLBACK statements.

Variables are useful for creating large text or binary objects for INSERT or UPDATE statements from Embedded SQL programs.

Variables created by CREATE VARIABLE can be used in any SQL statement or in any procedure or trigger. Local variables in procedures and triggers are declared within a compound statement (see "Compound statements" on page 230).

Example

The following code fragment could be used to insert a large text value into the database.

```
EXEC SQL BEGIN DECLARE SECTION;
char            buffer[5000];
EXEC SQL END DECLARE SECTION;

EXEC SQL CREATE VARIABLE hold_blob LONG VARCHAR;

EXEC SQL SET hold_blob = '';
for(;;) {
    /* read some data into buffer ... */
    size = fread( buffer, 1, 5000, fp );
    if( size <= 0 ) break;

    /* add data to blob using concatenation
       Note that concatenation works for binary data too! */
    EXEC SQL SET hold_blob = hold_blob || :buffer;
}

EXEC SQL INSERT INTO some_table VALUES ( 1, hold_blob );

EXEC SQL DROP VARIABLE hold_blob;
```

CREATE VIEW Statement

Syntax

```
CREATE  VIEW
    . . .   [creator.]view-name  [( column-name, ... )]
    . . .   AS  select-without-order-by
    . . .   [ WITH CHECK OPTION ]
```

Purpose

To create a view on the database. Views are used to give a different perspective on the data even though it is not stored that way.

Usage

Anywhere.

Authorization

Must have RESOURCE authority and SELECT permission on the tables in the view definition. Must have DBA authority to create a view for another user.

Side effects

Automatic commit.

See also

DROP Statement, CREATE TABLE Statement.

Description

The CREATE VIEW statement creates a view with the given name. A view can be created for another user by specifying the **creator**. A view name can be used in place of a table name in SELECT, DELETE, UPDATE, and INSERT statements. Views, however, do not physically exist in the database as tables. They are derived each time they are used. The view is derived as the result of the SELECT statement specified in the CREATE VIEW command. Table names used in a view should be qualified by the user ID of the table creator. Otherwise, a different user ID might not be able to find the table or might get the wrong table.

The columns in the view are given the names specified in the column name list. If the column name list is not specified, then the view columns are given names from the select list items. In order to use the names from the select list items, the items must be a simple column name or they must have an alias-name specified (see "SELECT Statement" on page 984).

Views can be updated provided the SELECT statement defining the view does not contain a GROUP BY clause, an aggregate function, or involve a UNION operation.

The WITH CHECK OPTION clause rejects any updates and inserts to the view that do not meet the criteria of the view as defined by its SELECT statement.

The SELECT statement must not have an ORDER BY clause on it. It may have a GROUP BY clause and may be a UNION.

Examples

Create a view showing all information for male employees only. This view has the same column names as the base table.

```
CREATE VIEW male_employee
    AS SELECT *
       FROM   Employee
       WHERE  Sex = 'M'
```

Create a view showing employees and the departments they belong to.

```
CREATE VIEW emp_dept
    AS SELECT emp_lname, emp_fname, dept_name
       FROM   Employee JOIN Department
       ON     Employee.dept_id = Department.dept_id
```

DBTOOL Statement

Syntax

```
                | alter-database
                | alter-writefile
                | backup-to
                | compress-database
                | create-database
                | create-writefile
DBTOOL

                | dbinfo-database
                | drop-database
                | translate
                | uncompress-database
                | unload-collation
                | unload-tables
                | validate-tables


alter-database:

   ALTER DATABASE name

   . . .    | NO [ TRANSACTION ] LOG                          |
            | SET [ TRANSACTION ] LOG TO filename             |


alter-writefile:

   ALTER WRITEFILE name [ REFER TO dbname ]


backup-to:

   BACKUP TO directory

   . . .    | [ DBFILE ] [ WRITE FILE ] [ [ TRANSACTION ] LOG ]  |
            | [ ALL FILES ]                                      |

   . . .    | [ RENAME [ TRANSACTION ] LOG ]                     |
            | [ TRUNCATE [ TRANSACTION ] LOG ]                   |

   . . .    [ NOCONFIRM ] USING connection-string
```

compress-database:

```
COMPRESS DATABASE filename [ TO filename ]
```

create-database:

```
CREATE DATABASE filename

...     | [ NO [ TRANSACTION ] LOG ]                    |
        | [ [ TRANSACTION ] LOG TO filename ]           |

...     | [ IGNORE CASE ]                               |
        | [ RESPECT CASE ]                              |

...     [ PAGE SIZE n ] [ COLLATION name ]

...     [ ENCRYPT ] [ TRAILING SPACES ]
```

create-writefile:

```
CREATE WRITEFILE name FOR DATABASE name

...     [ [ TRANSACTION ] LOG TO logname ] [ NOCONFIRM ]
```

dbinfo-database:

```
DBINFO DATABASE filename TO filename [ [ WITH ] PAGE USAGE ]
...     USING connection-string
```

drop-database:

```
DROP DATABASE name [ NOCONFIRM ]
```

translate:

```
TRANSLATE [ TRANSACTION ] LOG FROM logname

...     [ TO sqlfile ] [ WITH ROLLBACKS ]

...     | [ USERS u1, u2, ... , un ]                   |
        | [ EXCLUDE USERS u1, u2, ... , un ]            |

...     [ LAST CHECKPOINT ] [ ANSI ] [ NOCONFIRM ]
```

```
uncompress-database:

   UNCOMPRESS DATABASE filename [ TO filename ] [ NOCONFIRM ]

unload-collation:

   UNLOAD COLLATION [ name ] TO filename
   . . .    USING connection-string

   . . .    [ EMPTY MAPPINGS ] [ HEX | HEXADECIMAL ] [ NOCONFIRM ]

unload-tables:

   UNLOAD TABLES TO directory [ RELOAD FILE TO filename ]

   . . .    | [ DATA ]                                                    |
            | [ SCHEMA ]                                                  |

   . . .    [ UNORDERED ] [ VERBOSE ] USING connection-string

validate-tables:

   VALIDATE TABLES [ t1, t2, ..., tn ] USING connection-string

connection-string:    string of connection parameters
```

Purpose

To invoke one of the database tools.

Usage

ISQL.

Authorization

None.

Side effects

None.

See also

"SQL Anywhere Components" on page 679, the chapter "An Overview of SQL Anywhere" on page 13.

Description

The DBTOOL command invokes one of the database utilities. All of the database utilities are available without leaving ISQL.

The following table lists the database utility invoked by each DBTOOL command. See "SQL Anywhere Components" on page 679 for more information on the database utility programs.

Command	Database tool
DBTOOL ALTER DATABASE	DBLOG
DBTOOL ALTER WRITEFILE	DBWRITE
DBTOOL BACKUP TO	DBBACKUP
DBTOOL COMPRESS DATABASE	DBSHRINK
DBTOOL CREATE DATABASE	DBINIT
DBTOOL CREATE WRITEFILE	DBWRITE
DBTOOL DBINFO DATABASE	DBINFO
DBTOOL DROP DATABASE	DBERASE
DBTOOL TRANSLATE	DBTRAN
DBTOOL UNCOMPRESS DATABASE	DBEXPAND
DBTOOL UNLOAD COLLATION	DBCOLLAT
DBTOOL UNLOAD TABLES	DBUNLOAD
DBTOOL VALIDATE TABLES	DBVALID

Declaration section

Syntax

EXEC SQL BEGIN DECLARE SECTION;

C declarations

EXEC SQL END DECLARE SECTION;

Purpose

To declare host variables in an Embedded SQL program. Host variables are used to exchange data with the database.

Usage

Embedded SQL.

Authorization

None.

See also

"Host variables" on page 544.

Description

A declaration section is simply a section of C variable declarations surrounded by the BEGIN DECLARE SECTION and END DECLARE SECTION statements. A declaration section makes the SQL preprocessor aware of C variables that will be used as host variables. Not all C declarations are valid inside a declaration section. See "Host variables" on page 544 for more information.

Examples

```
EXEC SQL BEGIN DECLARE SECTION;
char            *emp_lname, initials[5];
int             dept;
EXEC SQL END DECLARE SECTION;
```

DECLARE CURSOR Statement

Syntax

```
DECLARE  cursor-name

  . . .              [ | UNIQUE      |
                       | SCROLL      |
                       | NO SCROLL   |
                       | DYNAMIC SCROLL |
                       | INSENSITIVE | ]

  . . .              | CURSOR FOR  statement      |
                     | CURSOR FOR  statement-name |

  . . .              [ FOR UPDATE | FOR READ ONLY ]

cursor-name:         identifier
statement-name:      identifier, or host-variable
```

Purpose

To declare a cursor. Cursors are the primary means for retrieving data from the database using Embedded SQL.

Usage

Embedded SQL, procedures, triggers, and batches. The **statement-name** and **host-variable** formats are for Embedded SQL only.

Authorization

None.

Side effects

None.

See also

"Compound statements" on page 230, PREPARE Statement, OPEN Statement, EXPLAIN Statement, SELECT Statement, CALL Statement.

Description

The DECLARE statement declares a cursor with the specified name for a SELECT statement or a CALL statement.

When a cursor is declared UNIQUE, the query is forced to return all the columns required to uniquely identify each row. Often this will mean ensuring that all of the columns in the primary key or a uniqueness table constraint are returned. Any columns that are required but were not specified will be added. A

DESCRIBE done on a UNIQUE cursor sets the following additional flags in the indicator variables:

- DT_KEY_COLUMN The column is part of the key for the row.
- DT_HIDDEN_COLUMN The column was added to the query, since it was required to uniquely identify the rows.

A cursor declared FOR READ ONLY may not be used in an UPDATE (positioned) or a DELETE (positioned) operation. FOR UPDATE is the default.

A cursor declared NO SCROLL is restricted to FETCH NEXT and FETCH RELATIVE 0 seek operations. A cursor declared SCROLL or DYNAMIC SCROLL can use all formats of the FETCH command. DYNAMIC SCROLL is the default.

SCROLL cursors behave differently from DYNAMIC SCROLL cursors when the rows in the cursor are modified or deleted after the first time the row is read. SCROLL cursors have more predictable behavior when changes happen.

Each row fetched in a SCROLL cursor is remembered. If one of these rows is deleted, either by your program or by another program in a multiuser environment, it creates a "hole" in the cursor. If you fetch the row at this "hole" with a SCROLL cursor, SQL Anywhere returns the error SQLSTATE_NO_CURRENT_ROW indicating that the row has been deleted, and leaves the cursor positioned on the "hole". (A DYNAMIC SCROLL cursor will just skip the "hole" and retrieve the next row.) This allows your application to remember row positions within a cursor and be assured that these positions will not change. For example, an application could remember that Cobb is the second row in the cursor for `SELECT * FROM employee`. If the first employee (Whitney) is deleted while the SCROLL cursor is still open, FETCH ABSOLUTE 2 will still position on Cobb while FETCH ABSOLUTE 1 will return SQLSTATE_NO_CURRENT_ROW. Similarly, if the cursor is on Cobb, FETCH PREVIOUS will return SQLSTATE_NO_CURRENT_ROW.

In addition, a fetch on a SCROLL cursor will return the warning SQLSTATE_ROW_UPDATED_WARNING if the row has changed since it was last read. (The warning only happens once; fetching the same row a third time will not produce the warning.) Similarly, an UPDATE (positioned) or DELETE (positioned) statement on a row that has been modified since it was last fetched will return the error SQLSTATE_ROW_UPDATED_SINCE_READ and abort the statement. An application must FETCH the row again before the UPDATE or DELETE will be permitted. Note that an update to any column will cause the warning/error, even if the column is not referenced by the cursor. For example, a cursor on Surname and Initials would report the update even if only the Birthdate column were modified. These update warning and error conditions will not occur in bulk

operations mode (-b database engine command line switch) when row locking is disabled. See "Tuning bulk operations" on page 307.

SQL Anywhere maintains more information about SCROLL cursors than DYNAMIC SCROLL cursors; thus, DYNAMIC SCROLL cursors are more efficient and should be used unless the consistent behavior of SCROLL cursors is required. There is no extra overhead in SQL Anywhere for DYNAMIC SCROLL cursors versus NO SCROLL cursors.

NOTE

The behavior of SCROLL cursors has changed since Watcom SQL Version 3.0. The SCROLL cursors in Watcom SQL Version 3.0 were equivalent to DYNAMIC SCROLL cursors now. Cursors declared with the default scrolling behavior will not have changed since the old default behavior was the SCROLL and the new default behavior is DYNAMIC SCROLL.

A cursor declared INSENSITIVE has its membership fixed when it is opened; a temporary table is created with a copy of all the original rows. FETCHING from an INSENSITIVE cursor does not see the effect of any other INSERT, UPDATE, or DELETE statement, or any other PUT, UPDATE WHERE CURRENT, DELETE WHERE CURRENT operations on a different cursor. It does see the effect of PUT, UPDATE WHERE CURRENT, DELETE WHERE CURRENT operations on the same cursor.

INSENSITIVE cursors make it easier to write an application that deals with cursors, since you only have to worry about changes you make explicitly to the cursor; you do not have to worry about actions taken by other users or by other parts of your application.

INSENSITIVE cursors can be expensive if the cursor is on a lot of rows. ALso, INSENSITIVE cursors are not affected by ROLLBACK or ROLLBACK TO SAVEPOINT; the ROLLBACK is not an operation on the cursor that changes the cursor contents.

INSENSITIVE cursors meet the ODBC requirements for static cursors.

Embedded SQL

Statements are named using the PREPARE command. Cursors can be declared only for a prepared SELECT or CALL.

The DECLARE cursor statement does not generate any C code.

Cursor-name is a string and is supplied by the programmer.

Embedded SQL examples

```
1.  EXEC SQL DECLARE cur_employee SCROLL CURSOR FOR
        SELECT * FROM employee ;

2.  EXEC SQL PREPARE employee_statement
        FROM 'SELECT emp_lname FROM employee' ;
    EXEC SQL DECLARE cur_employee CURSOR FOR employee_statement ;
```

Procedure/trigger example

```
BEGIN
    DECLARE cur_employee CURSOR FOR
                SELECT emp_lname
                FROM   employee ;
    DECLARE name CHAR(40) ;

    OPEN cur_employee;
    LOOP
        FETCH NEXT cur_employee into name ;
            ...
    END LOOP
    CLOSE cur_employee;
END
```

DECLARE TEMPORARY TABLE Statement

Syntax

```
DECLARE LOCAL TEMPORARY TABLE table-name

. . . (   | column-definition [ column-constraint ... ]                    |, ... )
          | table-constraint                                                |

. . . [   | ON COMMIT DELETE ROWS                                          |   ]
          | ON COMMIT PRESERVE ROWS                                        |
```

Purpose

To declare a local temporary table.

Usage

Embedded SQL, procedures, triggers, and batches.

Authorization

Must have RESOURCE authority.

Side effects

None.

See also

"Compound statements" on page 230, CREATE TABLE Statement

Description

The DECLARE LOCAL TEMPORARY TABLE statement declares a temporary table. See "CREATE TABLE Statement" on page 859 for definitions of **column-definition**, **column-constraint**, and **table-constraint**.

Declared local temporary tables within compound statements exist within the compound statement. (See "Compound statements" on page 230). Otherwise, the declared local temporary table exists until the end of the connection.

By default, the rows of a temporary table are deleted on COMMIT.

Embedded SQL example

```
1.  EXEC SQL DECLARE LOCAL TEMPORARY TABLE MyTable (
        number INT
    );
```

Procedure/trigger example

```
BEGIN
    DECLARE LOCAL TEMPORARY TABLE TempTab (
        number INT
    );
    ...
END
```

DELETE Statement

Syntax

```
DELETE [FROM] [creator.]table-name
  ...  [FROM table-list]
  ...  [WHERE search-condition]
```

Purpose

To delete rows from the database.

Usage

Anywhere.

Authorization

Must have DELETE permission on the table.

Side effects

None.

See also

TRUNCATE TABLE Statement, INSERT Statement, INPUT Statement, "FROM Clause" on page 915.

Description

The DELETE statement deletes all the rows from the named table that satisfy the search condition. If no WHERE clause is specified, all rows from the named table are deleted.

The DELETE statement can be used on views provided the SELECT command defining the view has only one table in the FROM clause and does not contain a GROUP BY clause, an aggregate function, or involve a UNION operation.

The optional second FROM clause in the DELETE statement allows rows to be deleted based on joins. If the second FROM clause is present, the WHERE clause qualifies the rows of this second FROM clause. Rows are deleted from the table name given in the first FROM clause.

For a full description of the FROM clause and joins, see "FROM Clause" on page 915.

Examples

Remove employee 105 from the database

```
DELETE
FROM   employee
WHERE  emp_id = 105
```

Remove all data prior to 1993 from the fin_data table.

```
DELETE
FROM   fin_data
WHERE  year < 1993
```

Remove all names from the **contact** table if they are already present in the **customer** table.

```
DELETE
FROM   contact
FROM   contact, customer
WHERE  contact.last_name = customer.lname
  AND  contact.first_name = customer.fname
```

DELETE (positioned) Statement

Syntax

```
DELETE [FROM [creator.]table-name]
  ...    WHERE CURRENT OF cursor-name

cursor-name:    identifier, or host-variable
table-name:     identifier
creator:        identifier
```

Purpose

To delete the data at the current location of a cursor.

Usage

Embedded SQL, procedures, triggers, and batches. The **host-variable** format is for Embedded SQL only.

Authorization

Must have DELETE permission on tables used in the cursor.

Side effects

None.

See also

UPDATE Statement, UPDATE (positioned) Statement, INSERT Statement, PUT Statement.

Description

This form of the DELETE statement deletes the current row of the specified cursor. The current row is defined to be the last row fetched from the cursor.

The positioned DELETE command can be used on a cursor open on a view as long as the SELECT command defining the view does not contain the GROUP BY clause or an aggregate function.

If the cursor is for a joined query (including using a view containing a join), then the FROM clause must be used. Only the current row of the specified table is deleted. The other tables involved in the join are not affected.

Example

Remove the current row from the database.

```
DELETE
WHERE  CURRENT OF cur_employee
```

DESCRIBE Statement

Syntax

```
DESCRIBE
   ...[  | OUTPUT                                  |  ]
         | SELECT LIST FOR                        |
         | INPUT                                  |
         | BIND VARIABLES FOR                     |
         | ALL                                    |

   ...   | statement-name                         | INTO sqlda-name
         | cursor-name                            |

statement-name:   identifier, or host-variable
cursor-name:      declared cursor
sqlda-name:       identifier
```

Purpose

To get information about the host variables required to store data retrieved from the database or host variables used to pass data to the database.

Usage

Embedded SQL.

Authorization

None.

Side effects

None.

See also

DECLARE CURSOR Statement, PREPARE Statement.

Description

The DESCRIBE statement sets up the named SQLDA to describe either the OUTPUT (equivalently SELECT LIST) or the INPUT (BIND VARIABLES) for the named statement. In the INPUT case, DESCRIBE BIND VARIABLES does not set up the data types in the SQLDA: this needs to be done by the application. The ALL keyword allows you to describe INPUT and OUTPUT in one SQLDA. If describing a statement, the statement must have been previously prepared using the PREPARE command with the same statement name and the SQLDA must have been previously allocated (see function **alloc_sqlda**).

If describing a cursor, the cursor must have been previously declared. The default action is to describe the OUTPUT. Only SELECT statements and CALL

statements have OUTPUT. A DESCRIBE OUTPUT on any other statement will indicate no output by setting the **sqld** field of the SQLDA to zero.

SELECT The DESCRIBE OUTPUT statement fills in the data type and length in the SQLDA for each select list item. The name field is also filled in with a name for the select list item. If an alias is specified for a select list item, the name will be that alias. Otherwise, the name will be derived from the select list item: if the item is a simple column name, it will be used, otherwise, a substring of the expression will be used. DESCRIBE will also put the number of select list items in the **sqld** field of the SQLDA.

When the statement being described is a UNION of two or more SELECT statements, the column names returned for DESCRIBE OUTPUT are the same column names which would be returned for the first SELECT statement.

CALL The DESCRIBE OUTPUT statement fills in the data type, length, and name in the SQLDA for each INOUT or OUT parameter in the procedure. DESCRIBE OUTPUT will also put the number of INOUT or OUT parameters in the **sqld** field of the SQLDA.

CALL (result set) The DESCRIBE OUTPUT statement fills in the data type, length, and name in the SQLDA for each RESULT column in the procedure definition. DESCRIBE OUTPUT will also put the number of result columns in the **sqld** field of the SQLDA.

A *bind variable* is a value supplied by the application when the database executes the statements. Bind variables can be considered parameters to the statement. DESCRIBE INPUT will fill in the name fields in the SQLDA with the bind variable names. DESCRIBE INPUT will also put the number of bind variables in the **sqld** field of the SQLDA.

DESCRIBE uses the indicator variables in the SQLDA to provide additional information. DT_PROCEDURE_IN and DT_PROCEDURE_OUT are bits that are set in the indicator variable when a CALL statement is described. DT_PROCEDURE_IN indicates an IN or INOUT parameter and DT_PROCEDURE_OUT indicates an INOUT or OUT parameter. Procedure RESULT columns will have both bits clear. After a describe OUTPUT, these bits can be used to distinguish between statements that have result sets (need to use OPEN, FETCH, RESUME, CLOSE) and statements that do not (need to use EXECUTE). DESCRIBE INPUT will only set DT_PROCEDURE_IN and DT_PROCEDURE_OUT appropriately when a bind variable is an argument to a CALL statement; bind variables within an expression that is an argument in a CALL statement will not set the bits.

DESCRIBE ALL will allow you to describe INPUT and OUTPUT with one request to the database engine. This has a performance benefit in a multiuser environment. The INPUT information will be filled in the SQLDA first, followed by the OUTPUT information. The **sqld** field will contain the total number of INPUT and OUTPUT variables. The DT_DESCRIBE_INPUT bit in the indicator variable will be set for INPUT variables and clear for OUTPUT variables.

Refer to "The SQL descriptor area" on page 566 for more information on the use of the SQLDA structure.

Example

```
sqlda = alloc_sqlda( 3 );
EXEC SQL DESCRIBE OUTPUT FOR employee_statement INTO sqlda;

if( sqlda->sqld > sqlda->sqln ) {
    actual_size = sqlda->sqld;
    free_sqlda( sqlda );
    sqlda = alloc_sqlda( actual_size );
    EXEC SQL DESCRIBE OUTPUT FOR employee_statement INTO sqlda;
}
```

DISCONNECT Statement

Syntax

```
DISCONNECT

                | connection-name                |
          ...   | [ CURRENT ]                     |
                | ALL                             |

connection-name:    identifier, string or host-variable
```

Purpose

To drop a connection with the database.

Usage

Embedded SQL, ISQL.

Authorization

None.

Side effects

None.

See also

CONNECT Statement, SET CONNECTION Statement.

Description

The DISCONNECT command drops a connection with the database engine and releases all resources used by it. If the connection to be dropped was named on the CONNECT statement, then the name can be specified. Specifying ALL will drop all of the application's connections to all database environments. CURRENT is the default and will drop the current connection.

An implicit ROLLBACK is executed on connections that are dropped.

Embedded SQL example

```
EXEC SQL DISCONNECT :conn_name
```

ISQL example

```
DISCONNECT ALL
```

DROP Statement

Syntax

```
DROP
                | DATATYPE datatype-name                          |
                | DBSPACE  dbspace-name                           |
                | INDEX  [[creator].table-name.]index-name        |
                | TABLE  [creator.]table-name                     |
   . . .        | VIEW  [creator.]view-name                       |
                | PROCEDURE  [creator.]procedure-name             |
                | FUNCTION [creator.]function-name                |
                | TRIGGER  [[creator.]table-name.]trigger-name    |
```

Purpose

To remove user-defined data types, dbspaces, indexes, tables, views, procedures, functions, and triggers from the database.

Usage

Anywhere.

Authorization

For DROP DBSPACE, must have DBA authority. For others, must be the creator of the table, index, view, procedure, function, or trigger, or have DBA authority.

Side effects

Automatic commit. Clears the Data window in ISQL. DROP TABLE and DROP INDEX close all cursors for the current connection.

See also

CREATE DATATYPE Statement, CREATE DBSPACE Statement, CREATE TABLE Statement, CREATE INDEX Statement, CREATE VIEW Statement, CREATE PROCEDURE Statement, CREATE FUNCTION Statement, CREATE TRIGGER Statement, ALTER TABLE Statement.

Description

The DROP command removes the definition of the indicated database structure. If the structure is a dbspace, then all tables in that dbspace must be dropped prior to dropping the dbspace. If the structure is a table, all data in the table is automatically deleted as part of the dropping process. Also, all indexes and keys for the table are dropped by the DROP TABLE command.

DROP TABLE, DROP INDEX and DROP DBSPACE are prevented whenever the statement affects a table that is currently being used by another connection.

DROP PROCEDURE and DROP FUNCTION are prevented when the procedure is in use by another connection.

DROP DATATYPE is prevented if the data type is used in a table. You must change data types on all columns defined on the user-defined data type in order to drop the data type.

Examples

Drop the department table from the database.

```
DROP TABLE department
```

Drop the emp_dept view from the database.

```
DROP VIEW emp_dept
```

DROP CONNECTION Statement

Syntax

```
DROP CONNECTION connection-id
```

Purpose

To drop a connection to the database, belonging to any user.

Usage

Anywhere. You must be connected to the same database on the same server as the connection id to execute this statement.

Authorization

Must have DBA authority.

Side effects

None.

See also

CONNECT Statement.

Description

The DROP CONNECTION statement disconnects a user from the database by dropping the connection to the database.

The *connection-id* for the connection is obtained using the **connection_property** function to request the connection number. The following statement returns the connection id of the current connection:

```
SELECT connection_property( 'number' )
```

Example

The following statement drops connection with id number 4.

```
DROP CONNECTION 4
```

DROP OPTIMIZER STATISTICS Statement

Syntax

```
DROP OPTIMIZER STATISTICS
```

Purpose

To reset optimizer statistics.

Usage

Anywhere.

Authorization

Must have DBA authority.

Side effects

None.

Description

The DROP OPTIMIZER STATISTICS command resets optimizer statistics. The SQL Anywhere optimizer maintains statistics as it evaluates queries and uses these statistics to make better decisions optimizing subsequent queries. These statistics are stored permanently in the database. The DROP OPTIMIZER STATISTICS command resets these statistics to default values. This command is most useful when first learning about the optimizer. It helps to better understand the process used by the optimizer.

DROP PUBLICATION Statement

Syntax

DROP PUBLICATION [creator.]publication-name

Purpose

To drop a SQL Remote publication.

Usage

Anywhere.

Authorization

Must have DBA authority.

Side effects

Automatic commit. All subscriptions to the publication are dropped.

See also

CREATE PUBLICATION Statement.

Description

The DROP PUBLICATION statement drops an existing publication from the database.

Publication definitions are held at both consolidated and remote databases in a SQL Remote installation.

Example

```
DROP PUBLICATION pub_contact
```

DROP REMOTE MESSAGE TYPE Statement

Syntax

DROP REMOTE MESSAGE TYPE message-system

message-system:

|MAPI |
|FILE |
|other |*

* A VIM link and an SMTP/POP link are expected to be available in the first quarter of 1996.

Purpose

To delete a message type definition from a database.

Usage

Anywhere.

Authorization

Must have DBA authority. To be able to drop the type, there must be no user granted REMOTE or CONSOLIDATE permissions with this type.

Side effects

Automatic commit.

See also

"Adding SQL Remote message types" on page 381, CREATE REMOTE MESSAGE TYPE Statement, ALTER REMOTE MESSAGE TYPE Statement.

Description

The statement removes a message type from a database.

Example

The following statement drops the FILE message type from a database.

```
DROP REMOTE MESSAGE TYPE file
```

DROP STATEMENT Statement

Syntax

DROP STATEMENT [creator.]statement-name

statement-name: identifier, or host-variable

Purpose

To free statement resources.

Usage

Embedded SQL.

Authorization

Must have prepared the statement.

Side effects

None.

See also

PREPARE Statement.

Description

The DROP STATEMENT command frees resources used by the named prepared statement. These resources are allocated by a successful PREPARE command, and are normally not freed until the database connection is released.

Example

```
1.  EXEC SQL DROP STATEMENT S1;

2.  EXEC SQL DROP STATEMENT :stmt;
```

DROP VARIABLE Statement

Syntax

```
DROP VARIABLE identifier
```

Purpose

To eliminate a SQL variable.

Usage

Anywhere.

Authorization

None.

Side effects

None.

See also

CREATE VARIABLE Statement, SET Statement.

Description

The DROP VARIABLE command eliminates a SQL variable previously created using the CREATE VARIABLE command. Variables will be automatically eliminated when the database connection is released. However, variables are often used for large objects. Thus, eliminating them after use may free up significant resources (primarily disk space).

DROP SUBSCRIPTION Statement

Syntax

DROP SUBSCRIPTION TO publication-name [(string)]

. . . FOR userid,. . .

Purpose

To drop a subscription for a user from a publication.

Usage

Anywhere.

Authorization

Must have DBA authority.

Side effects

Automatic commit.

See also

CREATE SUBSCRIPTION Statement.

Description

Drops a SQL Remote subscription for a user ID to a publication in the current database. The user ID will no longer receive updates when data in the publication is changed.

In SQL Remote, publications and subscriptions are two-way relationships. If you drop a subscription for a remote user to a publication on a consolidated database, you should also drop the subscription for the consolidated database on the remote database to prevent updates on the remote database being sent to the consolidated database.

Example

The following statement drops a subscription for the user ID **SamS** to the publication **pub_contact**.

```
DROP SUBSCRIPTION TO pub_contact
FOR  SamS
```

EXECUTE Statement

Syntax

Format 1

```
EXECUTE  statement-name

  . . . [   | USING DESCRIPTOR sqlda-name                          |
            | [ . . . ARRAY :nnn  ]                               | ]
            | USING host-variable-list                            |

  . . . [   | INTO DESCRIPTOR into-sqlda-name                      |  ]
            | INTO into-host-variable-list                        |
```

statement-name:	identifier, or host-variable
sqlda-name:	identifier
into-sqlda-name:	identifier

Format 2

```
EXECUTE  IMMEDIATE  statement
```

statement:	string, or host-variable

Purpose

To execute a SQL statement.

Usage

Embedded SQL.

Authorization

Permissions are checked on the statement being executed.

Side effects

None.

See also

PREPARE Statement, DECLARE CURSOR Statement.

Description

Format 1 executes the named dynamic statement which was previously prepared. If the dynamic statement contains host variable place holders which supply information for the request (bind variables), then either the **sqlda-name** must specify a C variable which is a pointer to an SQLDA which contains enough descriptors for all bind variables occurring in the statement, or the bind variables must be supplied in the **host-variable-list**.

The optional ARRAY clause can be used with prepared INSERT statements, to allow wide inserts, which insert more than one row at a time and which may improve performance. The value `nnn` is the number of rows to be inserted. The SQLDA must contain nnn * (columns per row) variables. The first row is placed in SQLDA variables 0 to (columns per row)-1, and so on.

OUTPUT from a SELECT statement or a CALL statement is put either into the variables in the variable list or into the program data areas described by the named SQLDA. The correspondence is one to one from the OUTPUT (select list or parameters) to either the host variable list or the SQLDA descriptor array.

Format 2 is a short form to PREPARE and EXECUTE a statement that does not contain bind variables or output. The SQL statement contained in the string or host-variable is immediately executed.

The EXECUTE statement can be used for any SQL command that can be prepared. Cursors are used for SELECT statements or CALL statements that return many rows from the database (see "Cursors in Embedded SQL" on page 555).

After successful execution of an INSERT, UPDATE or DELETE command, the *sqlerrd[2]* field of the SQLCA (SQLCOUNT) is filled in with the number of rows affected by the operation.

Examples

```
1.  EXEC SQL EXECUTE IMMEDIATE
        'DELETE FROM employee WHERE emp_id = 105';

2.  EXEC SQL PREPARE del_stmt FROM
        'DELETE FROM employee WHERE emp_id = :a';
    EXEC SQL EXECUTE del_stmt USING :employee_number;

3.  EXEC SQL PREPARE sel1 FROM
        'SELECT emp_lname FROM employee WHERE emp_id = :a';
    EXEC SQL EXECUTE sel1 USING :employee_number INTO :emp_lname;
```

EXECUTE IMMEDIATE Statement

Syntax

EXECUTE IMMEDIATE 'statement'

Purpose

To enable dynamically constructed statements to be executed from within a procedure. For information about the Embedded SQL EXECUTE IMMEDIATE statement, see "EXECUTE Statement" on page 902.

Usage

Procedures, triggers, and batches only

Authorization

None. The statement is executed with the permissions of the owner of the procedure, not with the permissions of the user who calls the procedure.

Side effects

None. However, if the statement is a data definition statement with an automatic commit as a side effect, then that commit does take place.

See also

CREATE PROCEDURE Statement, Compound statements.

Description

The EXECUTE IMMEDIATE statement extends the range of statements that can be executed from within procedures and triggers. It allows dynamically prepared statements to be executed, such as statements that are constructed using the parameters passed in to a procedure.

Literal strings in the statement must be enclosed in single quotes, and the statement must be on a single line.

Example

The following procedure creates a table, where the table name is supplied as a parameter to the procedure.

The EXECUTE IMMEDIATE statement must all be on a single line if you run this example.

```
CREATE PROCEDURE CreateTableProc( IN tablename char(30) )
BEGIN
    EXECUTE IMMEDIATE 'CREATE TABLE ' || tablename ||
            ' ( column1 INT PRIMARY KEY)'
END
```

To call the procedure and create a table **mytable**:

```
CALL CreateTableProc( 'mytable' )
```

EXIT Statement

Syntax

```
| EXIT |
| QUIT |
| BYE  |
```

Purpose

To leave ISQL.

Usage

ISQL.

Authorization

None.

Side effects

Will do a commit if option COMMIT_ON_EXIT is ON (default); otherwise will do a rollback.

See also

SET COMMIT_ON_EXIT.

Description

Leave the ISQL environment and return to the operating system. This will close your connection with the database. Before doing so, ISQL will perform a COMMIT operation if the COMMIT_ON_EXIT option is ON. If the option is OFF, ISQL will perform a ROLLBACK. The default action is to COMMIT any changes you have made to the database.

EXPLAIN Statement

Syntax

```
EXPLAIN PLAN FOR CURSOR cursor-name INTO host-variable

...     | INTO host-variable                                    |
        | USING DESCRIPTOR sqlda-name                           |

cursor-name:        identifier, or host-variable
sqlda-name:         identifier
```

Purpose

To retrieve a text specification of the optimization strategy used for a particular cursor.

Usage

Embedded SQL.

Authorization

Must have opened the named cursor.

Side effects

None.

See also

DECLARE CURSOR Statement, PREPARE Statement, FETCH Statement, CLOSE Statement, OPEN Statement, "Cursors in Embedded SQL" on page 555, "The SQL communication area" on page 550.

Description

The EXPLAIN statement retrieves a text representation of the optimization strategy for the named cursor. The cursor must be previously declared and opened.

The **host-variable** or SQLDA variable must be of string type. The optimization string specifies in what order the tables are being searched and also which indexes are being used for the searches if any. This string can be quite long, but most optimization strings will fit into 300 characters.

The format of this string is, in general:

```
table (index), table (index), ...
```

If a table has been given a correlation name, the correlation name will appear instead of the table name. The order that the table names appear in the list is the order in which they will be accessed by the database engine. After each table is

a parenthesized index name. This is the index that will be used to access the table. If no index will be used (the table will be scanned sequentially) then the letters "seq" will appear for the index name. If a particular SQL SELECT statement involves subqueries, then a colon (:) will separate each subquery's optimization string. These subquery sections will appear in the order that the database engine will execute the queries.

After successful execution of the EXPLAIN command, the **sqlerrd[3]** field of the SQLCA (SQLIOESTIMATE) will be filled in with an estimate of the number of input/output operations required to fetch all rows of the query.

A discussion with quite a few examples of the optimization string can be found in "Monitoring and Improving Performance" on page 261.

Examples

```
EXEC SQL BEGIN DECLARE SECTION;
char             plan[300];
EXEC SQL END DECLARE SECTION;

EXEC SQL DECLARE employee_cursor CURSOR FOR
    SELECT empnum, empname FROM employee WHERE name like :pattern;
EXEC SQL OPEN employee_cursor;
EXEC SQL EXPLAIN PLAN FOR CURSOR employee_cursor INTO :plan;

printf( "Optimization Strategy: '%s'\n", plan );
```

FETCH Statement

Syntax

```
FETCH    | [ NEXT ]                                  | cursor-name
         | PRIOR                                     |
         | FIRST                                     |
         | LAST                                      |
         | ABSOLUTE row-count                        |
         | RELATIVE row-count                        |

...[     | INTO host-variable-list                   |  ]
         | USING DESCRIPTOR sqlda-name               |
         | INTO variable-list                        |

...      [ PURGE ] [ BLOCK n ]

...      [ FOR UPDATE ] [ ARRAY fetch-count ]
```

cursor-name:	identifier, or host-variable
sqlda-name:	identifier
host-variable-list:	may contain indicator variables
row-count:	number or host variable
fetch-count:	integer or host variable

Purpose

To reposition a cursor and then get data from it.

Usage

Embedded SQL, procedures, triggers, and batches. The following clauses are for use in Embedded SQL only:

- USING DESCRIPTOR *sqlda-name*
- INTO *host-variable-list*
- PURGE
- BLOCK n
- ARRAY fetch-count
- Use of *host-variable* in cursor-name and row-count.

The following clause is for use in procedures and triggers only:

- INTO *variable-list*

Authorization

The cursor must be opened and the user must have SELECT permission on the tables referenced in the declaration of the cursor.

Side effects

None.

See also

DECLARE CURSOR Statement, PREPARE Statement, OPEN Statement, "Cursors in Embedded SQL" on page 555, "Using cursors in procedures and triggers" on page 239.

Description

The FETCH statement retrieves one row from the named cursor. The ARRAY clause allows so-called **wide fetches**, which retrieve more than one row at a time, and which may improve performance. The cursor must have been previously opened.

One or more rows from the result of the SELECT statement is put either into the variables in the variable list or into the program data areas described by the named SQLDA. In either case, the correspondence is one to one from the select list to either the host variable list or the SQLDA descriptor array.

The INTO clause is optional. If it is not specified, then the FETCH statement positions the cursor only (see the following paragraphs).

An optional positional parameter can be specified that allows the cursor to be moved before a row is fetched. The default is NEXT which causes the cursor to be advanced one row before the row is fetched. PRIOR causes the cursor to be backed up one row before fetching.

RELATIVE positioning is used to move the cursor by a specified number of rows in either direction before fetching. A positive number indicates moving forward and a negative number indicates moving backwards. Thus, a NEXT is equivalent to RELATIVE 1 and PRIOR is equivalent to RELATIVE -1. RELATIVE 0 retrieves the same row as the last fetch statement on this cursor.

The ABSOLUTE positioning parameter is used to go to a particular row. A zero indicates the position before the first row (see "Cursors in Embedded SQL" on page 555 or "Using cursors in procedures and triggers" on page 239). A one (1) indicates the first row, and so on. Negative numbers are used to specify an absolute position from the end of the cursor. A negative one (-1) indicates the last row of the cursor. FIRST is a short form for ABSOLUTE 1. LAST is a short form for ABSOLUTE -1.

The OPEN statement initially positions the cursor before the first row (see "Cursors in Embedded SQL" on page 555 or "Using cursors in procedures and triggers" on page 239). If the fetch includes a positioning parameter and the position is outside the allowable cursor positions, then the SQLSTATE_NOTFOUND warning is issued.

Cursor positioning problems

Inserts and some updates to DYNAMIC SCROLL cursors can cause problems with cursor positioning. The database engine will not put inserted rows at a predictable position within a cursor unless there is an ORDER BY clause on the SELECT statement. In some cases, the inserted row will not appear at all until the cursor is closed and opened again. With SQL Anywhere, this occurs if a temporary table had to be created to open the cursor (see "Temporary tables used in query processing" on page 271 for a description). The UPDATE statement may cause a row to move in the cursor. This will happen if the cursor has an ORDER BY that uses an existing index (a temporary table is not created).

The FOR UPDATE clause indicates that the fetched row will subsequently be updated with an UPDATE WHERE CURRENT OF CURSOR statement. This clause causes the database engine to put a write lock on the row. The lock will be held until the end of the current transaction. See "How locking works" on page 204.

Using the FETCH statement in Embedded SQL

The DECLARE statement must appear before the FETCH statement in the C source code, and the OPEN statement must be executed before the FETCH statement. If a host variable is being used for the cursor name, then the DECLARE statement actually generates code and thus must be executed before the FETCH statement.

In the multiuser environment, rows may be fetched by the client more than one at a time. Note that in QNX, the client is linked into the application so this will always happen by default. This is referred to as block fetching or multi-row fetching. The first fetch causes several rows to be sent back from the server. The client buffers these rows and subsequent fetches are retrieved from these buffers without a new request to the server. The BLOCK clause gives the client and server a hint as to how many rows may be fetched by the application. The special value of 0 means the request will be sent to the server and a single row will be returned (no row blocking). The PURGE clause causes the client to flush its buffers of all rows and then send the fetch request to the server. Note that this fetch request may return a block of rows.

If the SQLSTATE_NOTFOUND warning is returned on the fetch, then the *sqlerrd[2]* field of the SQLCA (SQLCOUNT) will contain the number of rows that the attempted fetch exceeded the allowable cursor positions. (A cursor can be on a row, before the first row or after the last row.) The value is 0 if the row was not found but the position is valid, for example, executing FETCH RELATIVE 1 when positioned on the last row of a cursor. The value will be positive if the attempted fetch was further beyond the end of the cursor, and negative if the attempted fetch was further before the beginning of the cursor.

After successful execution of the fetch command, the *sqlerrd[1]* field of the SQLCA (SQLIOCOUNT) will be incremented by the number of input/output operations required to perform the fetch. This field is actually incremented on every database command.

To use wide fetches in Embedded SQL, include the fetch statement in your code as follows:

```
EXEC SQL FETCH . . . ARRAY nnn
```

where ARRAY nnn is the last item of the FETCH statement. The fetch count *nnn* can be a host variable. The SQLDA must contain nnn * (columns per row) variables. The first row is placed in SQLDA variables 0 to (columns per row)-1, and so on.

The engine returns in SQLCOUNT the number of records fetched and always returns a SQLCOUNT greater than zero unless there is an error. Older versions of the engine or server only return a single row and the SQLCOUNT is set to zero. Thus a SQLCOUNT of zero with no error condition indicates one valid row has been fetched.

Embedded SQL example

```
EXEC SQL DECLARE cur_employee CURSOR FOR
        SELECT emp_id, emp_lname FROM employee ;
EXEC SQL OPEN cur_employee;
EXEC SQL FETCH cur_employee
        INTO :emp_number, :emp_name:indicator;
```

For a detailed example of using wide fetches, see the section "Fetching more than one row at a time" on page 562.

Procedure/trigger example

```
BEGIN
    DECLARE cur_employee CURSOR FOR
                SELECT emp_lname
                FROM   employee ;
    DECLARE name CHAR(40) ;

    OPEN cur_employee;
    LOOP
        FETCH NEXT cur_employee into name ;
            ...
    END LOOP
    CLOSE cur_employee;
END
```

FOR Statement

Syntax

```
[ statement-label : ]
   . . .FOR  for-loop-name  AS  cursor-name
   . . .      CURSOR FOR  statement
   . . .      [ FOR UPDATE | FOR READ ONLY ]
   . . .      DO  statement-list
   . . .END  FOR  [ statement-label ]
```

Purpose

Repeat the execution of a statement list once for each row in a cursor.

Usage

Procedures, triggers, and batches only.

Authorization

None.

Side effects

None.

See also

LOOP Statement, DECLARE CURSOR Statement, FETCH Statement, LEAVE Statement.

Description

The FOR statement is a control statement that allows you to execute a list of SQL statements once for each row in a cursor. The FOR statement is equivalent to a compound statement with a DECLARE for the cursor and a DECLARE of a variable for each column in the result set of the cursor followed by a loop that fetches one row from the cursor into the local variables and executes *statement-list* once for each row in the cursor.

The name and data type of the local variables that are declared are derived from the *statement* used in the cursor. With a SELECT statement, the data type will be the data type of the expressions in the select list. The names will be the select list item aliases where they exist; otherwise, they will be the name of the columns. Any select list item that is not a simple column reference must have an alias. With a CALL statement, the names and data types will be taken from the RESULT clause in the procedure definition.

The LEAVE statement can be used to resume execution at the first statement after the END FOR. If the ending *statement-label* is specified, it must match the beginning *statement-label*.

Example

```
FOR names AS curs CURSOR FOR SELECT emp_lname FROM employee DO
    CALL search_for_name( emp_lname );
END FOR;
```

FROM Clause

Syntax

```
. . . FROM  table-expr, ...

table-expr:

        | table-spec                                                    |
        | table-expr  join-type  table-spec  [ ON condition ]           |
        | ( table-expr, ... )                                           |


table-spec:

        [userid . ] table-name [ [AS] correlation-name ]


join-type:

        | CROSS JOIN                                                    |
        | [ NATURAL | KEY ] JOIN                                        |
        | [ NATURAL | KEY ] INNER JOIN                                  |
        | [ NATURAL | KEY ] LEFT OUTER JOIN                             |
        | [ NATURAL | KEY ] RIGHT OUTER JOIN                            |
```

Purpose

To specify the database tables or views involved in a SELECT or UPDATE statement.

Usage

Anywhere.

Authorization

Must be connected to the database.

Side effects

None.

See also

SELECT Statement, UPDATE Statement, DELETE Statement, Search conditions.

Description

The SELECT and UPDATE commands require a table list to specify which tables will be used by the command.

Note
Although this description refers to tables, it applies to views unless otherwise noted.

The FROM table list creates a result set consisting of all the columns from all the tables specified. Initially, all combinations of rows in the component tables are in the result set, and the number of combinations is usually reduced by *join* conditions and/or WHERE conditions.

Tables owned by a different user can be qualified by specifying the *user ID*. Tables owned by groups to which the current user belongs will be found by default without specifying the user ID (see "Referring to tables owned by groups" on page 321).

The *correlation name* is used to give a temporary name to the table for this SQL command only. This is useful when referencing columns which must be qualified by a table name but the table name is long and cumbersome to type. The correlation name is also necessary to distinguish between table instances when referencing the same table more than once in the same query. If no correlation name is specified, then the table name is used as the correlation name for the current command.

If the same correlation name is used twice for the same table in a table expression, that table is treated as if it were only listed once. For example, in

```
SELECT  *
FROM    sales_order
        KEY JOIN sales_order_items,
        sales_order
        KEY JOIN employee
```

the two instances of the **sales_order** table are treated as one instance, and is equivalent to

```
SELECT  *
FROM    sales_order_items
        KEY JOIN sales_order
        KEY JOIN employee
```

whereas

```
SELECT  *
FROM    Person HUSBAND, Person WIFE
```

would be treated as two instances of the Person table, with different correlation names HUSBAND and WIFE.

Joining tables

A JOIN reduces the result set based on the *join type* and *join condition*. The join types are described below and the join condition is specified after the keyword ON.

Parentheses can also be used to join one table to more than one other table. For example,

```
A  JOIN  (B,C)
```

joins table A to both tables B and C.

Table expressions can be arbitrarily complex. For example,

```
A  JOIN  B  JOIN  C

A  JOIN  ( B, C  JOIN  D )
```

are legal and meaningful table list expressions. (Any of the valid join types could have been used in the above examples.)

Cross joins

A CROSS JOIN does not restrict the results of the join. The query

```
SELECT  *
FROM    table1
        CROSS JOIN table2
```

has a result set as follows:

- As long as **table1** is not the same name as **table2**:
  - The columns of the result set includes all columns in **table1** and also all columns in **table2**.
  - There is one row in the result set for each combination of a row in **table1** and a row in **table2**. If **table1** has `n1` rows and **table2** has `n2` rows, the query returns `n1 * n2` rows.
- If **table1** is the same table as **table2**, and neither is given a correlation name, the result set is simply the rows of **table1**.

Generated join conditions

Natural joins and key joins are **generated join conditions**: that is, the keyword KEY or NATURAL indicates a restriction on the join results.

For a natural join, the generated join condition is based on the names of columns in the tables being joined; for a key join, the condition is based on a foreign key relationship between the two tables.

Natural joins

A NATURAL JOIN restricts the results by comparing the values of columns in the two tables with the same column name. An error is reported if there are no common column names. A join condition can optionally be specified which further restricts the results of the join.

Column names such as Description or Address often cause a NATURAL JOIN to return different results than expected.

Key joins

A KEY JOIN restricts the result set based on a foreign key relationship between the two tables. (This was the only type of automatic join supported in Watcom SQL Version 3.0.) A join condition can optionally be specified which further restricts the results of the join. A key join is valid if exactly one foreign key is identified between the two tables; otherwise, an error indicating the ambiguity is reported.

Parentheses can also be used to join one table to more than one other table. For example

```
A  KEY JOIN  (B,C)
```

joins table A to both tables B and C.

A join involving parentheses is valid if there is an unambiguous join for each table listed in the parentheses.

A KEY JOIN with a view is valid if there is a valid KEY JOIN with exactly one of the tables in the FROM table list of the view definition.

There are two conditions where the meaning of a KEY JOIN is ambiguous. In the first condition, there are two tables A and B where A has a foreign key for B and B has a foreign key for A. The second condition occurs when a table A has two foreign keys for a table B. In either case, the primary table *must* have a correlation name which is the same as the *role name* of the foreign key. Otherwise an error will be reported. For example, every SQL Anywhere database has a table called **SYSTABLEPERM** to hold permission information. Every row in the permission table has two foreign keys for the table describing user IDs (**SYSUSERPERM**). One foreign key is for the user ID giving the permission (role name **grantor**) and the other is the user ID getting the permission (role name **grantee**). A KEY JOIN for the **grantor** would look like the following:

```
SYSUSERPERM grantor
            KEY JOIN  SYSTABLEPERM
```

and a KEY JOIN for both would look like:

```
SYSUSERPERM grantor
            KEY JOIN  SYSTABLEPERM
            KEY JOIN  SYSUSERPERM grantee
```

INNER JOIN and OUTER JOIN

All joins described thus far have been INNER JOINs, which is the default. Each row of

```
customer INNER JOIN sales_order
```

contains the information from one **customer** row and one **sales_order** row. If a particular customer has placed no orders, there will be no information for that customer.

```
A LEFT OUTER JOIN B
```

includes all rows of table A whether or not there is a row in B that satisfies the join condition. If a particular row in A has no matching row in B, the columns in the join corresponding to table B will contain the NULL value. Similarly,

```
A RIGHT OUTER JOIN B
```

includes all rows of table B whether or not there is a row in A that satisfies the join condition.

Join conditions

A *join condition* can be specified for any join type except CROSS JOIN. The simplest form of join condition is to use it instead of using a KEY or NATURAL JOIN. In the sample database, the following are equivalent:

```
SELECT  *
FROM    sales_order
        JOIN customer
  ON    sales_order.cust_id = customer.id

SELECT  *
FROM    sales_order
        KEY JOIN customer
```

The following two are also equivalent:

```
SELECT   *
FROM     department
         JOIN employee
  ON     department.dept_id = employee.dept_id

SELECT   *
FROM     department
         NATURAL JOIN employee
```

With INNER JOINS, specifying a join condition is equivalent to adding the join condition to the WHERE clause. However, this is not true for OUTER JOINS.

A join condition on an OUTER JOIN is part of the join operation. For example, the statement:

```
SELECT   *
FROM     employee
         KEY LEFT OUTER JOIN Skill
  ON     skill_name = 'COBOL'
```

produces a table containing all employees whether or not they have the skill COBOL. Those who do not have the skill COBOL will have the NULL value in the **Skill** columns. On the other hand, the query:

```
SELECT   *
FROM     employee
         KEY LEFT OUTER JOIN Skill
WHERE    skill_name = 'COBOL'
```

can be thought of as two separate stages.

1. The first stage creates the table specified by the FROM clause:

   ```
   SELECT   *
   FROM     employee
            KEY LEFT OUTER JOIN Skill
   ```

   which has at least one row for each employee. Employees who do not have the skill COBOL will have the NULL value in the **Skill** table columns. For all other employees, there will be one row for each of their skills.
2. The second stage takes this result and applies the condition:

   ```
   WHERE skill_name = 'COBOL'
   ```

   which removes employees without any skills (since the condition is UNKNOWN), and only keeps the skill COBOL for the other employees. Thus, the result has no rows with the NULL value in the **Skill** columns, so it is clearly different from the result achieved using the join condition.

OUTER JOINS with join conditions can be complicated. If you are having problems using a join condition, try removing the WHERE clause from the statement to verify that the join is retrieving the rows you expected.

Join abbreviations

SQL Anywhere provides the following abbreviations for joins.

table1 [INNER | LEFT OUTER | RIGHT OUTER] JOIN table2

If there is no ON *join condition* specified, the join is assumed to be a KEY join. All key joins are a SQL Anywhere extension. Note that KEY JOIN is *not* assumed if a join condition is specified.

table1 JOIN table2 ON join-condition

The default join type is an INNER JOIN.

Examples

```
FROM employee

FROM employee NATURAL JOIN department

FROM customer
     KEY JOIN sales_order
     KEY JOIN sales_order_items
     KEY JOIN product
```

GET DATA Statement

Syntax

```
GET DATA cursor-name COLUMN column-num OFFSET start-offset

. . .   [ WITH TEXTPTR ]

. . .   | USING DESCRIPTOR sqlda-name                    |
        | INTO host-variable [, . . . ]                  |
```

cursor-name:	identifier, or host-variable
column-num:	integer or host-variable
start-offset:	integer or host-variable
sqlda-name:	identifier

Purpose

To get string or binary data for one column on the current row of a cursor. GET DATA is usually used to fetch LONG BINARY or LONG VARCHAR fields. See "SET Statement" on page 1008 for putting long values into the database.

Usage

Embedded SQL.

Authorization

The cursor must be opened and positioned on a row using FETCH.

Side effects

None.

See also

FETCH Statement, "Transact-SQL READTEXT statement" on page 485, "Transact-SQL WRITETEXT statement" on page 490.

Description

Get a piece of one column value from the row at the current cursor position. The value of *column-num* starts at one and identifies which column's data is to be fetched. That column must be of a string or binary type.

The *start-offset* indicates the number of bytes to skip over in the field value. Normally, this would be the number of bytes previously fetched. The number of bytes fetched on this GET DATA command is determined by the length of the target host variable.

The indicator value for the target host variable is a short integer, so it cannot always contain the number of bytes truncated. Instead, it contains a negative value if the field contains the NULL value, and a positive value (NOT

necessarily the number of bytes truncated) if the value is truncated, and zero if a non-NULL value is not truncated.

If the WITH TEXTPTR clause is given, a text pointer is retrieved into a second indicator variable or into the second field in the SQLDA. This text pointer can be used with the &tsq. READ TEXT and WRITE TEXT statements.

The total length of the data is returned in the SQLCOUNT field of the SQLCA structure.

Example

The following example uses GET DATA to fetch a **binary large object** (often called a **blob**).

```
EXEC SQL BEGIN DECLARE SECTION;
DECL_BINARY(1000)   piece;
short               ind;
EXEC SQL END DECLARE SECTION;
long                offset;
int                 size;

/* Open a cursor on a long varchar field */
EXEC SQL DECLARE big_cursor CURSOR FOR
    SELECT long_data FROM some_table
     WHERE key_id = 2;
EXEC SQL OPEN big_cursor;
EXEC SQL FETCH big_cursor INTO :buffer;
FOR( offset = 0; ; offset += piece.len ) {
    EXEC SQL GET DATA big_cursor COLUMN 1
        OFFSET :offset INTO :piece:ind;
    /* Done if the NULL value */
    if( ind < 0 ) break;
    write_out_piece( &piece );
    /* Done when the piece was not truncated */
    if( ind == 0 ) break;
}
EXEC SQL CLOSE big_cursor;
```

GET OPTION Statement

Syntax

```
GET OPTION [ userid. ] option-name

   ...    | INTO host-variable                                    |
          | USING DESCRIPTOR sqlda-name                           |
```

userid:	identifier, string or host-variable
sqlda-name:	identifier
option-name:	identifier, string or host-variable
host-variable:	indicator variable allowed

Purpose

To find the current setting of an option.

Usage

Embedded SQL.

Authorization

Must have DBA authority to get someone else's option settings.

Side effects

None.

See also

SET OPTION Statement.

Description

The GET OPTION command gets the option setting of the named option for **userid** or for the connected user if **userid** is not specified. This will be either the user's personal setting or the **PUBLIC** setting if there is no setting for the connected user. If the option specified is a database option and the user has a temporary setting for that option, then the temporary setting will be retrieved.

Example

```
EXEC SQL GET OPTION 'date_format' INTO :datefmt;
```

GRANT Statement

Syntax

Format 1

```
GRANT  CONNECT TO  userid,...  IDENTIFIED  BY  password,...
```

Format 2

```
              | DBA                                          |
GRANT         | RESOURCE                                     |
              | GROUP                                        |
              | MEMBERSHIP IN GROUP userid,...               |

  . . .  TO  userid,...
```

Format 3

```
              | ALTER                                        |
              | DELETE                                       |
              | INSERT                                       |
GRANT         | REFERENCES                                   | ,...
              | SELECT                                       |
              | UPDATE  [(column-name,...)]                  |
              | ALL  [PRIVILEGES]                            |

  . . .  ON  [creator.]table-name  TO  userid, ...  [WITH GRANT OPTION]
```

Format 4:

```
GRANT  EXECUTE  ON  [creator.]procedure-name  TO  userid,...
```

Purpose

To give permissions to specific users.
To create new user IDs.

Usage

Anywhere.

Authorization

For Format 1 or 2, must:

- Be changing your own password using GRANT CONNECT
- Be adding members to your own user ID, or
- Have DBA authority

For Format 3, must have:

- Created the table, or
- Been granted permissions on the table with GRANT OPTION, or
- DBA authority

For Format 4, must have:

- Created the procedure, or
- DBA authority

Side effects

Automatic commit.

See also

REVOKE Statement.

Description

The GRANT command is used to grant database permissions to individual user IDs and groups. It is also used to create and delete users and groups.

Formats 1 and 2 of the GRANT command are used for granting special privileges to users as follows:

CONNECT Creates a new user. GRANT CONNECT can also be used by any user to change their own password. To create a user with the empty string as the password, type:

```
GRANT CONNECT TO userid IDENTIFIED BY ""
```

To create a user with no password, type:

```
GRANT CONNECT TO userid
```

A user with no password cannot connect to the database. This is useful when creating groups when you do not want anyone to connect to the group user ID. The password must be a valid identifier, as described in "Watcom-SQL language elements" on page 752.

DBA Database Administrator authority gives a user permission to do anything. This is usually reserved for the person in the organization who is looking after the database.

RESOURCE Allows the user to create tables and views.

GROUP Allows the user(s) to have members. See "Managing groups" on page 318 for a complete description.

MEMBERSHIP IN GROUP userid,...
This allows the user(s) to inherit table permissions from a group and to reference tables created by the group without qualifying the table name. See "Managing groups" on page 318 for a complete description.

Format 3 of the GRANT command is used to grant permission on individual tables or views. The table permissions can be listed together, or specifying ALL grants all six permissions at once. The permissions have the following meaning:

ALTER The users will be allowed to alter this table with the ALTER TABLE command. This permission is not allowed for views.

DELETE The users will be allowed to delete rows from this table or view.

INSERT The users will be allowed to insert rows into the named table or view.

REFERENCES The users will be allowed to create indexes on the named tables, and foreign keys which reference the named tables.

SELECT The users will be allowed to look at information in this view or table.

UPDATE [(column-name,...)] The users will be allowed to update rows in this view or table. If column names are specified, then the users will be allowed to update only those columns. UPDATE permissions on columns cannot be granted for views, only for tables.

ALL All of the above permissions.

INDEX and ALTER permissions are not allowed on views. If columns are specified for an UPDATE permission, then UPDATE permission will be given on only those columns.

If WITH GRANT OPTION is specified, then the named user ID is also given permission to GRANT the same permissions to other user IDs.

DBA authority is required to grant permissions on views.

Format 4 of the GRANT command is used to grant permission to execute a procedure.

Examples

Make two new users for the database.

```
GRANT
CONNECT     TO Laurel, Hardy
IDENTIFIED BY Stan, Ollie
```

Grant permissions on the employee table to user Laurel

```
GRANT
SELECT, UPDATE ( street )
ON      employee
TO      Laurel
```

Allow the user Hardy to execute the Calculate_Report procedure.

```
GRANT
EXECUTE ON Calculate_Report
TO      Hardy
```

GRANT CONSOLIDATE Statement

Syntax

```
GRANT CONSOLIDATE TO userid, . . .

    . . .    TYPE message-system, . . .

    . . .    ADDRESS address-string, . . .

message-system:

        |MAPI    |
        |FILE    |
        |other   |*
```

* A VIM link and an SMTP/POP link are expected to be available in the first quarter of 1996.

Purpose

To identify the database immediately above the current database in a SQL Remote hierarchy, who will receive messages from the current database.

Usage

Anywhere.

Authorization

Must have DBA authority.

Side effects

Automatic commit.

See also

GRANT REMOTE Statement, REVOKE CONSOLIDATE Statement, GRANT PUBLISH Statement.

Description

In a SQL Remote installation, the database immediately above the current database in a SQL Remote hierarchy must be granted CONSOLIDATE permissions. GRANT CONSOLIDATE is issued at a remote database to identify its consolidated database. Each database can have only one user ID with CONSOLIDATE permissions: you cannot have a database that is a remote database for more than one consolidated database.

The consolidated user is identified by a message system, identifying the method by which messages are sent to and received from the consolidated user. The currently supported message systems are MAPI and FILE. The VIM system is

expected to be in beta by the time of release. The address-name must be a valid address for the message-system, enclosed in single quotes.

For the FILE message type, the address is a subdirectory of the directory pointed to by the SQLREMOTE environment variable.

The GRANT CONSOLIDATE statement is required for the consolidated database to receive messages, but does not by itself subscribe the consolidated database to any data. To subscribe to data, a subscription must be created for the consolidated user ID to one of the publications in the current database. Running the database extraction utility at a consolidated database creates a remote database with the proper GRANT CONSOLIDATE statement aready issued.

Example

```
GRANT CONSOLIDATE TO con_db
TYPE    mapi
ADDRESS 'Consolidated Database'
```

GRANT PUBLISH Statement

Syntax

GRANT PUBLISH TO userid

Purpose

To identify the publisher of the current database.

Usage

Anywhere.

Authorization

Must have DBA authority.

Side effects

Automatic commit.

See also

GRANT REMOTE Statement, GRANT CONSOLIDATE Statement, REVOKE PUBLISH Statement, CREATE PUBLICATION Statement, CREATE SUBSCRIPTION Statement.

Description

Each database in a SQL Remote installation is identified in outgoing messages by a user ID, called the **publisher**. The GRANT PUBLISH statement identifies the publisher user ID associated with these outgoing messages.

Only one user ID can have PUBLISH authority. The user ID with PUBLISH authority is identified by the special constant CURRENT PUBLISHER. The following query identifies the current publisher:

```
SELECT CURRENT PUBLISHER
```

If there is no publisher, the special constant is NULL.

The current publisher special constant can be used as a default setting for columns. It is often useful to have a CURRENT PUBLISHER column as part of the primary key for replicating tables, as this helps prevent primary key conflicts due to updates at more than one site.

In order to change the publisher, you must first drop the current publisher using the REVOKE PUBLISH statement, and then create a new publisher using the GRANT PUBLISH statement.

Example

```
GRANT PUBLISH TO publisher_ID
```

GRANT REMOTE Statement

Syntax

GRANT REMOTE TO userid, . . .

. . . TYPE message-system, . . .

. . . ADDRESS address-string, . . .

message-system:

|MAPI |
|FILE |
|other |*

* A VIM link and an SMTP/POP link are expected to be available in the first quarter of 1996.

Purpose

To identify a database immediately below the current database in a SQL Remote hierarchy, who will receive messages from the current database. These are called remote users.

Usage

Anywhere.

Authorization

Must have DBA authority.

Side effects

Automatic commit.

See also

GRANT CONSOLIDATE Statement, REVOKE REMOTE Statement, GRANT PUBLISH Statement.

Description

In a SQL Remote installation, each database receiving messages from the current database must be granted REMOTE permissions.

The single exception is the database immediately above the current database in a SQL Remote hierarchy, which must be granted CONSOLIDATE permissions.

The remote user is identified by a message system, identifying the method by which messages are sent to and received from the consolidated user. The currently supported message systems are MAPI and FILE. The VIM message

system is expected to be in beta at the time of release. The address-name must be a valid address for the message-system, enclosed in single quotes.

For the FILE message type, the address is a subdirectory of the directory pointed to by the SQLREMOTE environment variable.

The GRANT REMOTE statement is required for the remote database to receive messages, but does not by itself subscribe the remote user to any data. To subscribe to data, a subscription must be created for the user ID to one of the publications in the current database, using the database extraction utility or the CREATE SUBSCRIPTION statement.

Example

```
GRANT REMOTE TO SamS
TYPE    mapi
ADDRESS 'Singer, Samuel'
```

HELP Statement

Syntax

HELP [topic]

Purpose

To receive help in the ISQL environment.

Usage

ISQL.

Authorization

None.

Side effects

None.

Description

The HELP command is used to enter the ISQL interactive help facility. The **topic** for help can be optionally specified. If **topic** is not specified, then the help system is entered at the index.

IF Statement

Syntax

```
        IF search-condition THEN statement-list
...     [ ELSEIF search-condition THEN statement-list ] ...
...     [ ELSE statement-list ]
...     END IF
```

Purpose

Provide conditional execution of SQL statements.

Usage

Procedures, triggers, and batches only.

Authorization

None.

Side effects

None.

See also

The chapter "Using Procedures, Triggers, and Batches" on page 215, Compound statements.

Description

The IF statement is a control statement that allows you to conditionally execute the first list of SQL statements whose *search-condition* evaluates to TRUE. If no *search-condition* evaluates to TRUE, and an ELSE clause exists, the *statement-list* in the ELSE clause is executed. Execution resumes at the first statement after the END IF.

Example

```
CREATE PROCEDURE TopCustomer (OUT TopCompany CHAR(35), OUT TopValue
INT)
BEGIN
    DECLARE err_notfound EXCEPTION FOR SQLSTATE '02000' ;
    DECLARE curThisCust CURSOR FOR
    SELECT company_name, CAST( sum(sales_order_items.quantity *
product.unit_price) AS INTEGER) VALUE
        FROM   customer
           LEFT OUTER JOIN sales_order
           LEFT OUTER JOIN sales_order_items
           LEFT OUTER JOIN product
        GROUP BY company_name ;
    DECLARE ThisValue INT ;
    DECLARE ThisCompany CHAR(35) ;
    SET TopValue = 0 ;
    OPEN curThisCust ;
    CustomerLoop:
        LOOP
            FETCH NEXT curThisCust INTO ThisCompany, ThisValue ;
            IF SQLSTATE = err_notfound THEN
                LEAVE CustomerLoop ;
            END IF ;
            IF ThisValue > TopValue THEN
                SET TopValue = ThisValue ;
                SET TopCompany = ThisCompany ;
            END IF ;
        END LOOP CustomerLoop ;
    CLOSE curThisCust ;
END
```

INCLUDE Statement

Syntax

INCLUDE filename

filename: identifier

Purpose

Include a file into a source program to be scanned by the SQL source language preprocessor.

Usage

Embedded SQL.

Authorization

None.

Side effects

None.

See also

SQLDA, SQLCA.

Description

The INCLUDE statement is very much like the C preprocessor **#include** directive. However, the SQL preprocessor will read the given file inserting its contents into the output C file. Thus, if an include file contains information that the SQL preprocessor requires, it should be included with the Embedded SQL INCLUDE command.

Two file names are specially recognized: SQLCA and SQLDA. Any C program using Embedded SQL must contain an

```
EXEC SQL INCLUDE SQLCA;
```

statement before any Embedded SQL statements. This statement must appear at a position in the C program where static variable declarations are allowed. Many Embedded SQL statements require variables (invisible to the programmer) which are declared by the SQL preprocessor at the position of the SQLCA include statement. The SQLDA file must be included if any SQLDA's are used.

INPUT Statement

Syntax

```
INPUT INTO [creator.]table-name
    . . . [ FROM filename | PROMPT ]
    . . . [ FORMAT input-format ]
    . . . [ BY ORDER | BY NAME ]
    . . . [ DELIMITED BY string ]
    . . . [ COLUMN WIDTHS (integer,...) ]
    . . . [ NOSTRIP ]
    . . . [ ( column-name, ... ) ]
```

Purpose

To import data into a database table from an external file or from the keyboard.

Usage

ISQL.

Authorization

Must have INSERT permission on the table or view.

Side effects

None.

See also

OUTPUT Statement, INSERT Statement, UPDATE Statement, DELETE Statement, SET OPTION Statement.

Description

The INPUT command allows efficient mass insertion into a database table. Lines of input are read either from the user via an input window (if PROMPT is specified) or from a file (if FROM filename is specified). If neither is specified, the input will be read from the command file containing the input command—this can even be directly from the ISQL editor. In this case, input is ended with a line containing only the string END.

These lines are inserted into the named table. If a column list is specified, the data is inserted into the specified columns of the named table.

Normally, the INPUT command stops when it attempts to insert a row that causes an error. Errors can be treated in different ways by setting the

ON_ERROR and CONVERSION_ERROR options (see "SET OPTION Statement" on page 989). ISQL will print a warning in the statistics window if any string values are truncated on INPUT. Missing values for NOT NULL columns will be set to zero for numeric types and to the empty string for non-numeric types.

The BY clause allows the user to specify whether the columns from the input file should be matched up with the table columns based on their ordinal position in the lists (ORDER, the default) or by their names (NAME). Not all input formats have column name information in the file. NAME is allowed only for those formats that do. They are the same formats that allow automatic table creation listed below: DBASEII, DBASEIII, DIF, FOXPRO, LOTUS, and WATFILE.

The DELIMITED BY clause allows you to specify a string to be used as the delimiter in ASCII input format.

COLUMN WIDTHS can be specified for FIXED format only. It specifies the widths of the columns in the input file. If COLUMN WIDTHS is not specified, then the widths are determined by the database column types.

Normally, for ASCII input format, trailing blanks will be stripped before the value is inserted. NOSTRIP can be used to suppress trailing blank stripping.

Each set of values must occupy one input line and must be in the format specified by the FORMAT clause or the format set by the SET INPUT_FORMAT command if the FORMAT clause is not specified. When input is entered by the user, an empty screen is provided for the user to enter one row per line in the input format.

Certain file formats contain information about column names and types. Using this information, the INPUT command will create the database table if it does not already exist. This is a very easy way to load data into the database. The formats that have enough information to create the table are: DBASEII, DBASEIII, DIF, FOXPRO, LOTUS, and WATFILE.

Allowable input formats are:

ASCII Input lines are assumed to be ASCII characters, one row per line, with values separated by commas. Alphabetic strings may be enclosed in apostrophes (single quotes) or quotation marks (double quotes). Strings containing commas must be enclosed in either single or double quotes. If single or double quotes are used, double the quote character to use it within the string. Optionally, you can use the DELIMITED BY clause to specify a different delimiter string than the default which is a comma.

Three other special sequences are also recognized. The two characters \n represent a newline character, \\ represents a single \,

and the sequence \xDD represents the character with hexadecimal code DD.

DBASE The file is in dBASE II or dBASE III format. ISQL will attempt to determine which of the two DBase formats the file is based on information in the file. If the table doesn't exist, it will be created.

DBASEII The file is in dBASE II format. If the table doesn't exist, it will be created.

DBASEIII The file is in dBASE III format. If the table doesn't exist, it will be created.

DIF Input file is in Data Interchange Format. If the table doesn't exist, it will be created.

FIXED Input lines are in fixed format. The width of the columns can be specified using the COLUMN WIDTHS clause. If they are not specified, then column widths in the file must be the same as the maximum number of characters required by any value of the corresponding database column's type.

FOXPRO The file is in FoxPro format (the FoxPro memo field is different than the dBASE memo field). If the table doesn't exist, it will be created.

LOTUS The file is a Lotus WKS format worksheet. INPUT assumes that the first row in the Lotus WKS format worksheet is column names. If the table doesn't exist, it will be created. In this case, the types and sizes of the columns created may not be correct because the information in the file pertains to a cell, not to a column.

WATFILE The input will be a WATFILE file. If the table doesn't exist, it will be created. WATFILE is a tabular file management tool available from WATCOM.

Input from a command file is terminated by a line containing END. Input from a file is terminated at the end of the file.

Example

```
INPUT   INTO employee
FROM    new_emp.inp
FORMAT  ascii;
```

INSERT Statement

Syntax

Format 1

```
INSERT INTO [creator.]table-name [( column-name, ... )]
    ... VALUES ( expression | DEFAULT, ... )
```

Format 2

```
INSERT INTO [creator.]table-name [( column-name, ... )]
    ... select-statement
```

Purpose

To insert a single row (format 1) or a selection of rows from elsewhere in the database (format 2) into a table.

Usage

Anywhere.

Authorization

Must have INSERT permission on the table.

Side effects

None.

See also

INPUT Statement, UPDATE Statement, DELETE Statement, PUT Statement.

Description

The INSERT command is used to add new rows to a database table.

Format 1 allows the insertion of a single row with the specified expression values. The keyword DEFAULT can be used to cause the default value for the column to be inserted. If the optional list of column names is given, the values are inserted one for one into the specified columns. If the list of column names is not specified, the values are inserted into the table columns in the order they were created (the same order as retrieved with SELECT *). The row is inserted into the table at an arbitrary position. (In relational databases, tables are not ordered.)

Format 2 allows the user to do mass insertion into a table with the results of a fully general SELECT statement. Insertions are done in an arbitrary order unless the SELECT statement contains an ORDER BY clause. The columns from the

select list are matched ordinally with the columns specified in the column list or the columns in the order they were created.

Note
The NUMBER(*) function is useful for generating primary keys with format 2 of the INSERT statement (see "Functions" on page 765).

Inserts can be done into views provided the SELECT statement defining the view has only one table in the FROM clause and does not contain a GROUP BY clause, an aggregate function, or involve a UNION operation.

Character strings inserted into tables are always stored in the case they are entered, regardless of whether the database is case sensitive or not. Thus a string **Value** inserted into a table is always held in the database with an upper-case V and the remainder of the letters lower case. SELECT statements return the string as **Value**. If the database is not case-sensitive, however, all comparisons make **Value** the same as **value**, **VALUE**, and so on. Further, if a single-column primary key already contains an entry **Value**, an INSERT of **value** is rejected, as it would make the primary key not unique.

Performance hint
To insert many rows into a table, it is more efficient to declare a cursor and use the PUT statement to insert the rows, where possible, than to carry out many separate INSERT statements.

Examples

Add an Eastern Sales department to the database.

```
INSERT
INTO   department ( dept_id, dept_name )
VALUES ( 230, 'Eastern Sales' )
```

Fill the table dept_head with the names of department heads and their departments.

```
INSERT
INTO   dept_head (name, dept)
       SELECT emp_fname || ' ' || emp_fname AS name,
              dept_name
       FROM   employee JOIN department
              ON emp_id = dept_head_id
```

LEAVE Statement

Syntax

LEAVE statement-label

Purpose

Continue execution by leaving a compound statement or LOOP.

Usage

Procedures, triggers, and batches only.

Authorization

None.

Side effects

None.

See also

The chapter "Using Procedures, Triggers, and Batches" on page 215, Compound statements, LOOP Statement, FOR Statement.

Description

The LEAVE statement is a control statement that allows you to leave a labeled compound statement or a labelled loop. Execution resumes at the first statement after the compound statement or loop.

The compound statement that is the body of a procedure or triggers has an implicit label that is the same as the name of the procedure or trigger.

Examples

The following fragment shows how the LEAVE statement is used to leave a loop.

```
SET i = 1;
lbl:
    LOOP
        INSERT
        INTO Counters ( number )
        VALUES ( i ) ;
        IF i >= 10 THEN
            LEAVE lbl ;
        END IF ;
        SET i = i + 1
    END LOOP lbl
```

The following example fragment uses LEAVE in a nested loop.

```
outer_loop:
    LOOP
        SET i = 1;
        inner_loop:
            LOOP
                . . .
                SET i = i + 1;
                IF i >= 10 THEN
                    LEAVE outer_loop
                END IF
            END LOOP inner_loop
        END LOOP outer_loop
```

LOAD TABLE Statement

Syntax

```
LOAD [ INTO ] TABLE [ creator ].table-name
   ...    FROM 'filename-string'
   ...    [ FORMAT 'ascii'] [ DELIMITED BY string ]
   ...    [ STRIP { on | off } ] [ QUOTES { on | off } ]
   ...    [ ESCAPES { on | off } ]
```

Purpose

To import data into a database table from an external ascii-format file.

Usage

Anywhere.

Authorization

Must have INSERT permission on the table.

Side effects

Triggers, including referential integrity actions, are not fired by the LOAD TABLE statement. A COMMIT is performed at the end of the load.

See also

UNLOAD TABLE Statement, INPUT Statement.

Description

The LOAD TABLE statement allows efficient mass insertion into a database table from an ascii file. LOAD TABLE is more efficient than the ISQL statement INPUT and can be called from any client application.

LOAD TABLE places an exclusive lock on the whole table; it does not fire any triggers associated with the table.

You can use LOAD TABLE on a temporary table, but the temporary table must have been declared with the ON COMMIT PRESERVE ROWS clause, as LOAD TABLE does a COMMIT after the load.

The following list describes each of the clauses of the statement.

file-name string

The filename-string is passed to the engine as a string. The string is therefore subject to the same formatting requirements as other SQL strings. In particular:

- To indicate directory paths, the backslash character \ must be represented by two backslashes. Therefore, the statement to load data from the file **c:\temp\input.dat** into the employee table is:

```
LOAD TABLE employee FROM 'c:\\temp\\input.dat' ...
```

- The pathname is relative to the database engine, not to the client application. If you are running the statement on a database server on some other computer, the directory names refers to directories on the server machine, not on the client machine.
- You can use UNC path names to load data from files on computers other than the server. For example, on a Windows for Workgroups or Windows NT network you may use the following statement to load data from a file on the client machine:

```
LOAD TABLE employee FROM '\\\\client\\temp\\input.dat'
```

FORMAT option

The only file format currently supported is ASCII. Input lines are assumed to be ASCII characters, one row per line, with values separated by the column delimiter character.

DELIMITED BY option

The default column delimiter character is a comma. You can specify an alternative column delimiter by providing a string. Only the first ASCII character of the string is read. The same formatting requirements apply as to other SQL strings. In particular, to specify tab-delimited values the hexadecimal ASCII code of the tab character (9) is used. The DELIMITED BY clause is as follows:

```
. . .DELIMITED BY '\x09' . . .
```

STRIP option With STRIP turned on (the default), trailing blanks are stripped from values before they are inserted. To turn the STRIP option off, the clause is as follows:

```
. . .STRIP OFF . . .
```

QUOTES option

With QUOTES turned on (the default), the LOAD statement looks for a quote character. The quote character is either an apostrophe (single quote) or a quotation mark (double quote). The first such character encountered in the input file is treated as the quote character for the input file.

With quotes on, column delimiter characters can be included in column values. Also, quote characters are assumed not to be part of the value. Therefore, a line of the form

```
'123 High Street, Anytown',(715)398-2354
```

is treated as two values, not three, despite the presence of the comma in the address. Also, the quotes surrounding the address are not inserted into the database.

To include a quote character in a value, with QUOTES on, you must use two quotes. The following line includes a value in the third column that is a single quote character:

```
'123 High Street, Anytown','(715)398-2354',''''
```

ESCAPES option

With ESCAPES turned on (the default), characters following the backslash character are recognized and interpreted as special characters by the database engine. New line characters can be included as the combination \n, other characters can be included in data as hexadecimal ASCII codes, such as \x09 for the tab character. A sequence of two backslash characters (\\ ) is interpreted as a single backslash.

LOOP Statement

Syntax

```
[ statement-label : ]
   . . .[ WHILE  search-condition ] LOOP
   . . .      statement-list
   . . .END  LOOP  [ statement-label ]
```

Purpose

Repeat the execution of a statement list.

Usage

Procedures, triggers, and batches only.

Authorization

None.

Side effects

None.

See also

FOR Statement, LEAVE Statement.

Description

The WHILE and LOOP statements are control statements that allow you to repeatedly execute a list of SQL statements while a *search-condition* evaluates to TRUE. The LEAVE statement can be used to resume execution at the first statement after the END LOOP.

If the ending *statement-label* is specified, it must match the beginning *statement-label*.

Examples

A While loop in a procedure.

```
 . . .
SET i = 1 ;
WHILE i <= 10 LOOP
    INSERT INTO Counters( number ) VALUES ( i ) ;
    SET i = i + 1 ;
END LOOP ;
 . . .
```

A labeled loop in a procedure.

```
SET i = 1;
lbl:
    LOOP
        INSERT
        INTO Counters( number )
        VALUES ( i ) ;
        IF i >= 10 THEN
            LEAVE lbl ;
        END IF ;
        SET i = i + 1 ;
    END LOOP lbl
```

MESSAGE Statement

Syntax

```
MESSAGE expr
```

Purpose

To display a message on the message window of the database engine or server.

Usage

Procedures, triggers, and batches only.

Authorization

Must be connected to the database.

Side effects

None.

See also

CREATE PROCEDURE Statement.

Description

The MESSAGE statement displays an expression on the message window of the database engine or server (not on the client window). It is used primarily for debugging of procedures and triggers.

Example

The following procedure displays a message on the engine message window:

```
CREATE PROCEDURE message_test ()
BEGIN
    MESSAGE 'procedure called successfully' ;
END
```

The statement

```
CALL message_test()
```

displays the string *procedure called successfully* on the database engine message window.

NULL value

Syntax

NULL

Purpose

To specify a value that is unknown or not applicable.

Usage

Anywhere.

Authorization

Must be connected to the database.

Side effects

None.

See also

Expressions, Search conditions.

Description

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

missing the field does have a value, but that value is unknown, or

inapplicable the field does not apply for this particular row.

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

The NULL value introduces the concept of three valued logic to SQL. The NULL value compared using any comparison operator with any value including the NULL value is "UNKNOWN." The only search condition that will return "TRUE" is the **IS NULL** predicate. In SQL, rows are selected only if the search condition in the WHERE clause evaluates to TRUE; rows that evaluate to UNKNOWN or FALSE are *not* selected. This IS [NOT] <truth-value> clause where truth-value is one of TRUE, FALSE or UNKNOWN can also be used to select rows where the NULL value is involved. See "Search conditions" on page 803 for a description of this clause.

In the following examples, the column **Salary** contains the NULL value.

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

Figure 19. *Comparison with NULL*

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

The NULL value also has an interesting property when used in numeric expressions. The result of *any* numeric expression involving the NULL value is the NULL value. This means that if the NULL value is added to a number, the result is the NULL value—not a number. If you want the NULL value to be treated as 0, you must use the **isnull(<expression>, 0)** function (see "Functions" on page 765).

Many common errors in formulating SQL queries are caused by the behavior of NULL. You will have to be careful to avoid these problem areas. See "Search conditions" on page 803 for a description of the effect of three-valued logic when combining search conditions.

Example

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

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

OPEN Statement

Syntax

```
OPEN cursor-name

... [   IUSING DESCRIPTOR sqlda-name                    I   ]
        I USING host-variable, ...                     I

... [ WITH HOLD ] [ ISOLATION LEVEL n ] [ BLOCK n ]

cursor-name:        identifier, or host-variable
sqlda-name:         identifier
```

Purpose

To open a previously declared cursor to access information from the database.

Usage

Embedded SQL, procedures, triggers, and batches.

The USING DESCRIPTOR **sqlda-name**, **host-variable** and BLOCK **n** formats are for Embedded SQL only.

Authorization

Must have SELECT permission on all tables in a SELECT statement or EXECUTE permission on the procedure in a CALL statement.

When the cursor is on a CALL statement, OPEN causes the procedure to execute until the first result set (SELECT statement with no INTO clause) is encountered. If the procedure completes and no result set is found, the SQLSTATE_PROCEDURE_COMPLETE warning is set.

Side effects

None.

See also

DECLARE CURSOR Statement, PREPARE Statement, FETCH Statement, RESUME Statement, CLOSE Statement, "Cursors in Embedded SQL" on page 555, "Using cursors in procedures and triggers" on page 239.

Description

The OPEN statement opens the named cursor. The cursor must be previously declared.

By default, all cursors are automatically closed at the end of the current transaction (COMMIT or ROLLBACK executed). The optional WITH HOLD clause will keep the cursor open for subsequent transactions. It will remain open

until the end of the current connection or until an explicit CLOSE statement is executed. Cursors are automatically closed when a connection is terminated.

The ISOLATION LEVEL clause allows this cursor to be opened at an isolation level different from the current setting of the ISOLATION_LEVEL option. All operations on this cursor will be performed at the specified isolation level regardless of the option setting. If this clause is not specified, then the cursor's isolation level for the entire time the cursor is open is the value of the ISOLATION_LEVEL option when the cursor is opened. See "How locking works" on page 204.

The cursor is positioned before the first row (see "Cursors in Embedded SQL" on page 555 or "Using cursors in procedures and triggers" on page 239).

Embedded SQL

If the cursor name is specified by an identifier or string, then the corresponding DECLARE statement must appear prior to the OPEN in the C program; if the cursor name is specified by a host variable, then the DECLARE cursor statement must execute before the OPEN statement.

The optional USING clause specifies the host variables that will be bound to the place-holder bind variables in the SELECT statement for which the cursor has been declared.

The multiuser support fetches rows in blocks (more than 1 at a time). By default, the number of rows in a block is determined dynamically based on the size of the rows and how long it takes the database engine to fetch each row. The application can specify a maximum number of rows that should be contained in a block by specifying the BLOCK clause. For example, if you are fetching and displaying 5 rows at a time, use **BLOCK 5**. Specifying **BLOCK 0** will cause 1 record at a time to be fetched and also cause a FETCH RELATIVE 0 to always fetch the row again.

After successful execution of the OPEN command, the *sqlerrd[3]* field of the SQLCA (SQLIOESTIMATE) will be filled in with an estimate of the number of input/output operations required to fetch all rows of the query. Also, the *sqlerrd[2]* field of the SQLCA (SQLCOUNT) will be filled in with either the actual number of rows in the cursor (a value greater than or equal to 0), or an estimate thereof (a negative number whose absolute value is the estimate). It will be the actual number of rows if the database engine can compute it without counting the rows. The database can also be configured to always return the actual number of rows (see the ROW_COUNTS option in "SET OPTION Statement" on page 989), but this can be expensive.

Examples

The following examples show the use of OPEN in Embedded SQL.

```
1.  EXEC SQL OPEN employee_cursor;

2.  EXEC SQL PREPARE emp_stat FROM
        'SELECT empnum, empname FROM employee WHERE name like ?';
    EXEC SQL DECLARE employee_cursor CURSOR FOR emp_stat;
    EXEC SQL OPEN employee_cursor USING :pattern;
```

Procedure/trigger example

```
BEGIN
    DECLARE cur_employee CURSOR FOR
                SELECT emp_lname
                FROM   employee ;
    DECLARE name CHAR(40) ;

    OPEN cur_employee;
    LOOP
        FETCH NEXT cur_employee into name ;
            ...
    END LOOP
    CLOSE cur_employee;
END
```

OUTPUT Statement

Syntax

```
OUTPUT TO filename
   . . . [ FORMAT output_format ]
   . . . [ DELIMITED BY string ]
   . . . [ QUOTE string [ ALL ] ]
   . . . [ COLUMN WIDTHS (integer,...) ]
```

Purpose

To output the current query results to a file.

Usage

ISQL.

Authorization

None.

Side effects

The current query results that are displayed in the ISQL data window are repositioned to the top.

See also

SELECT Statement.

Description

The OUTPUT command copies the information retrieved by the *current query* to **filename**. The output format can be specified with the optional FORMAT clause. If no FORMAT clause is specified, the OUTPUT_FORMAT option setting is used (see "SET OPTION Statement" on page 989).

The *current query* is the SELECT or INPUT command which generated the information displayed in the ISQL data window. The OUTPUT command will report an error if there is no current query.

The DELIMITED BY and QUOTE clauses are for the ASCII output format only. The delimiter string will be placed between columns (default comma) and the quote string will be placed around string values (default '—single quote). If ALL is specified in the QUOTE clause, then the quote string will be placed around all values, not just strings.

The COLUMN WIDTH clause is used to specify the column widths for the FIXED format output.

Allowable output formats are:

ASCII The output is an ASCII format file with one row per line in the file. All values are separated by commas and strings are enclosed in apostrophes (single quotes). The delimiter and quote strings can be changed using the DELIMITED BY and QUOTE clauses. If ALL is specified in the QUOTE clause, then all values (not just strings) will be quoted.

Three other special sequences are also used. The two characters \n represent a newline character, \\ represents a single \, and the sequence \xDD represents the character with hexadecimal code DD. This is the default output format.

DBASEII The output is a dBASE II format file with the column definitions at the top of the file. Note that a maximum of 32 columns can be output. Also, note that columns longer than 255 characters will be truncated in the file.

DBASEIII The output is a dBASE III format file with the column definitions at the top of the file. Note that a maximum of 128 columns can be output. Also, note that columns longer than 255 characters will be truncated in the file.

DIF The output is a file in the standard Data Interchange Format.

FIXED The output is fixed format with each column having a fixed width. The width for each column can be specified using the COLUMN WIDTH clause. If this clause is omitted, the width for each column is computed from the data type for the column, and is large enough to hold any value of that data type. No column headings are output in this format.

FOXPRO The output is a FoxPro format file (the FoxPro memo field is different than the dBASE memo field) with the column definitions at the top of the file. Note that a maximum of 128 columns can be output. Also, note that columns longer than 255 characters will be truncated in the file.

LOTUS The output is a Lotus WKS format worksheet. Column names will be put as the first row in the worksheet. Note that there are certain restrictions on the maximum size of Lotus WKS format worksheets that other software (such as Lotus 1-2-3) can load. There is no limit to the size of file ISQL can produce.

SQL The output is an ISQL INPUT command required to recreate the information in the table.

TEXT The output is a TEXT format file which prints the results in columns with the column names at the top and vertical lines separating the columns. This format is similar to that used to display data in the ISQL data window.

WATFILE The output is a WATFILE format file with the column definitions at the top of the file. WATFILE is a tabular file management tool available from WATCOM.

Example

Place the contents of the employee table in a file, in ascii format.

```
SELECT *
FROM   employee ;
OUTPUT TO employee.txt
FORMAT ASCII
```

PARAMETERS Statement

Syntax

```
PARAMETERS parameter1, parameter2, ...
```

Purpose

To specify parameters to an ISQL command file.

Usage

ISQL.

Authorization

None.

Side effects

None.

See also

"Command Files" on page 117, READ Statement.

Description

The PARAMETERS command specifies how many parameters there are to a command file and also gives names to those parameters so that they can be referenced later in the command file.

Parameters are referenced by putting:

```
{parameter1}
```

into the file where you wish the named parameter to be substituted. There must be no spaces between the braces and the parameter name.

If a command file is invoked with fewer than the required number of parameters, ISQL prompts for values of the missing parameters. See "Command Files" on page 117.

Example

The following ISQL command file takes two parameters.

```
PARAMETERS  department_id, file ;
SELECT emp_lname
FROM   employee
WHERE  dept_id = {department_id}
>#{file}.dat;
```

PASSTHROUGH Statement

Syntax

1. PASSTHROUGH FOR userid,. . .
2. PASSTHROUGH FOR SUBSCRIPTION

   . . . TO [(creator)].publication-name [(constant)]

3. PASSTHROUGH STOP

Purpose

To start or stop passthrough mode for SQL Remote administration. Forms 1 and 2 start passthrough mode, while form 3 stops passthrough mode.

Usage

Anywhere.

Authorization

Must have DBA authority.

Side effects

None.

See also

Description

In passthrough mode, any SQL statements are executed by the database server, and are also placed into the transaction log to be sent in messages to subscribers. The recipients of the passthrough SQL statements are either a list of user IDs (form 1) or all subscribers to a given publication.

Passthrough mode may be used to apply changes to a remote database from the consolidated database.

Example

```
PASSTHROUGH FOR rem_db ;
 . . .
( SQL statements to be executed at the remote database )
 . . .
PASSTHROUGH STOP ;
```

PREPARE Statement

Syntax

```
PREPARE statement-name FROM statement

statement-name:    identifier, or host-variable
statement:         string, or host-variable
```

Purpose

To prepare a statement to be executed later or used for a cursor.

Usage

Embedded SQL.

Authorization

None.

Side effects

Any statement previously prepared with the same name is lost.

See also

DECLARE CURSOR Statement, DESCRIBE Statement, OPEN Statement, EXECUTE Statement, DROP STATEMENT Statement.

Description

The PREPARE statement prepares a SQL statement from the **statement** and associates the prepared statement with **statement-name**. This statement name is referenced to execute the statement, or to open a cursor if the statement is a SELECT statement. **Statement-name** may be a host variable of type **a_sql_statement_number** defined in the **sqlca.h** header file that is automatically included. If an identifier is used for the **statement-name**, then only one statement per module may be prepared with this **statement-name**.

If a host variable is used for **statement-name**, it must have the type **short int**. There is a typedef for this type in **sqlca.h** called **a_sql_statement_number**. This type is recognized by the SQL preprocessor and can be used in a DECLARE section. The host variable is filled in by the database during the PREPARE command and need not be initialized by the programmer.

The following is a list of statements that can be PREPARED.

ALTER
CALL
COMMENT ON

CREATE
DELETE
DROP
GRANT
INSERT
REVOKE
SELECT
SET OPTION
UPDATE
VALIDATE TABLE

NOTE: For compatibility reasons, preparing COMMIT, PREPARE TO COMMIT, and ROLLBACK statements is still supported. However, we recommend that you do all transaction management operations with static Embedded SQL because certain application environments may require it. Also, other Embedded SQL systems do not support dynamic transaction management operations.

Drop statement after use

You should make sure that you DROP the statement after use. If you do not, then the memory associated with the statement is not reclaimed.

Example

```
EXEC SQL PREPARE employee_statement FROM
    'SELECT * from employee';
```

PREPARE TO COMMIT Statement

Syntax

PREPARE TO COMMIT

Purpose

To check whether a COMMIT can be performed.

Usage

Anywhere.

Authorization

None.

Side effects

None.

See also

COMMIT Statement, ROLLBACK Statement.

Description

The PREPARE TO COMMIT statement tests whether a COMMIT can be performed successfully. The statement will cause an error if a COMMIT is not possible without violating the integrity of the database.

Example

The following sequence of statements allows the delete to take place, even though it causes integrity violations. The PREPARE TO COMMIT statement causes an error reporting the violation.

```
SET OPTION wait_for_commit= 'ON' ;
DELETE
FROM   department
WHERE  dept_id = 100 ;
PREPARE TO COMMIT ;
```

PUT Statement

Syntax

```
PUT  cursor-name

...     | USING  DESCRIPTOR  sqlda-name                    |
        | [ . . . ARRAY :nnn  ]                            |
        | FROM  host-variable-list                         |
```

cursor-name:	identifier, or host-variable
sqlda-name:	identifier
host-variable-list:	may contain indicator variables

Purpose

To insert a row into the table(s) specified by the cursor. See "SET Statement" on page 1008 for putting LONG VARCHAR or LONG BINARY values into the database.

Usage

Embedded SQL.

Authorization

Must have INSERT permission.

Side effects

None.

See also

UPDATE Statement, UPDATE (positioned) Statement, DELETE Statement, DELETE (positioned) Statement, INSERT Statement.

Description

Inserts a row into the named cursor. Values for the columns are taken from the SQLDA or the host variable list in a one-to-one correspondence with the columns in the INSERT statement (for an INSERT cursor) or the columns in the select list (for a SELECT cursor).

If the **sqldata** pointer in the SQLDA is the null pointer, then no value is specified for that column. If the column has a DEFAULT VALUE associated with it, that will be used, otherwise a NULL value will be used. If no values are specified for any of the columns of one particular table involved in the cursor, then no row will be inserted into that table.

The optional ARRAY clause can be used to carry out wide puts, which insert more than one row at a time and which may improve performance. The value

nnn is the number of rows to be inserted. The SQLDA must contain nnn * (columns per row) variables. The first row is placed in SQLDA variables 0 to (columns per row)-1, and so on.

NOTE: Values can be specified for columns of **one table only**. Inserting into two different tables through a cursor is not supported.

NOTE: When inserting into a cursor, the position of the inserted row is unpredictable. If the cursor involves a temporary table, the inserted record will not show up in the current cursor at all.

Example

```
EXEC SQL PUT cur_employee FROM :emp_id, :emp_lname;
```

READ Statement

Syntax

```
READ filename [ parameters ]
```

Purpose

To read ISQL commands from a file.

Usage

ISQL.

Authorization

None.

Side effects

None.

See also

"Command Files" on page 117, PARAMETERS Statement.

Description

The READ command reads a sequence of ISQL commands from the named file. This file can contain any valid ISQL command including other READ commands. READ commands can be nested to any depth. To find the command file, ISQL will first search the current directory, then the directories specified in the environment variable **SQLPATH**, then the directories specified in the environment variable **PATH**. If the named file has no file extension, ISQL will also search each directory for the same file name with the extension `.sql`.

Parameters can be listed after the name of the command file. These parameters correspond to the parameters named on the PARAMETERS command at the beginning of the command file (see "PARAMETERS Statement" on page 960). ISQL will then substitute the corresponding parameter wherever the source file contains

```
{parameter-name}
```

where **parameter-name** is the name of the appropriate parameter.

The parameters passed to a command file can be identifiers, numbers, quoted identifiers, or strings. When quotes are used around a parameter, the quotes are put into the text during the substitution. Parameters which are not identifiers, numbers, or strings (contain spaces or tabs) must be enclosed in square brackets (`[]`). This allows for arbitrary textual substitution in the command file.

If not enough parameters are passed to the command file, ISQL prompts for values for the missing parameters.

Examples

```
1.  READ status.rpt '160'

2.  READ birthday.sql [>= '1988-1-1'] [<= '1988-1-30']
```

RELEASE SAVEPOINT Statement

Syntax

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

Purpose

Release a savepoint within the current transaction.

Usage

Anywhere.

Authorization

There must have been a corresponding SAVEPOINT within the current transaction.

Side effects

None.

See also

SAVEPOINT Statement, ROLLBACK TO SAVEPOINT Statement.

Description

Release a savepoint. The *savepoint-name* is an identifier specified on a SAVEPOINT command within the current transaction. If *savepoint-name* is omitted, the most recent savepoint is released.

For a description of savepoints, see "Savepoints within transactions" on page 210. Releasing a savepoint does not do any type of COMMIT. It simply removes the savepoint from the list of currently active savepoints.

RESIGNAL Statement

Syntax

RESIGNAL

Purpose

Resignal an exception condition.

Usage

Within an exception handler in a procedure or trigger.

Authorization

None.

Side effects

None.

See also

"Using exception handlers in procedures and triggers" on page 249, Compound statements, SIGNAL Statement.

Description

Within an exception handler, RESIGNAL allows you to quit the compound statement with the exception still active. The exception will be handled by another exception handler or returned to the application. Any actions by the exception handler before the RESIGNAL are not undone.

Example

The following fragment returns all exceptions except for Column Not Found to the application.

```
DECLARE COLUMN_NOT_FOUND EXCEPTION FOR SQLSTATE '52003';
EXCEPTION
    WHEN COLUMN_NOT_FOUND THEN
        SET message='Column not found' ;
    WHEN OTHERS THEN
        RESIGNAL ;
```

RESUME Statement

Syntax

1. RESUME cursor-name
2. RESUME [ALL]

cursor-name: identifier, or host-variable

Purpose

To resume a procedure following a query.

Usage

Form 1: Embedded SQL, procedures, triggers, and batches. Form 1 using a **host-variable** is for Embedded SQL only. Form 2: ISQL.

Authorization

The cursor must have been previously opened.

Side effects

None.

See also

"Returning results from procedures" on page 235, DECLARE CURSOR Statement.

Description

This statement resumes execution of a procedure that returns result sets. The procedure executes until the next result set (SELECT statement with no INTO clause) is encountered. If the procedure completes and no result set is found, the SQLSTATE_PROCEDURE_COMPLETE warning is set. This warning is also set when you RESUME a cursor for a SELECT statement.

The ISQL RESUME statement (Format 2) resumes the current procedure. If ALL is not specified, executing RESUME displays the next result set or, if no more result sets are returned, completes the procedure.

The ISQL RESUME ALL statement cycles through all result sets in a procedure, without displaying them, and completes the procedure. This is useful mainly in testing procedures.

Examples

Embedded SQL example

```
1.  EXEC SQL RESUME cur_employee;

2.  EXEC SQL RESUME :cursor_var;
```

ISQL examples

```
CALL sample_proc() ;
RESUME ALL;
```

RETURN Statement

Syntax

RETURN [(expression)]

Purpose

To exit from a function or procedure unconditionally, optionally providing a return value. Statements following RETURN are not executed.

Usage

Procedures, triggers, and batches only

Authorization

None.

Side effects

None.

See also

CREATE FUNCTION Statement, CREATE PROCEDURE Statement, Compound statements.

Description

A RETURN statement causes an immediate exit from the function or procedure. If *expression* is supplied, the value of *expression* is returned as the value of the function or procedure.

Within a function, the expression should be of the same data type as the function's RETURNS data type.

RETURN for procedures is for use with Transact-SQL-compatible procedures. For information, see "Transact-SQL Procedure Language" on page 497.

Example

The following function returns the product of three numbers:

```
CREATE FUNCTION product ( a numeric,
                          b numeric ,
                          c numeric) RETURNS numeric
BEGIN
        RETURN ( a * b * c ) ;
END
```

Calculate the product of three numbers:

```
SELECT product (2, 3, 4)
```

```
   product(2, 3, 4)
   24
```

The following procedure uses the RETURN statement to avoid executing a complex query if it is meaningless:

```
CREATE PROCEDURE customer_products
                ( in customer_id integer DEFAULT NULL)
RESULT ( id integer, quantity_ordered integer )
BEGIN
    IF customer_id NOT IN (SELECT id FROM customer)
    OR  customer_id IS NULL THEN
       RETURN
    ELSE
        SELECT product.id,sum(sales_order_items.quantity)
        FROM   product,
               sales_order_items,
               sales_order
        WHERE  sales_order.cust_id=customer_id
          AND  sales_order.id=sales_order_items.id
          AND  sales_order_items.prod_id=product.id
        GROUP  BY product.id
    END IF
END
```

REVOKE Statement

Syntax

Format 1

```
          | CONNECT                                  |
          | DBA                                      |
REVOKE    | RESOURCE                                 |
          | GROUP                                    |
          | MEMBERSHIP IN GROUP  userid,...          |

  . . .  FROM  userid,...
```

Format 2

```
          | ALTER                                    |
          | DELETE                                   |
          | INSERT                                   |
REVOKE    | REFERENCE                                | ,...
          | SELECT                                   |
          | UPDATE  [(column-name,...)]              |
          | ALL  [PRIVILEGES]                        |

  . . .  ON  [creator.]table-name  FROM  userid,...
```

Format 3

```
REVOKE  EXECUTE  ON  [creator.]procedure-name  FROM  userid,...
```

Purpose

To remove permissions for specified user(s).

Usage

Anywhere.

Authorization

Must be the grantor of the permissions that are being revoked or must have DBA authority.

Side effects

Automatic commit.

See also

GRANT Statement.

Description

The REVOKE command is used to remove permissions that were given using the GRANT command. Format 1 is used to revoke special user permissions and

format 2 is used to revoke table permissions. Format 3 is used to revoke permission to execute a procedure. REVOKE CONNECT is used to remove a user ID from a database. REVOKE GROUP will automatically REVOKE MEMBERSHIP from all members of the group.

Examples

Prevent user Dave from updating the employee table.

```
REVOKE UPDATE ON employee FROM dave ;
```

Revoke resource permissions from user Jim.

```
REVOKE RESOURCE FROM Jim ;
```

Disallow the Finance group from executing the procedure sp_customer_list

```
REVOKE EXECUTE ON sp_customer_list
FROM   finance ;
```

Drop user ID FranW from the database.

```
REVOKE CONNECT FROM FranW
```

REVOKE CONSOLIDATE Statement

Syntax

REVOKE CONSOLIDATE FROM userid,. . .

Purpose

To stop a consolidated database from receiving SQL Remote messages from this database.

Usage

Anywhere.

Authorization

Must have DBA authority.

Side effects

Automatic commit. Drops all subscriptions for the user.

See also

Description

CONSOLIDATE permissions must be granted at a remote database for the user ID representing the consolidated database. The REVOKE CONSOLIDATE statement removes the consolidated database user ID from the list of users receiving messages from the current database.

Example

```
REVOKE CONSOLIDATE FROM condb
```

REVOKE PUBLISH Statement

Syntax

REVOKE PUBLISH FROM userid

Purpose

To terminate the identification of the named userid as the CURRENT publisher.

Usage

Anywhere.

Authorization

Must have DBA authority.

Side effects

Automatic commit.

See also

GRANT PUBLISH Statement, REVOKE REMOTE Statement, CREATE PUBLICATION Statement, CREATE SUBSCRIPTION Statement.

Description

Each database in a SQL Remote installation is identified in outgoing messages by a **publisher** user ID. The current publisher user ID can be found using the CURRENT PUBLISHER special constant. The following query identifies the current publisher:

```
SELECT CURRENT PUBLISHER
```

The REVOKE PUBLISH statement ends the identification of the named user ID as the publisher.

You should not REVOKE PUBLISH from a database while the database has active SQL Remote publications or subscriptions.

Issuing a REVOKE PUBLISH statement at a database has several consequences for a SQL Remote installation:

- You will not be able to insert data into any tables with a CURRENT PUBLISHER column as part of the primary key. publisher user ID, and so will not be accepted by recipient databases.

If you change the publisher user ID at any consolidated or remote database in a SQL Remote installation, you must ensure that the new publisher user ID is granted REMOTE permissions on all databases receiving messages from the

database. This will generally require all subscriptions to be dropped and recreated.

Example

```
REVOKE PUBLISH FROM publisher_ID
```

REVOKE REMOTE Statement

Syntax

REVOKE REMOTE FROM userid,. . .

Purpose

To stop a user from being able to receive SQL Remote messages from this database.

Usage

Anywhere.

Authorization

Must have DBA authority.

Side effects

Automatic commit. Drops all subscriptions for the user.

See also

Description

REMOTE permissions are required for a user ID to receive messages in a SQL Remote replication installation. The REVOKE REMOTE statement removes a user ID from the list of users receiving messages from the current database.

Example

```
REVOKE REMOTE FROM SamS
```

ROLLBACK Statement

Syntax

```
ROLLBACK [ WORK ]
```

Purpose

To undo any changes made since the last COMMIT or ROLLBACK.

Usage

Anywhere.

Authorization

Must be connected to the database.

Side effects

Closes all cursors not opened WITH HOLD.

See also

COMMIT Statement, ROLLBACK TO SAVEPOINT Statement.

Description

The ROLLBACK command ends a logical unit of work (transaction) and undoes all changes made to the database during this transaction. A *transaction* is the database work done between COMMIT or ROLLBACK commands on one database connection.

ROLLBACK TO SAVEPOINT Statement

Syntax

```
ROLLBACK TO SAVEPOINT [savepoint-name]
```

Purpose

To cancel any changes made since a SAVEPOINT.

Usage

Anywhere.

Authorization

There must have been a corresponding SAVEPOINT within the current transaction.

Side effects

None.

See also

SAVEPOINT Statement, RELEASE SAVEPOINT Statement, ROLLBACK Statement.

Description

The ROLLBACK TO SAVEPOINT command will undo any changes that have been made since the SAVEPOINT was established. Changes made prior to the SAVEPOINT are not undone; they are still pending. For a description of savepoints, see "Savepoints within transactions" on page 210.

The *savepoint-name* is an identifier that was specified on a SAVEPOINT command within the current transaction. If *savepoint-name* is omitted, the most recent savepoint is used. Any savepoints since the named savepoint are automatically released.

SAVEPOINT Statement

Syntax

```
SAVEPOINT [savepoint-name]
```

Purpose

To establish a savepoint within the current transaction.

Usage

Anywhere.

Authorization

None.

Side effects

None.

See also

RELEASE SAVEPOINT Statement, ROLLBACK TO SAVEPOINT Statement.

Description

Establish a savepoint within the current transaction. The *savepoint-name* is an identifier that can be used in an RELEASE SAVEPOINT or ROLLBACK TO SAVEPOINT command. All savepoints are automatically released when a transaction ends. See "Savepoints within transactions" on page 210.

Savepoints that are established while a trigger is executing or while an atomic compound statement is executing are automatically released when the atomic operation ends.

SELECT Statement

Syntax

```
SELECT [ ALL | DISTINCT ] select-list

...[    | INTO host-variable-list                                  |   ]
        | INTO variable-list                                       |

...     [ FROM table-list ]

...     [ WHERE search-condition ]

...     [ GROUP BY column-name, ... ]

...     [ HAVING search-condition ]

...     | [ ORDER BY expression [ ASC | DESC ], ... ]              |
        | [ ORDER BY integer [ ASC | DESC ], ... ]                 |


select-list:
        | table-name.*                                             |, ...
        | expression [ [ AS ] alias-name ]                         |
        | *                                                        |
```

Purpose

To retrieve information from the database.

Usage

Anywhere

The INTO clause with *host-variable-list* is used in Embedded SQL only. The INTO clause with *variable-list* is used in procedures and triggers only.

Authorization

Must have SELECT permission on the named tables and views.

Side effects

None.

See also

FROM Clause, Search conditions, CREATE VIEW Statement, UNION Operation, Expressions, DECLARE Statement, OPEN Statement, FETCH Statement.

Description

The SELECT statement is used for retrieving results from the database. A SELECT statement can be used in ISQL to browse data in the database or to export data from the database to an external file.

A SELECT statement can also be used in procedures and triggers or in Embedded SQL. The SELECT statement with an INTO clause is used for retrieving results from the database when the SELECT statement only returns one row. For multiple row queries, you must use cursors. (See DECLARE, OPEN and FETCH .) A SELECT statement can also be used to return a result set from a procedure. (See "Returning results from procedures" on page 235). (Procedures are supported in SQL Anywhere Server products only).

The various parts of the SELECT command are described below:

ALL or DISTINCT

If neither ALL nor DISTINCT is specified, ALL rows which satisfy the clauses of the SELECT command are retrieved. If DISTINCT is specified, duplicate output rows are eliminated. This is called the *projection* of the result of the command. In many cases, commands take significantly longer to execute when DISTINCT is specified. Thus, the use of DISTINCT should be reserved for cases where it is necessary.

If DISTINCT is used, the command cannot contain an aggregate function with a DISTINCT parameter.

select list

The select list is a list of expressions separated by commas specifying what will be retrieved from the database. If asterisk (*) is specified, it is expanded to select all columns of all tables in the FROM clause (**table-name.*** is expanded to select all columns of the named table). Aggregate functions are allowed in the select list (see "Functions" on page 765). Subqueries are also allowed in the select list (see "Expressions" on page 795). Each subquery must be within parentheses.

Alias-names can be used throughout the query to represent the aliased expression. Alias names are also displayed by ISQL at the top of each column of output from the SELECT command. If the optional alias name is not specified after an expression, ISQL will display the expression.

INTO host-variable-list

This clause is used in Embedded SQL only. It specifies where the results of the SELECT statement will go. There must be one host-variable item for each item in the select list. Select list items are put into the host variables in order. An indicator host variable

is also allowed with each **host-variable** so the program can tell if the select list item was NULL.

INTO variable-list

This clause is used in procedures and triggers only. It specifies where the results of the SELECT statement will go. There must be one variable for each item in the select list. Select list items are put into the variables in order.

FROM table-list

Rows are retrieved from the tables and views specified in the **table list**. Joins can be specified using join operators. For more information, see "FROM Clause" on page 915. A SELECT statement with no FROM clause can be used to display the values of expressions not derived from tables. For example:

```
SELECT @@version
```

displays the value of the global variable @@version. This is equivalent to:

```
SELECT  @@version
FROM    DUMMY
```

WHERE search-condition

The **search-condition** restricts the rows that will be selected from the tables named in the FROM clause. It is also used to do joins between multiple tables. This is accomplished by putting a condition in the WHERE clause that relates a column or group of columns from one table with a column or group of columns from another table. Both tables must be listed in the FROM clause.

See "Search conditions" on page 803 for a full description.

GROUP BY { column-name | alias | function }, ...

Group multiple rows together from the database. You can group by columns or alias names or functions. GROUP BY expressions must also appear in the select list. The result of the query contains one row for each distinct set of values in the named columns, aliases, or functions. The resulting rows are often referred to as **groups** since there is one row in the result for each group of rows from the table list. For the sake of GROUP BY, all NULL values are treated as identical. Aggregate functions can then be applied to these groups to get meaningful results.

When GROUP BY is used, the select list, HAVING clause and ORDER BY clause cannot reference any identifiers except those

named in the GROUP BY clause. The exception is that the select list and HAVING clause may contain aggregate functions.

HAVING search-condition

Restricts which groups will be selected based on the group values and not on the individual row values. The HAVING clause can only be used if either the command has a GROUP BY clause or if the select list consists solely of aggregate functions. Any column names referenced in the HAVING clause must either be in the GROUP BY clause or be used as a parameter to an aggregate function in the HAVING clause.

ORDER BY expression, ...

Sort the results of a query. Each item in the ORDER BY list can be labeled as ASC for ascending order or DESC for descending order. Ascending is assumed if neither is specified. If the expression is an integer **N**, then the query results will be sorted by the **N**'th item in the select list.

Embedded SQL example

```
SELECT count(*) INTO :size FROM employee
```

Examples

List all the tables and views in the system catalog.

```
SELECT tname
FROM   SYS.SYSCATALOG
WHERE  tname LIKE 'SYS%' ;
```

list all customers and the total value of their orders

```
SELECT company_name,
       CAST( sum(sales_order_items.quantity *
       product.unit_price) AS INTEGER) VALUE
FROM   customer LEFT OUTER JOIN sales_order
                LEFT OUTER JOIN sales_order_items
                LEFT OUTER JOIN product
GROUP  BY company_name
ORDER  BY VALUE DESC
```

How many employees are there?

```
SELECT count(*)
FROM   Employee ;
```

SET CONNECTION Statement

Syntax

SET CONNECTION [connection-name]

connection-name: identifier, string or host-variable

Purpose

To change the active database connection.

Usage

Embedded SQL, ISQL.

Authorization

None.

Side effects

None.

See also

CONNECT Statement, DISCONNECT Statement.

Description

The SET CONNECTION command changes the active database connection to **connection-name**. The current connection state is saved and will be resumed when it again becomes the active connection. If **connection-name** is omitted and there is a connection that was not named, that connection becomes the active connection.

NOTE: When cursors are opened in Embedded SQL, they are associated with the current connection. When the connection is changed, the cursor names will not be accessible. The cursors remain active and in position and will become accessible when the associated connection becomes active again.

Example

The following example is in Embedded SQL.

```
EXEC SQL SET CONNECTION :conn_name;
```

From ISQL, set the current connection to the connection named conn1.

```
SET CONNECTION conn1 ;
```

SET OPTION Statement

Syntax

1. SET [TEMPORARY] OPTION

   ... [userid. | PUBLIC.]option-name = [option-value]

2. SET PERMANENT

3. SET

userid:	identifier, string or host-variable
option-name:	identifier, string or host-variable
option-value:	host-variable (indicator allowed), string, identifier, or number

Note: Formats 2 and 3 are ISQL only.

Purpose

To change ISQL, database, and replication options.

Usage

All (format 1), ISQL (all formats).

Authorization

None required to set your own options. Must have DBA authority to set database or replication options for another user or PUBLIC.

Side effects

If TEMPORARY is not specified, an automatic commit is performed.

See also

"The database engine" on page 685, CONFIGURE Statement, GET OPTION Statement.

The classes of options are:

- Database options.
- Replication options.
- ISQL options.

Description

The SET command is used to change options for both the database and the ISQL environment. The **userid** parameter can be specified to change someone else's options, however, you must have DBA authority to do so. Specifying

TEMPORARY will change the option setting for the current connection only. In this case, **userid** cannot be specified. Also, in embedded SQL, only database options can be set temporarily.

Changing a **PUBLIC** option sets the option value that will be used for any userid which has not SET their own value for that option. An option cannot be set for a user ID other than **PUBLIC** unless there is already a **PUBLIC** setting for that option. Only a user with DBA authority is allowed to set options for another user ID (including **PUBLIC**).

If the **option-value** is omitted, the specified option setting will be deleted from the database. If it was a personal option setting, then the value used will revert back to the PUBLIC setting. If a TEMPORARY option is deleted, the option setting will revert back to the permanent setting.

Temporary setting
Only your own ISQL and database options can be made TEMPORARY; options for other software and any options set for another user will produce an error if TEMPORARY is specified.

SET PERMANENT (format 2) stores all current ISQL options in the **SYS.SYSOPTIONS** table in the database. These settings will automatically be established every time ISQL is started for the current userid.

Format 3 is used to display all of the current option settings. If there are temporary options set for ISQL or the database engine, these will be displayed; otherwise, the permanent option settings are displayed.

The options are divided into two groups: *Database options* and *ISQL options*.

Database options

OPTION	VALUES	DEFAULT
AUTOMATIC_TIMESTAMP	ON,OFF	OFF
ALLOW_NULLS_BY_DEFAULT	ON,OFF	ON
BLOCKING	ON,OFF	ON
CHECKPOINT_TIME	number of minutes	60
CONVERSION_ERROR	ON,OFF	ON
COOPERATIVE_COMMITS	ON,OFF	ON
COOPERATIVE_COMMIT_TIMEOUT	integer	250
DATE_FORMAT	string	'YYYY-MM-DD' 'MM/DD/YYYY' (1)
DATE_ORDER	'YMD','DMY','MDY'	'YMD' 'MDY' (1)
DELAYED_COMMITS	ON,OFF	OFF
DELAYED_COMMIT_TIMEOUT	integer	500
ISOLATION_LEVEL	0,1,2,3	0
PRECISION	number of digits	30
QUERY_PLAN_ON_OPEN	ON,OFF	OFF
QUOTED_IDENTIFIER	ON,OFF	ON
RECOVERY_TIME	number of minutes	2
ROW_COUNTS	ON,OFF	OFF
SCALE	number of digits	6
THREAD_COUNT	number of threads	0
TIME_FORMAT	string	'HH:NN:SS.SSS'
TIMESTAMP_FORMAT	string	'YYYY-MM-DD HH:NN:SS.SSS'
WAIT_FOR_COMMIT	ON,OFF	OFF

$(1) Default for databases created with Watcom SQL Version 3.0.

ALLOW_NULLS_BY_DEFAULT

Controls whether new columns created without specifying either NULL or NOT NULL are allowed contain NULL values. The ALLOW_NULLS_BY_DEFAULT option is included for Transact-SQL compatibility. The default is ON. See "Setting options for Transact-SQL-compatibility" on page 445.

AUTOMATIC_TIMESTAMP

Controls whether any new columns with the TIMESTAMP data type that do not have an explicit default value defined are given a default value of the Transact-SQL `timestamp` value.. The AUTOMATIC_TIMESTAMP option is included for Transact-SQL compatibility. The default is OFF. See "Setting options for Transact-SQL-compatibility" on page 445.

BLOCKING

Controls whether locking conflicts result in one user becoming blocked or an error condition being returned. The default is ON. See "Transaction blocking and deadlock" on page 208.

CHECKPOINT_TIME

Set the maximum desired length of time that the database engine will run without doing a checkpoint. The time is specified in minutes (the default value set by DBINIT is 60). This option is used with the RECOVERY_TIME option to decide when checkpoints should be done. See "The checkpoint log" on page 333.

Because this is a global option for the database, only the PUBLIC setting is used. Individual settings for users will have no effect. Also, changing this option does not take effect immediately. You must shut down the database engine and restart for the change to take effect.

CONVERSION_ERROR

Controls whether conversion errors will be reported by the database, or ignored. If conversion errors are ignored, the NULL value is used in place of the value that could not be converted.

COOPERATIVE_COMMITS

When set to ON (the default), the database engine or server does not reply to a COMMIT statement until the transaction log entry for the COMMIT has been written to disk. This option must be set to ON for ANSI/ISO COMMIT behavior. When this option is ON, the log is written to disk according to the COOPERATIVE_COMMIT_TIMEOUT option setting. Setting COOPERATIVE_COMMITS to ON, and the COOPERATIVE_COMMIT_TIMEOUT option to a high value, promotes overall engine throughput at the cost of a longer turnaround time for the individual connection.

COOPERATIVE_COMMIT_TIMEOUT

This option only has meaning when COOPERATIVE_COMMITS is set to ON. The COOPERATIVE_COMMIT_TIMEOUT option setting governs when a COMMIT entry in the transaction log is written to disk. With COOPERATIVE_COMMITS set to ON, the database engine waits for the number of milliseconds set in the COOPERATIVE_COMMIT_TIMEOUT option for other connections to fill a page of the log before writing to disk. The default setting is 250 milliseconds.

DATE_FORMAT

Sets the format used for dates retrieved from the database. The format is a string using the following symbols:

Symbol	Description
yy	Two digit year
yyyy	Four digit year
mm	Two digit month, or two digit minutes if following a colon (as in 'hh:mm')
mmm[m...]	Character short form for months—as many characters as there are m's
dd	Two digit day of month
ddd[d...]	Character short form for day of the week
hh	Two digit hours
nn	Two digit minutes
aa	Am or pm (12 hour clock)
pp	Pm if needed (12 hour clock)
f	Use French days and months

Each symbol will be substituted with the appropriate data for the date being formatted. Any format symbol that represents character rather than digit output can be put in upper case which will cause the substituted characters to also be in upper case. For numbers, using mixed case in the format string will suppress leading zeros.

Note

The default date format has changed since Watcom SQL Version 3.0. The new format corresponds to ISO date format specifications ('YYYY-MM-DD'). Databases created with Watcom SQL Version 3.0 will still use the old option value. In addition, the new default date format does not specify hours and minutes. TIMESTAMP_FORMAT includes hours, minutes and seconds.

DELAYED_COMMITS

When set to ON (the default is OFF), the database engine or server replies to a COMMIT statement immediately instead of waiting until the transaction log entry for the COMMIT has been written to disk. This option must be set to OFF for ANSI/ISO COMMIT behavior. There is a slight chance that a transaction may be lost even though committed if a system failure occurs after the engine replies to a COMMIT, but before the page is written to disk. When this option is ON, the log is written to disk when the log page is full or according to the DELAYED_COMMIT_TIMEOUT option setting, whichever is first. Setting DELAYED_COMMITS to ON, and the DELAYED_COMMIT_TIMEOUT option to a high value, promotes response time at the cost of security.

DELAYED_COMMIT_TIMEOUT

This option only has meaning when DELAYED_COMMITS is set to ON. With DELEAYED_COMMITS set ON, the DELAYED_COMMIT_TIMEOUT option setting governs when a COMMIT entry in the transaction log is written to disk. With DELAYED_COMMITS set to ON, the database engine waits for the number of milliseconds set in the DELAYED_COMMIT_TIMEOUT option for other connections to fill a page of the log before writing the current page contenst to disk. The default setting is 500 milliseconds.

DATE_ORDER

The database option DATE_ORDER is used to determine whether 10/11/12 is Oct 11 1912, Nov 12 1910, or Nov 10 1912. The option can have the value 'MDY', 'YMD', or 'DMY'.

Note
The default date order has changed since Watcom SQL Version 3.0. The new order ('YMD') corresponds to ISO date format specifications. Databases created with Watcom SQL Version 3.0 will still use the old option value which was 'MDY'.

ISOLATION_LEVEL

Controls the locking isolation level as follows.

- **0** Allow dirty reads, non-repeatable reads, and phantom rows.
- **1** Prevent dirty reads. Allow non-repeatable reads and phantom rows.
- **2** Prevent dirty reads and guarantee repeatable reads. Allow phantom rows.
- **3** Serializable. Do not allow dirty reads, guarantee repeatable reads, and do not allow phantom rows.

The default is 0. See "Isolation levels and consistency" on page 205.

PRECISION

Specifies the maximum number of digits in the result of any decimal arithmetic. Precision is the total number of digits to the left and right of the decimal point. The SCALE option specifies the minimum number of digits after the decimal point when an arithmetic result is truncated to the maximum PRECISION.

Multiplication, division, addition, subtraction, and aggregate functions can all have results that exceed the maximum precision.

For example, when a DECIMAL(8,2) is multiplied with a DECIMAL(9,2), the result could require a DECIMAL(17,4). If PRECISION is 15, only 15 digits will be kept in the result. If SCALE is 4, the result will be a DECIMAL(15,4). If

SCALE is 2, the result will be a DECIMAL(15,2). In both cases, there is a possibility for overflow.

PRECISION

Specifies the maximum number of digits in the result of any decimal arithmetic. Precision is the total number of digits to the left and right of the decimal point. The SCALE option specifies the minimum number of digits after the decimal point when an arithmetic result is truncated to the maximum PRECISION.

Multiplication, division, addition, subtraction, and aggregate functions can all have results that exceed the maximum precision.

For example, when a DECIMAL(8,2) is multiplied with a DECIMAL(9,2), the result could require a DECIMAL(17,4). If PRECISION is 15, only 15 digits will be kept in the result. If SCALE is 4, the result will be a DECIMAL(15,4). If SCALE is 2, the result will be a DECIMAL(15,2). In both cases, there is a possibility for overflow.

QUERY_PLAN_ON_OPEN

In previous versions of the product, each time an OPEN was done on a cursor, the engine would return in the SQLCA **sqlerrmc** field a string representing the query plan (limited to 70 bytes). A more complete description can be obtained using the EXPLAIN statement or the PLAN function. For this reason, computing and returning the query plan on an OPEN is needed only for compatibility with old applications. The **QUERY_PLAN_ON_OPEN** option controls whether the plan is returned on an OPEN. By default, the setting is `off`.

QUOTED_IDENTIFIER

Controls whether strings enclosed in double quotes are interpreted as identifiers (ON) or as literal strings (OFF). If set to ON then strings enclosed in single quotes are interpreted as literal strings. The QUOTED_IDENTIFIER option is included for Transact-SQL compatibility. The default is ON. See "Setting options for Transact-SQL-compatibility" on page 445.

ROW_COUNTS

Specifies whether the database will always count the number of rows in a query when it is opened. If this option is off, the row count (see "The SQL communication area" on page 550) will usually only be an estimate. If this option is on, the row count will always be accurate but opening queries will take significantly longer in many cases.

SCALE

Specifies the minimum number of digits after the decimal point when an arithmetic result is truncated to the maximum PRECISION.

Multiplication, division, addition, subtraction, and aggregate functions can all have results that exceed the maximum precision. See the PRECISION option for an example.

THREAD_COUNT

Sets the number of execution threads that will be used in the database engine while running with multiple users. This number controls the number of concurrent requests that the database engine will process. A value of 0 (the default) means an operating system specific value:

16-bit DOS	3
16-bit QNX	8
16-bit Windows	8
All 32-bit	20

Because this is a global option for the database, only the PUBLIC setting is used. Individual settings for users will have no effect. Also, changing this option does not take effect immediately. You must shut down the database engine and restart for the change to take effect.

TIME_FORMAT

Sets the format used for times retrieved from the database. The format is a string using the following symbols:

- **hh** Two digit hours (24 hour clock)
- **nn** Two digit minutes
- **mm** Two digit minutes if following a colon (as in 'hh:mm')
- **ss[.s...]** Two digit seconds plus optional fraction

Each symbol will be substituted with the appropriate data for the date being formatted. Any format symbol that represents character rather than digit output can be put in uppercase which will cause the substituted characters to also be in uppercase. For numbers, using mixed case in the format string will suppress leading zeros.

TIMESTAMP_FORMAT

Sets the format used for timestamps retrieved from the database. The format is a string using the following symbols:

Symbol	Description
yy	Two digit year
yyyy	Four digit year
mm	Two digit month, or two digit minutes if following a colon (as in 'hh:mm')
mmm[m...]	Character short form for months—as many characters as there are m's
dd	Two digit day of month
ddd[d...]	Character short form for day of the week
hh	Two digit hours
nn	Two digit minutes
aa	Am or pm (12 hour clock)
pp	Pm if needed (12 hour clock)
f	Use French days and months

Each symbol will be substituted with the appropriate data for the date being formatted. Any format symbol that represents character rather than digit output can be put in uppercase which will cause the substituted characters to also be in uppercase. For numbers, using mixed case in the format string will suppress leading zeros.

WAIT_FOR_COMMIT

If this option is on, the database will not check foreign key integrity until the next COMMIT command. Otherwise, all foreign keys not created with the CHECK ON COMMIT option are checked as they are inserted, updated or deleted.

Replication options

OPTION	VALUES	DEFAULT
DELETE_OLD_LOGS	ON,OFF	OFF
REPLICATE_ALL	ON,OFF	OFF
VERIFY_ALL_COLUMNS	ON,OFF	OFF

DELETE_OLD_LOGS

This option is used by SQL Remote and by the SQL Anywhere Replication Agent. The default setting is OFF. When set to ON, the Message Agent (DBREMOTE) deletes each old transaction log when all the changes it contains have been sent and confirmed as received.

REPLICATE_ALL

This option is used by the SQL Anywhere LTM only. The default setting is OFF. When it is set to ON, the entire database is set to act as a primary site in a Replication Server installation. All changes to the database are sent to Replication Server by the LTM.

VERIFY_ALL_COLUMNS

This option is used by SQL Remote only. The default setting is OFF. When set to ON, messages containing updates published by the local database are sent with all column values included, and a conflict in any column triggers a RESOLVE UPDATE trigger at the subscriber database.

ISQL options

OPTION	VALUES	DEFAULT
AUTO_COMMIT	ON,OFF	OFF
AUTO_REFETCH	ON,OFF	ON
BELL	ON,OFF	ON
COMMAND_DELIMITER	string	';'
COMMIT_ON_EXIT	ON,OFF	ON
ECHO	ON,OFF	ON
HEADINGS	ON,OFF	ON
INPUT_FORMAT	ASCII FIXED DIF DBASE DBASEII DBASEIII FOXPRO LOTUS WATFILE	ASCII
ISQL_LOG	file-name	''
NULLS	string	'(NULL)'
ON_ERROR	STOP CONTINUE PROMPT EXIT NOTIFY_CONTINUE NOTIFY_STOP NOTIFY_EXIT	PROMPT
OUTPUT_FORMAT	TEXT ASCII FIXED DIF DBASEII DBASEIII FOXPRO LOTUS SQL WATFILE	ASCII
OUTPUT_LENGTH	integer	0
STATISTICS	0,3,4,5,6	3
TRUNCATION_LENGTH	integer	30

AUTO_COMMIT

If AUTO_COMMIT is `'on'`, a database COMMIT is performed after each successful command and a ROLLBACK after each failed command. Otherwise, a COMMIT or ROLLBACK is only performed when the user issues a COMMIT or ROLLBACK command or if the user issues a SQL command which causes an automatic commit such as the CREATE TABLE command.

AUTO_REFETCH

If AUTO_REFETCH is `'on'`, then the current query results displayed in the Data window will be refetched from the database after **any** INSERT, UPDATE or DELETE command. Depending on how complicated the query is, this may take some time. For this reason, it can be turned `'off'`.

BELL

Controls whether the bell will sound when an error occurs.

COMMAND_DELIMITER

Sets the string indicating the termination of a command in ISQL. The default setting is a single semi-colon ';'. This must be changed in order to create triggers and procedures from ISQL using the CREATE TRIGGER or CREATE PROCEDURE statement. The change prevents the semi-colons terminating individual statements within the trigger or procedure from being interpreted as the termination of the CREATE statement itself.

A common setting for this option is '\'.

If the command delimiter is set to a string beginning with a character that is valid in identifiers, the command delimiter must be preceded by a space.

COMMIT_ON_EXIT

Controls whether a COMMIT or ROLLBACK is done when leaving ISQL.

ECHO

Controls whether commands are echoed before they are executed. This is most useful when using the READ command to execute an ISQL command file.

HEADINGS

Controls whether headings will be displayed for the results of a SELECT command.

INPUT_FORMAT

Sets the default data format expected by the INPUT command.

Certain file formats contain information about column names and types. Using this information, the INPUT command will create the database table if it does not already exist. This is a very easy way to load data into the database. The formats that have enough information to create the table are: DBASEII, DBASEIII, DIF, FOXPRO, LOTUS, and WATFILE.

Allowable input formats are:

ASCII Input lines are assumed to be ASCII characters, one row per line, with values separated by commas. Alphabetic strings may be enclosed in apostrophes (single quotes) or quotation marks (double quotes). Strings containing commas must be enclosed in either single or double quotes. If single or double quotes are used, double the quote character to use it within the string. Optionally, you can use the DELIMITED BY clause to specify a different delimiter string than the default which is a comma.

Three other special sequences are also recognized. The two characters \n represent a newline character, \\ represents a single \, and the sequence \xDD represents the character with hexadecimal code DD.

DBASE The file is in dBASE II or dBASE III format. ISQL will attempt to determine which of the two DBase formats the file is based on information in the file. If the table doesn't exist, it will be created.

DBASEII The file is in dBASE II format. If the table doesn't exist, it will be created.

DBASEIII The file is in dBASE III format. If the table doesn't exist, it will be created.

DIF Input file is in Data Interchange Format. If the table doesn't exist, it will be created.

FIXED Input lines are in fixed format. The width of the columns can be specified using the COLUMN WIDTHS clause. If they are not specified, then column widths in the file must be the same as the maximum number of characters required by any value of the corresponding database column's type.

FOXPRO The file is in FoxPro format (the FoxPro memo field is different than the dBASE memo field). If the table doesn't exist, it will be created.

LOTUS The file is a Lotus WKS format worksheet. INPUT assumes that the first row in the Lotus WKS format worksheet is column names. If the table doesn't exist, it will be created. In this case, the types and sizes of the columns created may not be correct because the information in the file pertains to a cell, not to a column.

WATFILE The input will be a WATFILE file. If the table doesn't exist, it will be created. WATFILE is a tabular file management tool available from WATCOM.

ISQL_LOG

If ISQL_LOG is set to a non-empty string, all ISQL commands are added to the end of the named file. Otherwise, if ISQL_LOG is set to the empty string ISQL commands are not logged.

NOTE: This is logging of an individual ISQL session only. See "Backup and Data Recovery" on page 331 for a description of the transaction log which logs all changes to the database by all users.

NULLS

Specifies how NULL values in the database will be displayed. The default is `'(NULL)'` (including the parentheses).

ON_ERROR

Controls what happens if an error is encountered while reading commands from a command file as follows:

STOP ISQL will stop reading commands from the file and return to the command window for input.

PROMPT
ISQL will prompt the user to see if the user wishes to continue.

CONTINUE
The error will be ignored and ISQL will continue reading commands from the command file. The INPUT command will continue with the next row, skipping the row that caused the error.

EXIT ISQL will terminate.

NOTIFY_CONTINUE
The error is displayed in a message box with a single Continue button. Execution continues once the button is clicked.

NOTIFY_STOP
The error is displayed in a message box with a single Stop button. Execution of the script stops once the button is clicked.

NOTIFY_EXIT
The error is displayed in a message box with a single Exit button. ISQL terminates once the button is clicked.

OUTPUT_FORMAT

Sets the output format for the data retrieved by the SELECT command and redirected into a file. This is also the default output format for the OUTPUT command. The valid output formats are:

ASCII The output is an ASCII format file with one row per line in the file. All values are separated by commas and strings are enclosed in apostrophes (single quotes). The delimiter and quote strings can be changed using the DELIMITED BY and QUOTE clauses. If ALL is specified in the QUOTE clause, then all values (not just strings) will be quoted.

Three other special sequences are also used. The two characters \n represent a newline character, \\ represents a single \, and the sequence \xDD represents the character with hexadecimal code DD. This is the default output format.

DBASEII The output is a dBASE II format file with the column definitions at the top of the file. Note that a maximum of 32 columns can be output. Also, note that columns longer than 255 characters will be truncated in the file.

DBASEIII The output is a dBASE III format file with the column definitions at the top of the file. Note that a maximum of 128 columns can be output. Also, note that columns longer than 255 characters will be truncated in the file.

DIF The output is a file in the standard Data Interchange Format.

FIXED The output is fixed format with each column having a fixed width. The width for each column can be specified using the COLUMN WIDTH clause. If this clause is omitted, the width for each column is computed from the data type for the column, and is large enough to hold any value of that data type. No column headings are output in this format.

FOXPRO The output is a FoxPro format file (the FoxPro memo field is different than the dBASE memo field) with the column definitions at the top of the file. Note that a maximum of 128 columns can be output. Also, note that columns longer than 255 characters will be truncated in the file.

LOTUS The output is a Lotus WKS format worksheet. Column names will be put as the first row in the worksheet. Note that there are certain restrictions on the maximum size of Lotus WKS format worksheets

that other software (such as Lotus 1-2-3) can load. There is no limit to the size of file ISQL can produce.

SQL The output is an ISQL INPUT command required to recreate the information in the table.

TEXT The output is a TEXT format file which prints the results in columns with the column names at the top and vertical lines separating the columns. This format is similar to that used to display data in the ISQL data window.

WATFILE The output is a WATFILE format file with the column definitions at the top of the file. WATFILE is a tabular file management tool available from WATCOM.

OUTPUT_LENGTH

Controls the length used when ISQL exports information to an external file (using output redirection or the OUTPUT command). The default for Output_length is 0—no truncation.

STATISTICS

Controls whether execution times, optimization strategies and other statistics will be displayed in the statistics window. This option can be set to 0, 3, 4, 5, or 6. When 0, the statistics window is not displayed. Otherwise, the value represents the height of the statistics window in lines.

TRUNCATION_LENGTH

When SELECT statement results are displayed on the screen, each column of output is limited to the width of the screen. The TRUNCATION_LENGTH option is used to reduce the width of wide columns so that more than one column will fit on the screen. A value of 0 means that columns will not be truncated.

Examples

Set the date format option.

```
SET OPTION public.date_format = 'Mmm dd yyyy' ;
```

Set the 'wait for commit' option to on.

```
SET OPTION wait_for_commit = 'on' ;
```

Embedded SQL examples

1. EXEC SQL SET OPTION :user.:option_name = :value;
2. EXEC SQL SET TEMPORARY OPTION Date_format = 'mm/dd/yyyy';

SET SQLCA Statement

Syntax

```
SET SQLCA sqlca

sqlca:          identifier or string
```

Purpose

To tell the SQL preprocessor to use a SQLCA other than the default global `sqlca`.

Usage

Embedded SQL.

Authorization

None.

Side effects

None.

See also

"Multi-Threaded or Reentrant Code" on page 575.

Description

The SET SQLCA statement tells the SQL preprocessor to use a SQLCA other than the default global `sqlca`. The **sqlca** must be an identifier or string that is a C language reference to a SQLCA pointer. The current SQLCA pointer is implicitly passed to the database interface library on every Embedded SQL command. All embedded SQL statements that follow this statement in the C source file will use the new SQLCA. This statement is only necessary when you are writing code that is reentrant (see "Multi-Threaded or Reentrant Code" on page 575). The **sqlca** should reference a local variable. Any global or module static variable is subject to being modified by another thread.

Example

The following function could be found in a Windows DLL. Each application that uses the DLL has its own SQLCA.

```
an_sql_code FAR PASCAL ExecuteSQL( an_application *app, char *com )
{
    EXEC SQL BEGIN DECLARE SECTION;
    char                *sqlcommand;
    EXEC SQL END DECLARE SECTION;

    EXEC SQL SET SQLCA "&app->sqlca";
    sqlcommand = com;
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL EXECUTE IMMEDIATE :sqlcommand;
    return( SQLCODE );
}
```

SET Statement

Syntax

```
SET identifier = expression
```

Purpose

To assign a value to a SQL variable.

Usage

Anywhere.

Authorization

None.

Side effects

None.

See also

CREATE VARIABLE Statement, DROP VARIABLE Statement, Expressions.

Description

The SET command assigns a new value to a variable that was previously created using the CREATE VARIABLE command.

A variable can be used in a SQL statement anywhere a column name is allowed. If there is no column name that matches the identifier, the database engine checks to see if there is a variable that matches and uses its value.

Variables are local to the current connection, and disappear when you disconnect from the database or when you use the DROP VARIABLE command. They are not affected by COMMIT or ROLLBACK statements.

Variables are necessary for creating large text or binary objects for INSERT or UPDATE statements from embedded SQL programs because embedded SQL host variables are limited to 32,767 bytes.

Example

The following code fragment could be used to insert a large text value into the database.

```
EXEC SQL BEGIN DECLARE SECTION;
char            buffer[5000];
EXEC SQL END DECLARE SECTION;

EXEC SQL CREATE VARIABLE hold_text LONG VARCHAR;

EXEC SQL SET hold_text = '';
for(;;) {
    /* read some data into buffer ... */
    size = fread( buffer, 1, 5000, fp );
    if( size <= 0 ) break;

    /* add data to blob using concatenation */
    EXEC SQL SET hold_text = hold_text || :buffer;
}

EXEC SQL INSERT INTO some_table VALUES ( 1, hold_text );

EXEC SQL DROP VARIABLE hold_text;
```

The following code fragment could be used to insert a large binary value into the database.

```
EXEC SQL BEGIN DECLARE SECTION;
DECL_BINARY( 5000 )   buffer;
EXEC SQL END DECLARE SECTION;

EXEC SQL CREATE VARIABLE hold_blob LONG BINARY;

EXEC SQL SET hold_blob = '';
for(;;) {
    /* read some data into buffer ... */
    size = fread( &(buffer.array), 1, 5000, fp );
    if( size <= 0 ) break;

    /* add data to blob using concatenation
       Note that concatenation works for binary data too! */
    EXEC SQL SET hold_blob = hold_blob || :buffer;
}

EXEC SQL INSERT INTO some_table VALUES ( 1, hold_blob );

EXEC SQL DROP VARIABLE hold_blob;
```

SIGNAL Statement

Syntax

```
SIGNAL exception-name
```

Purpose

Signal an exception condition.

Usage

Procedures, triggers, and batches only.

Authorization

None.

Side effects

None.

See also

"Using exception handlers in procedures and triggers" on page 249, RESIGNAL Statement, Compound Statements.

Description

SIGNAL allows you to raise an exception. See "Using exception handlers in procedures and triggers" on page 249 for a description of how exceptions are handled.

START DATABASE Statement

Syntax

START DATABASE database-file [AS database-name] [ON engine-name]

Purpose

To start a database on the specified database engine

Usage

ISQL

Authorization

None

Side effects

None

See Also

STOP DATABASE Statement, CONNECT Statement

Description

The START DATABASE command starts a specified database on a specified database engine. The database engine must be running. The full path must be specified for the database-file unless the file is located in the current directory.

The START DATABASE command does not connect ISQL to the specified database: a CONNECT command needs to be issued in order to make a connection.

If *database-name* is not specified, a default name is assigned to the database. This default name is the root of the database file. For example, a database in file **c:\sqlany50\sademo.db** would be given the default name of sample.

If *engine-name* is not specified, the default database engine is assumed. The default database engine is the first started engine among those currently running.

Example

Start the database file c:\sqlany50\sample_2.db on the current engine.

```
start database 'c:\sqlany50\sample_2.db'
```

Start the database file c:\sqlany50\sample_2.db as sam2 on the engine named sample.

```
START DATABASE 'c:\sqlany50\sample_2.db'
AS     sam2
ON     sample ;
```

START ENGINE Statement

Syntax

START ENGINE AS engine-name [STARTLINE command-string]

Purpose

To start a database engine

Usage

ISQL

Authorization

None

Side effects

None

See Also

STOP ENGINE Statement, "The database engine" on page 685.

Description

The START ENGINE command starts a database engine. If you wish to specify a set of options for the engine, use the STARTLINE keyword together with a command string. Valid command strings are those that conform to the database engine command line description in "SQL Anywhere Components" on page 679.

Example

Start a database engine, named sample, without starting any databases on it.

```
START ENGINE AS sample ;
```

Start a database engine with a maximum cache size of 4 megabytes, loading the sample database.

```
START ENGINE AS sample
STARTLINE 'dbeng50w -c 4096 c:\sqlany50\sademo.db'
```

START SUBSCRIPTION Statement

Syntax

START SUBSCRIPTION TO publication-name [(string)]

. . . FOR userid,. . .

Purpose

To start a subscription for a user to a publication.

Usage

Anywhere.

Authorization

Must have DBA authority.

Side effects

Automatic commit.

See also

"Synchronizing databases" on page 398, CREATE SUBSCRIPTION Statement, SYNCHRONIZE SUBSCRIPTION Statement.

Description

A SQL Remote subscription is said to be *started* when publication updates are being sent from the consolidated database to the remote database.

The START SUBSCRIPTION statement is one of a set of statements that manage subscriptions. The CREATE SUBSCRIPTION statement defines the data that the subscriber is to receive. The SYNCHRONIZE SUBSCRIPTION statement ensures that the consolidated and remote databases are consistent with each other. The START SUBSCRIPTION statement is required to start messages being sent to the subscriber.

Data at each end of the subscription must be consistent before a subscription is started. It is recommended that you use the database extraction utility to manage the creation, synchronization, and starting of subscriptions. If you use the database extraction utility, you do not need to execute an explicit START SUBSCRIPTION statement. Also, the Message Agent starts subscriptions once they are synchronized.

Example

The following statement starts the subscription of user **SamS** to the **pub_contact** publication.

```
START SUBSCRIPTION TO pub_contact
FOR  SamS
```

STOP DATABASE Statement

Syntax

STOP DATABASE database-name [ON engine-name] [UNCONDITIONALLY]

Purpose

To stop a database on the specified database engine

Usage

ISQL

Authorization

None

Side effects

None

See Also

START DATABASE Statement, DISCONNECT Statement

Description

The STOP DATABASE command stops a specified database on a specified database engine. If *engine-name* is not specified, all running engines will be searched for a database of the specified name.

If the UNCONDITIONALLY keyword is supplied, the database will be stopped even if there are connections to the database. If UNCONDITIONALLY is not specified, the database will not be stopped if there are connections to it.

Examples

Stop the database named sample on the default engine.

```
STOP DATABASE sample ;
```

STOP ENGINE Statement

Syntax

STOP ENGINE engine-name [UNCONDITIONALLY]

Purpose

To stop a database engine

Usage

ISQL

Authorization

None

Side effects

None

See Also

START ENGINE Statement

Description

The STOP ENGINE command stops the specified database engine. If the UNCONDITIONALLY keyword is supplied, the database engine will be stopped even if there are connections to the engine. If UNCONDITIONALLY is not specified, the database engine will not be stopped if there are connections to it.

Example

Stop the database engine named sample.

```
STOP ENGINE sample
```

STOP SUBSCRIPTION Statement

Syntax

STOP SUBSCRIPTION TO publication-name [(constant)]

. . . FOR userid,. . .

Purpose

To stop a subscription for a user to a publication.

Usage

Anywhere.

Authorization

Must have DBA authority.

Side effects

Automatic commit.

See also

"Synchronizing databases" on page 398, CREATE SUBSCRIPTION Statement, SYNCHRONIZE SUBSCRIPTION Statement.

Description

A SQL Remote subscription is said to be *started* when publication updates are being sent from the consolidated database to the remote database.

The STOP SUBSCRIPTION statement prevents any further messages being sent to the subscriber. The START SUBSCRIPTION statement is required to restart messages being sent to the subscriber. However, you should ensure that the subscription is properly synchronized before restarting: that no messages have been missed.

Example

The following statement starts the subscription of user **SamS** to the **pub_contact** publication.

```
STOP SUBSCRIPTION TO pub_contact
FOR  SamS
```

SYNCHRONIZE SUBSCRIPTION Statement

Syntax

SYNCHRONIZE SUBSCRIPTION TO publication-name [(string)]

. . . FOR userid,. . .

Purpose

To synchronize a subscription for a user to a publication.

Usage

Anywhere.

Authorization

Must have DBA authority.

Side effects

Automatic commit.

See also

"Synchronizing databases" on page 398, CREATE SUBSCRIPTION Statement, START SUBSCRIPTION Statement.

Description

A SQL Remote subscription is said to be *synchronized* when the data in the remote database is consistent with that in the consolidated database, so that publication updates sent from the consolidated database to the remote database will not result in conflicts and errors.

To synchronize a subscription, a copy of the data in the publication at the consolidated database is sent to the remote database. The SYNCHRONIZE SUBSCRIPTION statement does this through the message system. It is recommended that where possible you use the database extraction utility instead to synchronize subscriptions without using a message system.

Example

The following statement synchronizes the subscription of user **SamS** to the **pub_contact** publication.

```
SYNCHRONIZE SUBSCRIPTION TO pub_contact
FOR  SamS
```

SYSTEM Statement

Syntax

```
SYSTEM [ operating-system-command ]
```

Purpose

To execute an operating system command from within ISQL.

Usage

ISQL (DOS and QNX only).

Authorization

None.

Side effects

None.

See also

COMMIT Statement, CONNECT Statement.

Restrictions

- The SYSTEM command must be entirely contained on one line.
- Comments are not allowed at the end of a SYSTEM command.

Description

Executes the specified operating system command. If no command is specified, the DOS command interpreter or QNX shell is started. You can return to ISQL by using the DOS **exit** command or by pressing **Ctrl+D** in QNX.

Example

```
SYSTEM date
```

TRUNCATE TABLE Statement

Syntax

```
TRUNCATE TABLE [creator.]table-name
```

Purpose

To delete all rows from a table, without deleting the table definition.

Usage

Anywhere.

Authorization

Must have DELETE permission on the table.

Side effects

Automatic commit. Delete triggers are not fired by the TRUNCATE TABLE statement.

See also

"DELETE Statement" on page 885.

Description

The TRUNCATE TABLE statement deletes all rows from a table. It is equivalent to a DELETE statement without a WHERE clause, except that no triggers are fired as a result of the TRUNCATE TABLE statement and each individual row deletion is not entered into the transaction log.

After a TRUNCATE TABLE statement, the table structure and all of the indexes continue to exist until you issue a DROP TABLE statement. The column definitions and constraints remain intact, and triggers and permissions remain in effect.

The TRUNCATE TABLE statement is entered into the transaction log as a single statement, like data definition statements. Each deleted row is not entered into the transaction log.

The TRUNCATE TABLE statement can be used on views provided the SELECT statement defining the view has only one table in the FROM clause and does not contain a GROUP BY clause, an aggregate function, or involve a UNION operation.

Example

Delete all rows from the **department** table.

```
TRUNCATE TABLE department
```

UNION Operation

Syntax

```
select-without-order-by UNION [ALL] select-without-order-by
    ... [ UNION [ALL] select-without-order-by ] ...
    ... [ ORDER BY integer [ ASC | DESC ], ... ]
```

Purpose

To combine the results of two or more select statements.

Usage

Anywhere.

Authorization

Must have SELECT permission for each of the component SELECT commands.

Side effects

None.

See also

SELECT Statement.

Description

The results of several SELECT commands can be combined into a larger result using UNION. The component SELECT commands must each have the same number of items in the select list, and cannot contain an ORDER BY clause.

The results of UNION ALL is simply the combined results of the component SELECT commands. The results of UNION are the same as UNION ALL except that duplicate rows are eliminated. Since eliminating duplicates requires extra processing, UNION ALL should be used instead of UNION where possible.

If corresponding items in two select lists have different data types, SQL Anywhere will choose a data type for the corresponding column in the result and automatically convert the columns in each component SELECT command appropriately.

If ORDER BY is used, only integers are allowed in the order by list. These integers specify the position of the columns to be sorted.

The column names displayed are the same column names which would be displayed for the first SELECT statement.

Examples

List all distinct surnames of employees and customers.

```
SELECT emp_lname
FROM   Employee
UNION
SELECT lname
FROM   Customer ;
```

UNLOAD TABLE Statement

Syntax

```
UNLOAD [ FROM ] TABLE [ creator ].table-name
    ...    TO 'filename-string'
    ...    [ FORMAT 'ascii'] [ DELIMITED BY string ]
    ...    [ QUOTES on | off ] [ ESCAPES on | off ]
    ...    [ ORDER on | off ]
```

Purpose

To export data from a database table into an external ascii-format file.

Usage

Anywhere.

Authorization

Must have SELECT permission on the table.

Side effects

None.

See also

LOAD TABLE Statement, OUTPUT Statement.

Description

The UNLOAD TABLE statement allows efficient mass exporting from a database table into an ascii file. UNLOAD TABLE is more efficient than the ISQL statement OUTPUT and can be called from any client application.

UNLOAD TABLE places an exclusive lock on the whole table.

For descriptions of the FORMAT and DELIMITED BY options, see "LOAD TABLE Statement" on page 946. The other options are as follows:

QUOTES option

With QUOTES turned on (the default), single quotes are placed around all exported strings.

ESCAPES option

With ESCAPES on (the default), backslash-character combinations are used to identify special characters where necessary on export.

ORDER option

With ORDER on (the default) the data is exported ordered by primary key values. With ORDER off, the data is exported in the same order you see when selecting from the table without an ORDER BY clause.

Exporting is slower with ORDER on. However, reloading using the LOAD TABLE statement is quicker because of the simplicity of the indexing step.

UPDATE Statement

Syntax

1. UPDATE table-list

   ... SET column-name = expression, ...

   ... [FROM table-list]

   ... [WHERE search-condition]

   ... [ORDER BY expression [ASC | DESC] ,...]

2. UPDATE table-list

   ... SET column-name = expression, ...

   ... [VERIFY (column-name, ...) VALUES (expression, ...)]

   ... [WHERE search-condition]

   ... [ORDER BY expression [ASC | DESC] ,...]

Purpose

To modify data in the database.

Usage

Anywhere.

Authorization

Must have UPDATE permission for the columns being modified.

Side effects

None.

See also

DELETE Statement, INSERT Statement, "FROM Clause" on page 915.

Description

The UPDATE statement is used to modify rows of one or more tables. Each named column is set to the value of the expression on the right hand side of the equal sign. There are no restrictions on the **expression**. Even **column-name** can be used in the expression—the old value will be used. For a description of the table list and how to do joins, see "FROM Clause" on page 915.

Format 2 is intended for use with SQL Remote only, in single-row updates executed by the Message Agent. The VERIFY clause contains a set of values

that are expected to be present in the row being updated. If the values do not match, any RESOLVE UPDATE triggers are fired before the UPDATE proceeds. The UPDATE does not fail if the VERIFY clause fails to match.

If no WHERE clause is specified, every row will be updated. If a WHERE clause is specified, then only those rows which satisfy the search condition will be updated.

Normally, the order that rows are updated doesn't matter. However, in conjunction with the NUMBER(*) function, an ordering can be useful to get increasing numbers added to the rows in some specified order. Also, if you wish to do something like add 1 to the primary key values of a table, it is necessary to do this in descending order by primary key, so that you do not get duplicate primary keys during the operation.

Views can be updated provided the SELECT command defining the view does not contain a GROUP BY clause, an aggregate function, or involve a UNION operation.

Character strings inserted into tables are always stored in the case they are entered, regardless of whether the database is case sensitive or not. Thus a character data type column updated with a string **Value** is always held in the database with an upper-case V and the remainder of the letters lower case. SELECT statements return the string as **Value**. If the database is not case-sensitive, however, all comparisons make **Value** the same as **value**, **VALUE**, and so on. Further, if a single-column primary key already contains an entry **Value**, an INSERT of **value** is rejected, as it would make the primary key not unique.

The optional FROM clause allows tables to be updated based on joins. If the FROM clause is present, the WHERE clause qualifies the rows of the FROM clause. Data is updated only in the table list immediately following the UPDATE keyword.

For a full description of the FROM clause and joins, see "FROM Clause" on page 915.

Examples

Transfer employee Philip Chin (employee 129) from the sales department to the marketing department.

```
UPDATE employee
SET    dept_id = 400
WHERE  emp_id = 129 ;
```

Sales orders currently start at id 2001. Renumber all existing sales orders by subtracting 2000 from the id.

```
UPDATE sales_order_items AS items ,
       sales_order AS orders
SET    items.id = items.id - 2000,
       orders.id = orders.id - 2000 ;
```

UPDATE (positioned) Statement

Syntax

Format 1:

```
UPDATE WHERE CURRENT OF cursor-name

   ...     | USING DESCRIPTOR sqlda-name                    |
           | FROM host-variable-list                        |
```

Format 2:

```
UPDATE table-list SET

   ... column-name = expression, ...

   ... WHERE CURRENT OF cursor-name
```

host-variable-list:	indicator variables allowed
sqlda-name:	identifier

Purpose

To modify the data at the current location of a cursor.

Usage

Embedded SQL, procedures, triggers, and batches. The USING DESCRIPTOR, FROM *host-variable-list*, and *host-variable* formats are for Embedded SQL only.

Authorization

Must have UPDATE permission on the columns being modified.

Side effects

None.

See also

UPDATE Statement, DELETE Statement, DELETE (positioned) Statement, INSERT Statement, FETCH Statement.

Description

This form of the UPDATE statement updates the current row of the specified cursor. The current row is defined to be the last row FETCHed from the cursor and the last operation on the cursor must not have been a DELETE Positioned.

For format 1, columns from the SQLDA or values from the host variable list correspond one-to-one with the select list items of the specified cursor. If the

sqldata pointer in the SQLDA is the null pointer, then the corresponding select list item is not updated.

In format 2, the requested columns are set to the specified values for the row at the current row of the specified query. The columns do not need to be in the select list of the specified open cursor. This format can be PREPAREd.

Example

```
UPDATE Employee
SET    emp_lname = 'Jones'
WHERE  CURRENT OF emp_cursor ;
```

VALIDATE TABLE Statement

Syntax

VALIDATE TABLE [creator.]table-name

Purpose

To validate a table in the database.

Usage

Anywhere.

Authorization

Must be the creator of the table or have DBA authority.

Side effects

None.

See also

DBVALID

Description

The VALIDATE TABLE command will scan every row of a table and look each row up in each index on the table. If the database file is corrupt, an error will be reported. This should not happen. However, because DOS and Windows are unprotected operating environments, other software can corrupt memory used by the database engine. This problem may be detected through software errors or crashes, or the corrupt memory could get written to the database creating a corrupt database file. Also, in any operating system, hardware problems with the disk could cause the database file to get corrupted. If you cannot determine how a database file became corrupted, please report the problem to WATCOM.

If you do have errors reported, you can drop all of the indexes and keys on a table and recreate them. Any foreign keys to the table will also need to be recreated. Another solution to errors reported by VALIDATE TABLE is to unload and reload your entire database. You should use the `-u` option of DBUNLOAD so that it will not try to use a possibly corrupt index to order the data.

WHENEVER Statement

Syntax

```
| WHENEVER SQLERROR                                    |
| WHENEVER SQLWARNING                                  |
| WHENEVER NOTFOUND                                    |

        | GOTO label                                     |
...     | STOP                                           |
        | CONTINUE                                       |
        | { C code; }                                    |

label:              identifier
```

Purpose

To specify error handling in an Embedded SQL program.

Usage

Embedded SQL.

Authorization

None.

Side effects

None.

Description

The WHENEVER statement is used to trap errors, warnings and exceptional conditions encountered by the database when processing SQL commands. The statement can be put anywhere in an Embedded SQL C program and does not generate any code. The preprocessor will generate code following each successive SQL statement. The error action remains in effect for all Embedded SQL statements from the source line of the WHENEVER statement until the next WHENEVER statement with the same error condition, or the end of the source file.

NOTE: The error conditions are in effect based on positioning in the C language source file and not on when the statements are executed.

The default action is CONTINUE.

Note that this statement is provided for convenience in simple programs. Most of the time, checking the *sqlcode* field of the SQLCA (SQLCODE) directly is the easiest way to check error conditions. In this case, the WHENEVER statement would not be used. If fact, all the WHENEVER statement does is

cause the preprocessor to generate an `if( SQLCODE )` test after each statement.

Examples

```
1.  EXEC SQL WHENEVER NOTFOUND GOTO done;

2.  EXEC SQL WHENEVER SQLERROR
              { PrintError( &sqlca ); return( FALSE ); };
```

Chapter 39

SQL Anywhere Database Error Messages

About this chapter

This chapter lists all database error messages reported by SQL Anywhere. Many of the errors contain the characters `%1`, `%2` and so on. These are replaced by the parameters to the error message.

Each error has a numeric error code, called the SQLCODE. Negative codes are considered errors while positive codes are warnings. The SQLCODE 0 indicates successful completion.

The error descriptions are listed in order by their SQLCODE value. To find an error message description if you do not have the SQLCODE value, look up the SQLCODE in one of the indexes provided. The chapter contains indexes alphabetically by message and by SQLSTATE value.

Contents

39.1 Error messages: alphabetic listing by message

This section provides an alphabetic listing of error messages, and their corresponding SQLCODE values. For a fuller description of the error, look up the SQLCODE value here and then look up the description in the section "Error message descriptions" on page 1046, which is ordered by SQLCODE.

SQL Code	*Error message*
-210	%1 has the row in %2 locked
-134	%1 not implemented
-110	'%1' already exists
-120	'%1' already has grant permission
-284	'%1' is already the publisher for this database
-140	'%1' is an unknown userid
-285	'%1' is not a remote user for this database
-123	'%1' is not a user group
0	(no message)
-624	A parameter to an external function is an unsupported datatype
-150	aggregate functions not allowed on this statement
-307	all threads are blocked
-125	ALTER clause conflict
-407	an argument passed to a SQL Anywhere HLI function was invalid
-298	attempted two active database requests
-98	authentication violation
-617	Calling functions outside the database engine are not supported
-160	can only describe a SELECT statement
-127	cannot alter a column in an index
-105	cannot be started -- %1
-157	cannot convert %1 to a %2
-269	cannot delete a column referenced in a trigger definition
-614	cannot drop a user that owns messages or datatypes
-270	cannot drop a user that owns procedures in runtime engine
-128	cannot drop a user that owns tables in runtime engine
-183	cannot find index named '%1'
-191	cannot modify column '%1' in table '%2'
-106	cannot open log file %1
-162	Cannot outer join a view with a UNION or GROUP BY
-295	cannot uniquely identify rows in cursor
-190	cannot update an expression
-212	CHECKPOINT command requires a rollback log
-88	client/server communications protocol mismatch
-113	column %1 in foreign key has a different definition than primary key

-144	column '%1' found in more than one table -- need a correlation name
-195	column '%1' in table '%2' cannot be NULL
-143	column '%1' not found
-267	COMMIT/ROLLBACK not allowed within atomic operation
-273	COMMIT/ROLLBACK not allowed within trigger actions
-73	communication buffer underflow
-85	communication error
-108	connection not found
-99	connections to database have been disabled
-142	correlation name '%1' not found
-622	Could not allocate resources to call external function
-621	Could not find the named function in the dynamic library
-620	Could not load the dynamic library
-172	cursor already open
-170	cursor has not been declared
-180	cursor not open
-623	data definition statements not allowed in procedures or triggers
-241	database backup not started
-96	database engine already running
-89	database engine not running in multi-user mode
-100	database engine not running
-77	database name not unique
-87	database name required to start engine
-266	database was initialized with an older version of the software
-97	database's page size too big
-231	dblib/database engine version mismatch
-138	dbspace '%1' not found
-306	deadlock detected
-304	disk full -- transaction rolled back
-121	do not have permission to %1
-78	Dynamic memory exhausted!
-184	error inserting into cursor
-296	error number %1 for RAISERROR is less than 17000
-171	error opening cursor
-107	error writing to log file
-251	foreign key '%1' for table '%2' duplicates an existing foreign key
-145	foreign key name '%1' not found
-287	format string argument number %1 is invalid
-149	function or column reference to '%1' in the select list must also appear in a GROUP BY
-305	I/O error %1 -- transaction rolled back
-250	identifier '%1' too long
-242	incomplete transactions prevent transaction log renaming
-196	index '%1' for table '%2' would not be unique

-111	index name '%1' not unique
-199	INSERT/DELETE on cursor can modify only one table
-301	internal database error %1 -- transaction rolled back
-263	invalid absolute or relative offset in FETCH
-159	invalid column number
103	invalid data conversion
-81	invalid database engine command line
-609	invalid datatype for column in WRITETEXT or READTEXT
-156	invalid expression near '%1'
-155	invalid host variable
-79	invalid local database switch
-187	invalid operation for this cursor
-192	invalid operation on joined tables
-200	invalid option '%1' -- no PUBLIC setting exists
-95	invalid parameter
-133	invalid prepared statement type
-272	invalid REFERENCES clause in trigger definition
-201	invalid setting for option '%1'
-405	invalid SQL Anywhere HLI callback function
-400	invalid SQL Anywhere HLI command syntax
-401	invalid SQL Anywhere HLI cursor name
-403	invalid SQL Anywhere HLI host variable name
-404	invalid SQL Anywhere HLI host variable value
-402	invalid SQL Anywhere HLI statement name
-130	invalid statement
-608	invalid TEXTPTR value used with WRITETEXT or READTEXT
-161	invalid type on DESCRIBE statement
-104	invalid userid and password on preprocessed module
-103	invalid userid or password
-209	invalid value for column '%1' in table '%2'
-262	label '%1' not found
-135	language extension
-618	Mismatch between external function platform specifier and current operating system
-139	more than one table is identified as '%1'
-619	Need a dynamic library name
-197	no current row of cursor
-181	no indicator variable provided for NULL result
-194	no primary key value for foreign key '%1' in table '%2'
-211	not allowed while %1 is using the database
-101	not connected to SQL database
-182	not enough fields allocated in SQLDA
-86	not enough memory to start
-188	not enough values for host variables
-152	number in ORDER BY is too large

-114	number of columns does not match SELECT
-122	operation would cause a group cycle
-615	parameter '%1' not found in procedure '%2'
-287	passthrough statement inconsistent with current passthrough
-119	primary key column '%1' already defined
-198	primary key for row in table '%1' is referenced in another table
-193	primary key for table '%1' is not unique
-265	procedure '%1' not found
105	procedure has completed
-215	procedure in use
-274	procedure or trigger calls have nested too deeply
-280	publication '%1' not found
-286	remote message type '%1' not found
-76	request denied -- no active databases
-75	request to start/stop database denied
-222	result set not allowed from within an atomic compound statement
-221	ROLLBACK TO SAVEPOINT not allowed
104	row has been updated since last time read
-208	row has changed since last read -- operation cancelled
100	row not found
-300	run time SQL error -- %1
-220	savepoint '%1' not found
-213	savepoints require a rollback log
-153	SELECT lists in UNION do not match in length
-185	SELECT returns more than one row
-232	server/database engine version mismatch
-84	specified database is invalid
-83	specified database not found
-406	SQL Anywhere HLI internal error
-132	SQL statement error
-230	sqlpp/dblib version mismatch
-299	statement interrupted by user
-151	subquery allowed only one select list item
-186	subquery cannot return more than one result
-282	subscription to '%1' for '%2' already exists
-283	subscription to '%1' for '%2' not found
-131	syntax error near '%1'
-118	table '%1' has no primary key
-281	table '%1' has publications
-136	table '%1' is in an outer join cycle
-141	table '%1' not found
-137	table '%1' requires a unique correlation name
-112	table already has a primary key
-126	table cannot have two primary keys
-214	table in use

-116	table must be empty
-302	terminated by user -- transaction rolled back
-74	the selected database is currently inactive
400	the supplied buffer was too small to hold all requested query results
-109	there are still active database connections
-261	there is already a variable named '%1'
-147	there is more than one way to join '%1' to '%2'
-146	there is no way to join '%1' to '%2'
-616	too many columns in table
-102	too many connections to database
-625	Too many parameters to this external procedure call
-611	Transact SQL feature not supported
-268	trigger '%1' not found
-271	trigger definition conflicts with existing triggers
-275	Triggers and procedures not supported in desktop engine
-243	unable to delete database file
-189	unable to find in index '%1' for table '%2'
-80	unable to start database engine
-82	unable to start specified database
-240	unknown backup operation
-148	unknown function '%1'
-610	user message %1 already exists
-612	user message %1 not found
-297	user-defined exception signalled
-613	user-defined type %1 not found
102	using temporary table
-158	value %1 too large for destination
106	value for column '%1' in table '%2' has changed
101	value truncated
-260	variable '%1' not found
200	warning
-154	wrong number of parameters to function '%1'
-207	wrong number of values for INSERT
-264	wrong number of variables in FETCH

39.2 Error messages: listing by SQLSTATE

SQL Anywhere supports the SQLSTATE error code, defined by the ISO/ANSI SQL/92 standard. Each SQLSTATE value is a five character string containing a two character class followed by a three character subclass. Each character can be one of the uppercase letters A through Z or the digits 0 through 9. A class that begins with A through H or 0 through 4 has been defined by the ANSI standard; other classes are implementation defined. Similarly, subclasses of standard classes that start with the same characters (A-H, 0-4) are standard. The

subclass 000 always means that no subclass code is defined. The most common SQLSTATE value is 00000 which indicates successful completion.

In SQL Anywhere, there are two ways to get SQLSTATE values. In Embedded SQL, the SQLSTATE is returned in the SQLCA. In ODBC, the *SQLError* function returns SQLSTATE values. Since the ODBC definition explicitly specifies which SQLSTATE values are to be returned, some ODBC SQLSTATE values correspond to more than one SQL Anywhere error condition. Consequently, many error conditions return one SQLSTATE value in Embedded SQL which precisely indicates the type of error, and return another less specific SQLSTATE value in ODBC. If the ODBC SQLSTATE value is different than that in Embedded SQL, it is explicitly specified in the error list.

In Embedded SQL, the SQLSTATE values are defined as constants in the file **sqlstate.h** in the **h** subdirectory. The descriptions below indicate the constant name for the SQLCODE for each state which begins with the characters 'SQLE_'. The corresponding SQLSTATE values begin with the characters 'SQLSTATE_'.

The SQLSTATE values used by SQL Anywhere have been derived from a number of sources. The class of the error is determined by:

1. Using a class defined by SQL-92 if it exists
2. Otherwise, using a class defined by IBM's SAA (starts with 5)
3. Otherwise, using a class defined by Watcom (starts with W)

The same rules are used for defining subclasses. In other words, a class or subclass starting with 0-4 or A-H are defined by ANSI, starting with 5 are defined by IBM, and starting with W are defined by Watcom. Similarly, the ODBC SQLSTATE classes and subclasses starting with S have been defined by Microsoft in the ODBC specification.

This section provides an alphabetic listing of SQLSTATE values, and their corresponding SQLCODE values. For a fuller description of the error, look up the SQLCODE value here and then look up the description in the section "Error message descriptions" on page 1046, which is ordered by SQLCODE.

SQL Code	*SQLSTATE*
0	00000
200	01000
101	01004
102	01W02
103	01W03
104	01W04
105	01W05
106	01W06

400	01WH1
100	02000
-188	07001
-182	07002
-171	07003
-160	07005
-161	07W01
-130	07W02
-133	07W03
-105	08001
-101	08003
-140	08004
-100	08W01
-108	08W02
-102	08W03
-99	08W04
-106	08W05
-109	08W06
-80	08W07
-81	08W08
-82	08W09
-83	08W10
-84	08W11
-85	08W12
-86	08W13
-87	08W14
-88	08W15
-89	08W16
-107	08W17
-230	08W18
-231	08W19
-232	08W20
-98	08W21
-97	08W22
-96	08W23
-95	08W24
-79	08W25
-78	08W26
-77	08W27
-76	08W28
-75	08W29
-74	08W30
-73	08W31
-184	09W01
-187	09W02

-197	09W03
-199	09W04
-295	09W05
-134	0A000
-135	0AW01
-611	0AW02
-275	0AW02
-185	21000
-186	21W01
-181	22002
-158	22003
-407	22W01
-208	22W02
-608	22W03
-195	23502
-194	23503
-196	23505
-209	23506
-193	23W01
-198	23W05
-180	24501
-172	24502
-170	24W01
-132	26501
-402	26W01
-103	28000
-104	28W01
-273	2D501
-401	34W01
-154	37505
-220	3B001
-221	3B002
-213	3BW01
-222	3BW02
-300	40000
-306	40001
-301	40W01
-302	40W02
-304	40W03
-305	40W04
-307	40W06
-121	42501
-120	42W01
-122	42W02
-123	42W03

-131	42W04
-148	42W05
-150	42W06
-155	42W07
-156	42W08
-403	42W09
-404	42W10
-400	42W11
-405	42W12
-159	42W13
-260	42W14
-261	42W15
-200	42W16
-201	42W17
-210	42W18
-211	42W19
-212	42W20
-214	42W21
-298	42W22
-215	42W23
-262	42W24
-263	42W25
-264	42W26
-266	42W27
-267	42W28
-274	42W29
-615	42W30
-144	52002
-143	52003
-119	52009
-110	52010
-139	52012
-141	52W01
-142	52W02
-183	52W03
-111	52W04
-126	52W05
-251	52W06
-145	52W07
-147	52W08
-265	52W09
-268	52W10
-271	52W11
-272	52W12
-138	52W13

-136	52W14
-137	52W15
-610	52W16
-612	52W17
-613	52W18
-162	52W19
-616	52W20
-623	52W21
-207	53002
-149	53003
-152	53005
-191	53008
-114	53011
-157	53018
-151	53023
-153	53026
-113	53030
-125	53W01
-190	53W02
-192	53W03
-146	53W04
-127	53W05
-269	53W06
-296	53W07
-287	53W08
-609	53W09
-250	54003
-118	55008
-112	55013
-116	55W02
-128	55W03
-270	55W04
-614	55W05
-299	57014
-280	5RW01
-281	5RW02
-282	5RW03
-283	5RW04
-284	5RW05
-285	5RW06
-286	5RW07
-287	5RW08
-297	99999
-240	WB001
-241	WB002

-242	WB003
-243	WB004
-189	WI005
-406	WI007
-617	WW003
-618	WW004
-619	WW005
-620	WW006
-621	WW007
-622	WW008
-624	WW009
-625	WW010

39.3 Error message descriptions

This section lists each of the error messages, with a description of a probable cause. The messages are arranged in order by SQLCODE value.

Negative codes are errors and positive codes are warnings. A SQLCODE value of 0 indicates successful completion.

Too many parameters to this external procedure call

SQL Code	-625
Constant	SQLE_TOO_MANY_PARAMETERS
SQL State	WW010
ODBC state	S1000
Probable cause	This is a Windows 32-bit specific error. There is a maximum of 256 parameters to an external function call.

A parameter to an external function is an unsupported datatype

SQL Code	-624
Constant	SQLE_DATATYPE_NOT_ALLOWED
SQL State	WW009
ODBC state	S1000
Probable cause	A parameter to a call to an external function is declared to be a datatype that is not supported.

data definition statements not allowed in procedures or triggers

SQL Code	-623
Constant	SQLE_DDL_NOT_ALLOWED_IN_PROCEDURES
SQL State	52W21
ODBC state	S1000

Probable cause The procedure or trigger definition contains a data definition statement (e.g. CREATE, DROP, GRANT, REVOKE, ALTER). These statements are not allowed within procedures or triggers.

Could not allocate resources to call external function

SQL Code -622
Constant SQLE_ERROR_CALLING_FUNCTION
SQL State WW008
ODBC state S1000
Probable cause The external function could not be called due to a shortage of operating system resources. If the operating system supports threads, the maximum thread count should be increased.

Could not find the named function in the dynamic library

SQL Code -621
Constant SQLE_COULD_NOT_FIND_FUNCTION
SQL State WW007
ODBC state S1000
Probable cause The external function could not be found in the dynamic library.

Could not load the dynamic library

SQL Code -620
Constant SQLE_COULD_NOT_LOAD_LIBRARY
SQL State WW006
ODBC state S1000
Probable cause The library named in an external function call could not be loaded.

Need a dynamic library name

SQL Code -619
Constant SQLE_REQUIRE_DLL_NAME
SQL State WW005
ODBC state S1000
Probable cause The name of the external function to call did not contain a library name specifier.

Mismatch between external function platform specifier and current operating system

SQL Code -618

Constant	SQLE_EXTERNAL_PLATFORM_FAILURE
SQL State	WW004
ODBC state	S1000
Probable cause	A call to an external entry point in a dynamically loaded module was qualified by an operating system which was not the operating system on which the engine/server is currently executing.

Calling functions outside the database engine are not supported

SQL Code	-617
Constant	SQLE_EXTERNAL_CALLS_NOT_SUPPORTED
SQL State	WW003
ODBC state	S1000
Probable cause	An attempt was made to call a stored procedure that, inturn, calls a function is a dynamically loaded module. The operating system on which this stored procedure was called does not support such an action.

too many columns in table

SQL Code	-616
Constant	SQLE_TOO_MANY_COLUMNS_IN_TABLE
SQL State	52W20
ODBC state	S1000
Probable cause	A CREATE TABLE or ALTER TABLE statement attempted to add a column to a table, but the resulting number of columns in the table would exceed the limit for the current database page size.

parameter '%1' not found in procedure '%2'

SQL Code	-615
Constant	SQLE_INVALID_PARAMETER_NAME
SQL State	42W30
ODBC state	37000
Probable cause	The procedure parameter name does not match a parameter for this procedure. Check the spelling of the parameter name.

cannot drop a user that owns messages or datatypes

SQL Code	-614
Constant	SQLE_USER_OWNS_MESSAGES_OR_DATATYPES
SQL State	55W05
ODBC state	37000

Probable cause A user to be dropped is the creator of a message or user-defined datatype. The message or user-defined datatype must be dropped first.

user-defined type %1 not found

SQL Code -613
Constant SQLE_USER_TYPE_NOT_FOUND
SQL State 52W18
ODBC state S0002
Parameter Name of the user-defined type.
Probable cause The user-defined type with this name does not exist in SYSUSERTYPE.

user message %1 not found

SQL Code -612
Constant SQLE_MESSAGE_NOT_FOUND
SQL State 52W17
ODBC state S0002
Parameter Message number.
Probable cause The message with this error number does not exist in SYSUSERMESSAGES.

Transact SQL feature not supported

SQL Code -611
Constant SQLE_TSQL_FEATURE_NOT_SUPPORTED
SQL State 0AW02
ODBC state 37000
Probable cause An attempt was made to use a feature of Transact SQL that is not supported.

user message %1 already exists

SQL Code -610
Constant SQLE_MESSAGE_ALREADY_EXISTS
SQL State 52W16
ODBC state 23000
Probable cause The message with this error number already exists in SYSUSERMESSAGES.

invalid datatype for column in WRITETEXT or READTEXT

SQL Code -609

Constant	SQLE_INVALID_TEXT_IMAGE_DATATYPE
SQL State	53W09
ODBC state	S1000
Probable cause	The column referenced in a WRITETEXT or READTEXT statement is not defined for storing text or image data.

invalid TEXTPTR value used with WRITETEXT or READTEXT

SQL Code	-608
Constant	SQLE_INVALID_TEXTPTR_VALUE
SQL State	22W03
ODBC state	S1000
Probable cause	The value supplied as the TEXTPTR for a WRITETEXT or READTEXT statement is invalid.

an argument passed to a SQL Anywhere HLI function was invalid

SQL Code	-407
Constant	SQLE_HLI_BAD_ARGUMENT
SQL State	22W01
Probable cause	One of the arguments passed to a WSQL HLI function was invalid. This may indicate that a pointer to a command string or result buffer is the null pointer.

SQL Anywhere HLI internal error

SQL Code	-406
Constant	SQLE_HLI_INTERNAL
SQL State	WI007
Probable cause	This is a SQL Anywhere internal error and should be reported to Watcom.

invalid SQL Anywhere HLI callback function

SQL Code	-405
Constant	SQLE_HLI_BAD_CALLBACK
SQL State	42W12
Probable cause	WSQL HLI needed to use a callback function, but the function has not been registered using the *wsqlregisterfuncs* entry point.

invalid SQL Anywhere HLI host variable value

SQL Code	-404
Constant	SQLE_HLI_BAD_HOST_VAR_VALUE
SQL State	42W10

Probable cause You have used a host variable, and the host variable value is too long.

invalid SQL Anywhere HLI host variable name

SQL Code -403
Constant SQLE_HLI_BAD_HOST_VAR_NAME
SQL State 42W09
Probable cause You have used a host variable, and the host variable callback function does not recognize it.

invalid SQL Anywhere HLI statement name

SQL Code -402
Constant SQLE_HLI_BAD_STATEMENT
SQL State 26W01
Probable cause The statement name indicated in your command is not a valid one. This typically indicates that you have failed to prepare the statement.

invalid SQL Anywhere HLI cursor name

SQL Code -401
Constant SQLE_HLI_BAD_CURSOR
SQL State 34W01
Probable cause The cursor name indicated in your command is not a valid one. For instance, this error would occur if you tried to close a cursor that had never even been declared.

invalid SQL Anywhere HLI command syntax

SQL Code -400
Constant SQLE_HLI_BAD_SYNTAX
SQL State 42W11
Probable cause The command string that you sent to *wsqlexec* cannot be understood. Make sure that all of the keywords in the command string are spelled properly, and that variable names (such as host variable, cursor or statement names) are not too long.

all threads are blocked

SQL Code -307
Constant SQLE_THREAD_DEADLOCK
SQL State 40W06

ODBC state 40001
Probable cause You have attempted to read or write a row and it is locked by another user. Also, all other threads (see database option THREAD_COUNT) are blocked waiting for a lock to be released. This is a deadlock situation and your transaction has been chosen as the one to rollback.

deadlock detected

SQL Code -306
Constant SQLE_DEADLOCK
SQL State 40001
Probable cause You have attempted to read or write a row and it is locked by another user. Also, the other user is blocked directly or indirectly on your own transaction. This is a deadlock situation and your transaction has been chosen as the one to rollback.

I/O error %1 -- transaction rolled back

SQL Code -305
Constant SQLE_DEVICE_ERROR
SQL State 40W04
ODBC state S1000
Probable cause SQL Anywhere has detected a problem with your hard disk. If you cannot find a hardware error using the operating system disk check utility (eg. in DOS, **chkdsk**, and in QNX, **chkfsys**), report the problem to Watcom. A ROLLBACK WORK command has been automatically executed.

disk full -- transaction rolled back

SQL Code -304
Constant SQLE_DEVICE_FULL
SQL State 40W03
ODBC state S1000
Probable cause Your hard disk is out of free space. A ROLLBACK WORK command has been automatically executed.

terminated by user -- transaction rolled back

SQL Code -302
Constant SQLE_TERMINATED_BY_USER
SQL State 40W02
ODBC state S1000

Probable cause The user has aborted a command while the database was executing. A ROLLBACK WORK command has been automatically executed. This will happen when the engine is running in bulk mode and the user aborts an INSERT, UPDATE, or DELETE operation.

internal database error %1 -- transaction rolled back

SQL Code -301
Constant SQLE_DATABASE_ERROR
SQL State 40W01
ODBC state S1000
Parameter Identification of the error.
Probable cause This error indicates an internal database error, and should be reported to Watcom. A ROLLBACK WORK command has been automatically executed.

run time SQL error -- %1

SQL Code -300
Constant SQLE_ERROR
SQL State 40000
ODBC state S1000
Parameter Identification of the error.
Probable cause This error indicates an internal database error, and should be reported to Watcom.

statement interrupted by user

SQL Code -299
Constant SQLE_INTERRUPTED
SQL State 57014
ODBC state S1000
Probable cause The user has aborted a statement during its execution. The database was able to stop the operation without doing a rollback. If the statement is INSERT, UPDATE, or DELETE, any changes made by the statement will be cancelled.

If the statement is a data definition command (for example CREATE TABLE), the command will be cancelled, but the COMMIT that was done as a side effect will not be cancelled.

attempted two active database requests

SQL Code -298
Constant SQLE_DOUBLE_REQUEST

SQL State 42W22
ODBC state S1000
Probable cause In Embedded SQL, you have attempted to submit a database request while you have another request in process. This often occurs in Windows when processing the WM_PAINT message causes a database request, and you get a second WM_PAINT before the database request has completed.

user-defined exception signalled

SQL Code -297
Constant SQLE_USER_DEFINED_EXCEPTION
SQL State 99999
ODBC state S1000
Probable cause A stored procedure or trigger signalled a user-defined exception. This error state is reserved for use within stored procedures or triggers which contain exception handlers, as a way of signalling an exception which can be guaranteed to not have been caused by the database engine.

error number %1 for RAISERROR is less than 17000

SQL Code -296
Constant SQLE_ERROR_NUMBER_OUT_OF_RANGE
SQL State 53W07
ODBC state 37000
Parameter Error number.
Probable cause The error number used in a RAISERROR statement is invalid. The number must be greater than or equal to 17000.

cannot uniquely identify rows in cursor

SQL Code -295
Constant SQLE_CANNOT_UNIQUELY_IDENTIFY_ROWS
SQL State 09W05
ODBC state 24000
Probable cause A UNIQUE cursor has been opened on a SELECT statement for which a set of columns uniquely identifying each row cannot be generated. One of the tables may not be defined with a primary key or uniqueness constraint, or the SELECT statement may involve a UNION or GROUP BY.

format string argument number %1 is invalid

SQL Code -287

Constant	SQLE_INVALID_FORMAT_STRING_ARG_NUM
SQL State	53W08
ODBC state	37000
Parameter	Argument number.
Probable cause	An argument number in the format string for a PRINT or RAISERROR statement is invalid. The number must be between 1 and 20 and must not exceed the number of arguments provided.

remote message type '%1' not found

SQL Code	-286
Constant	SQLE_NOT_REMOTE_TYPE
SQL State	5RW07
ODBC state	S0002
Parameter	Name of remote message type.
Probable cause	You have refered to a remote message type that is not defined in this database. CREATE REMOTE TYPE is used to define remote message types.

'%1' is not a remote user for this database

SQL Code	-285
Constant	SQLE_NOT_REMOTE_USER
SQL State	5RW06
ODBC state	S0002
Parameter	Name of user.
Probable cause	You have tried to CREATE a subscription for a user, or PASSTHROUGH for a user that is not a remote user of this database. You must GRANT REMOTE or GRANT CONSOLIDATE.

'%1' is already the publisher for this database

SQL Code	-284
Constant	SQLE_ONLY_ONE_PUBLISHER
SQL State	5RW05
ODBC state	S0002
Parameter	Name of the publisher.
Probable cause	You have tried to GRANT PUBLISH to a userid, when a publisher already exists.

subscription to '%1' for '%2' not found

SQL Code	-283

Constant	SQLE_SUBSCRIPTION_NOT_FOUND
SQL State	5RW04
ODBC state	S0002
Parameter	Name of the publication.
Parameter	Name of the user.
Probable cause	You have tried to drop, start, or synchronize a subscription that does not exist.

subscription to '%1' for '%2' already exists

SQL Code	-282
Constant	SQLE_SUBSCRIPTION_NOT_UNIQUE
SQL State	5RW03
ODBC state	S0002
Parameter	Name of the publication.
Parameter	Name of the user.
Probable cause	You have tried to create a subscription that already exists.

table '%1' has publications

SQL Code	-281
Constant	SQLE_TABLE_HAS_PUBLICATIONS
SQL State	5RW02
ODBC state	S0002
Parameter	Name of the publication that has publications.
Probable cause	You have attempted to drop a table that has publications defined.

publication '%1' not found

SQL Code	-280
Constant	SQLE_PUBLICATION_NOT_FOUND
SQL State	5RW01
ODBC state	S0002
Parameter	Name of the publication that could not be found.
Probable cause	You have misspelled the name of a publication, or you have connected with a different userid and forgotten to qualify a publication name with a user name.

Triggers and procedures not supported in desktop engine

SQL Code	-275
Constant	SQLE_PROCEDURES_NOT_IN_DESKTOP
SQL State	0AW02
ODBC state	S1000

Probable cause You have attempted to call a stored procedure or have modified a row in a table on which a trigger is defined and you are using the desktop engine. Triggers and stored procedures are not supported in the desktop engine. You must be running the full engine to use these features.

procedure or trigger calls have nested too deeply

SQL Code -274
Constant SQLE_NESTING_TOO_DEEP
SQL State 42W29
ODBC state 37000
Probable cause You have probably defined a procedure or trigger that causes unlimited recursion.

COMMIT/ROLLBACK not allowed within trigger actions

SQL Code -273
Constant SQLE_INVALID_TRIGGER_STATEMENT
SQL State 2D501
ODBC state 37000
Probable cause An attempt was made to execute a statement that is not allowed while performing a trigger action. COMMIT and ROLLBACK statements cannot be executed from a trigger.

invalid REFERENCES clause in trigger definition

SQL Code -272
Constant SQLE_INVALID_TRIGGER_COL_REFS
SQL State 52W12
ODBC state 37000
Probable cause The REFERENCES clause in a trigger definition is invalid. An OLD correlation name may have been specified in a BEFORE INSERT trigger, or a NEW correlation name may have been specified in an AFTER DELETE trigger. In both cases, the values do not exist and cannot be referenced.

trigger definition conflicts with existing triggers

SQL Code -271
Constant SQLE_TRIGGER_DEFN_CONFLICT
SQL State 52W11
ODBC state S0001

Probable cause A trigger definition could not be created because it conflicts with an existing trigger definition. A trigger with the same name may already exist.

cannot drop a user that owns procedures in runtime engine

SQL Code -270
Constant SQLE_USER_OWNS_PROCEDURES
SQL State 55W04
ODBC state 37000
Probable cause This error is reported by the runtime engine if you attempt to drop a user that owns procedures. Because this operation would result in dropping procedures, and the runtime engine cannot drop procedures, it is not allowed. Use the full engine.

cannot delete a column referenced in a trigger definition

SQL Code -269
Constant SQLE_COLUMN_IN_TRIGGER
SQL State 53W06
ODBC state S1000
Probable cause This error is reported if you attempt to delete a column that is referenced in a trigger definition. DROP the trigger before performing the ALTER command.

trigger '%1' not found

SQL Code -268
Constant SQLE_TRIGGER_NOT_FOUND
SQL State 52W10
ODBC state S0002
Parameter Name of the trigger that could not be found.
Probable cause You have misspelled the name of a trigger, or you have connected with a different userid and forgotten to qualify a trigger name with a user name.

COMMIT/ROLLBACK not allowed within atomic operation

SQL Code -267
Constant SQLE_ATOMIC_OPERATION
SQL State 42W28
ODBC state 37000
Probable cause A COMMIT or ROLLBACK statement was encountered while executing within an atomic operation.

database was initialized with an older version of the software

SQL Code	-266
Constant	SQLE_OLD_DBINIT
SQL State	42W27
ODBC state	37000
Probable cause	The database is missing some system table definitions required for this statement. These system table definitions are normally created when a database is initialized. The database should be unloaded and reloaded into a database that has been initialized with a newer version of SQL Anywhere or use DBUPGRADE to upgrade the database to the most recent version.

procedure '%1' not found

SQL Code	-265
Constant	SQLE_PROCEDURE_NOT_FOUND
SQL State	52W09
ODBC state	S0002
Parameter	Name of the procedure that could not be found.
Probable cause	You have misspelled the name of a procedure, or you have connected with a different userid and forgotten to qualify a procedure name with a user name.

wrong number of variables in FETCH

SQL Code	-264
Constant	SQLE_WRONG_NUM_OF_FETCH_VARIABLES
SQL State	42W26
ODBC state	37000
Probable cause	The number of variables specified in the FETCH statement does not match the number of select list items.

invalid absolute or relative offset in FETCH

SQL Code	-263
Constant	SQLE_INVALID_FETCH_POSITION
SQL State	42W25
ODBC state	37000
Probable cause	The offset specified in a FETCH was invalid or NULL.

label '%1' not found

SQL Code	-262
Constant	SQLE_LABEL_NOT_FOUND

SQL State	42W24
ODBC state	37000
Parameter	Name of the label that could not be found.
Probable cause	The label referenced in a LEAVE statement was not found.

there is already a variable named '%1'

SQL Code	-261
Constant	SQLE_VARIABLE_EXISTS
SQL State	42W15
ODBC state	37000
Probable cause	You have tried to CREATE a variable with the name of another variable that already exists.

variable '%1' not found

SQL Code	-260
Constant	SQLE_VARIABLE_NOT_FOUND
SQL State	42W14
ODBC state	37000
Probable cause	You have tried to DROP or SET the value of a SQL variable that was not created or was previously dropped.

foreign key '%1' for table '%2' duplicates an existing foreign key

SQL Code	-251
Constant	SQLE_DUPLICATE_FOREIGN_KEY
SQL State	52W06
ODBC state	23000
Parameter	The role name of the new foreign key.
Parameter	The table containing the foreign key.
Probable cause	You have attempted to define a foreign key that already exists.

identifier '%1' too long

SQL Code	-250
Constant	SQLE_IDENTIFIER_TOO_LONG
SQL State	54003
ODBC state	37000
Parameter	The identifier in error.
Probable cause	An identifier is longer than 128 characters.

unable to delete database file

SQL Code	-243

Constant SQLE_BACKUP_UNABLE_TO_DELETE_FILE
SQL State WB004
ODBC state S1000
Probable cause The specified file could not be deleted. The filename should not be the same as any database file that is currently in use.

incomplete transactions prevent transaction log renaming

SQL Code -242
Constant SQLE_BACKUP_CANNOT_RENAME_LOG_YET
SQL State WB003
ODBC state S1000
Probable cause The last page in the transaction log was read by a call to *db_backup*. One or more currently active connections have partially completed transactions, preventing the transaction log file from being renamed. The *db_backup* call should be reissued.

database backup not started

SQL Code -241
Constant SQLE_BACKUP_NOT_STARTED
SQL State WB002
ODBC state S1000
Probable cause A database backup could not be started. Either you do not have DBA authority, or another backup has started and not completed.

unknown backup operation

SQL Code -240
Constant SQLE_UNKNOWN_BACKUP_OPERATION
SQL State WB001
ODBC state S1000
Probable cause An invalid backup command operation was specified in a call to *db_backup*.

server/database engine version mismatch

SQL Code -232
Constant SQLE_SERVER_ENGINE_MISMATCH
SQL State 08W20
ODBC state 08001
Probable cause Your version of the database server software is not compatible with your version of the database engine.

dblib/database engine version mismatch

SQL Code	-231
Constant	SQLE_DBLIB_ENGINE_MISMATCH
SQL State	08W19
ODBC state	08001
Probable cause	Your executable uses a database interface library that does not match the version number of the database engine.

sqlpp/dblib version mismatch

SQL Code	-230
Constant	SQLE_PP_DBLIB_MISMATCH
SQL State	08W18
ODBC state	08001
Probable cause	Your executable has source files with Embedded SQL that were preprocessed with a preprocessor that does not match the database interface library.

result set not allowed from within an atomic compound statement

SQL Code	-222
Constant	SQLE_RESULT_NOT_ALLOWED
SQL State	3BW02
ODBC state	S1000
Probable cause	A SELECT statement with no INTO clause or a RESULT CURSOR statement are not allowed within an atomic compound statement.

ROLLBACK TO SAVEPOINT not allowed

SQL Code	-221
Constant	SQLE_ROLLBACK_NOT_ALLOWED
SQL State	3B002
ODBC state	S1000
Probable cause	A ROLLBACK TO SAVEPOINT within an atomic operation is not allowed to a savepoint established before the atomic operation.

savepoint '%1' not found

SQL Code	-220
Constant	SQLE_SAVEPOINT_NOTFOUND
SQL State	3B001
ODBC state	S1000

Parameter Name of savepoint.
Probable cause You attempted to rollback to a savepoint that does not exist.

procedure in use

SQL Code -215
Constant SQLE_PROCEDURE_IN_USE
SQL State 42W23
ODBC state 40001
Probable cause You have attempted to DROP a procedure that is being used by other active users of the database.

table in use

SQL Code -214
Constant SQLE_TABLE_IN_USE
SQL State 42W21
ODBC state 40001
Probable cause You have attempted to ALTER or DROP a table that is being used by other active users of the database.

savepoints require a rollback log

SQL Code -213
Constant SQLE_SAVEPOINTS_REQUIRE_UNDO
SQL State 3BW01
ODBC state S1000
Probable cause You cannot use savepoints when the database engine is running in bulk mode without a rollback log.

CHECKPOINT command requires a rollback log

SQL Code -212
Constant SQLE_CHECKPOINT_REQUIRES_UNDO
SQL State 42W20
ODBC state 40001
Probable cause You cannot use a CHECKPOINT command when the database engine is running in bulk mode without a rollback log.

not allowed while %1 is using the database

SQL Code -211
Constant SQLE_MUST_BE_ONLY_CONNECTION
SQL State 42W19
ODBC state 40001

Probable cause You have attempted to CREATE or DROP a dbspace and there are other active users of the database. You must be the only connection for these commands.

%1 has the row in %2 locked

SQL Code -210
Constant SQLE_LOCKED
SQL State 42W18
ODBC state 40001
Parameter Name of another user.
Parameter Table which generates the error.
Probable cause You have attempted to read or write a row and it is locked by another user. Note that this error will only be received if the database option BLOCKING is set to OFF. Otherwise, the requesting transaction will block until the row lock is released.

invalid value for column '%1' in table '%2'

SQL Code -209
Constant SQLE_INVALID_COLUMN_VALUE
SQL State 23506
ODBC state 23000
Parameter Name of the column that was assigned an invalid value.
Parameter Name of the table containing the column.
Probable cause An INSERT or UPDATE has specified a value for a column that violates a CHECK constraint, and the INSERT or UPDATE were not done because of the error. Note that a CHECK constraint is violated if it evaluates to FALSE; it is okay if it evaluates to TRUE or UNKNOWN.

row has changed since last read -- operation cancelled

SQL Code -208
Constant SQLE_ROW_UPDATED_SINCE_READ
SQL State 22W02
ODBC state (handled by ODBC driver)
Probable cause You have done a UPDATE (positioned) or DELETE (positioned) on a cursor declared as a SCROLL cursor, and the row you are changing has been updated since you read it. This prevents the 'lost update' problem.

wrong number of values for INSERT

SQL Code -207

Constant SQLE_WRONG_NUM_OF_INSERT_COLS
SQL State 53002
ODBC state 37000
Probable cause The number of values you are trying to insert does not match the number of columns specified in the INSERT command, or the number of columns in the table if no columns are specified.

invalid setting for option '%1'

SQL Code -201
Constant SQLE_INVALID_OPTION_SETTING
SQL State 42W17
ODBC state 37000
Parameter Name of the invalid option.
Probable cause You have supplied an invalid value for an option in the SET command. Some options only allow numeric values, while other options only allow the values `'on'` and `'off'`.

invalid option '%1' -- no PUBLIC setting exists

SQL Code -200
Constant SQLE_INVALID_OPTION
SQL State 42W16
ODBC state 37000
Parameter Name of the invalid option.
Probable cause You have probably misspelled the name of an option in the SET OPTION command. You can only define an option for a user if the database administrator has supplied a PUBLIC value for that option.

INSERT/DELETE on cursor can modify only one table

SQL Code -199
Constant SQLE_ONLY_ONE_TABLE
SQL State 09W04
ODBC state 37000
Probable cause You have attempted to INSERT into a cursor and have specified values for more than one table; or you have tried to DELETE from a cursor that involves a join. INSERT into one table at a time. For DELETE, use the FROM clause to specify which table you wish to delete from.

primary key for row in table '%1' is referenced in another table

SQL Code -198

Constant	SQLE_PRIMARY_KEY_VALUE_REF
SQL State	23W05
ODBC state	23000
Parameter	The name of the table with a primary key that is referenced.
Probable cause	You have attempted to delete or modify a primary key that is referenced elsewhere in the database.

no current row of cursor

SQL Code	-197
Constant	SQLE_NO_CURRENT_ROW
SQL State	09W03
ODBC state	24000
Probable cause	You have attempted to perform an operation on the current row of a cursor, but there is no current row. The cursor is before the first row of the cursor, after the last row or is on a row that has since been deleted.

index '%1' for table '%2' would not be unique

SQL Code	-196
Constant	SQLE_INDEX_NOT_UNIQUE
SQL State	23505
ODBC state	23000
Parameter	Name of the index that would not be unique.
Parameter	Name of the table that contains the index.
Probable cause	You have inserted or updated a row that has the same value as another row in some column, and there is a constraint that does not allow two rows to have the same value in that column.

column '%1' in table '%2' cannot be NULL

SQL Code	-195
Constant	SQLE_COLUMN_CANNOT_BE_NULL
SQL State	23502
ODBC state	23000
Parameter	Name of the column that cannot be NULL.
Parameter	Name of the table containing the column.
Probable cause	You have not supplied a value where a value is required.

no primary key value for foreign key '%1' in table '%2'

SQL Code	-194
Constant	SQLE_INVALID_FOREIGN_KEY
SQL State	23503

ODBC state	23000
Parameter	Name of the foreign key.
Parameter	Name of the table with the foreign key.
Probable cause	You have tried to insert or update a row that has a foreign key for another table, and the value for the foreign key is not NULL and there is not a corresponding value in the primary key

primary key for table '%1' is not unique

SQL Code	-193
Constant	SQLE_PRIMARY_KEY_NOT_UNIQUE
SQL State	23W01
ODBC state	23000
Parameter	Name of the table where the problem was detected.
Probable cause	You have tried to add a new row to a table where the new row has the same primary key as an existing row. The database has not added the incorrect row to the database. For example, you might have added a student with student number 86004 and there is already a row for a student with that number.

invalid operation on joined tables

SQL Code	-192
Constant	SQLE_NON_UPDATEABLE_VIEW
SQL State	53W03
ODBC state	37000
Probable cause	You have tried to delete from a query involving more than one table.

cannot modify column '%1' in table '%2'

SQL Code	-191
Constant	SQLE_CANNOT_MODIFY
SQL State	53008
ODBC state	37000
Parameter	Name of the column that cannot be changed.
Parameter	Name of the table containing the column.
Probable cause	You do not have permission to modify the column, or the table is actually a view and the column in the view is defined as an expression (such as `'column1+column2'`) that cannot be modified.

cannot update an expression

SQL Code	-190
Constant	SQLE_NON_UPDATEABLE_COLUMN
SQL State	53W02
ODBC state	37000
Probable cause	You have tried to update a column in a query that is a database expression rather than a column in a table.

unable to find in index '%1' for table '%2'

SQL Code	-189
Constant	SQLE_NOT_FOUND_IN_INDEX
SQL State	WI005
ODBC state	S1000
Parameter	Name of invalid index.
Parameter	Name of table containing the invalid index.
Probable cause	This is a SQL Anywhere internal error and should be reported to Watcom. You should be able to work around the error by dropping and recreating the index.

not enough values for host variables

SQL Code	-188
Constant	SQLE_NOT_ENOUGH_HOST_VARS
SQL State	07001
Probable cause	You have not provided enough host variables for either the number of bind variables, or the command, or the number of select list items.

invalid operation for this cursor

SQL Code	-187
Constant	SQLE_CURSOROP_NOT_ALLOWED
SQL State	09W02
ODBC state	24000
Probable cause	An operation that is not allowed was attempted on a cursor.

subquery cannot return more than one result

SQL Code	-186
Constant	SQLE_SUBQUERY_RESULT_NOT_UNIQUE
SQL State	21W01
ODBC state	37000

Probable cause The result of a subquery contains more than one row. If the subquery is in the WHERE clause, you might be able to use IN.

SELECT returns more than one row

SQL Code -185
Constant SQLE_TOO_MANY_RECORDS
SQL State 21000
ODBC state S1000
Probable cause An Embedded SELECT statement that does not use a cursor returns more than one result.

error inserting into cursor

SQL Code -184
Constant SQLE_PUT_CURSOR_ERROR
SQL State 09W01
ODBC state S1000
Probable cause An error has occurred while inserting into a cursor.

cannot find index named '%1'

SQL Code -183
Constant SQLE_INDEX_NOT_FOUND
SQL State 52W03
ODBC state S0012
Parameter Name of the index that cannot be found.
Probable cause A DROP INDEX command has named an index that does not exist. Check for spelling errors or whether the index name must be qualified by a userid.

not enough fields allocated in SQLDA

SQL Code -182
Constant SQLE_SQLDA_TOO_SMALL
SQL State 07002
Probable cause There are not enough fields in the SQLDA to retrieve all of the values requested.

no indicator variable provided for NULL result

SQL Code -181
Constant SQLE_NO_INDICATOR
SQL State 22002

ODBC state	S1000
Probable cause	You tried to retrieve a value from the database that was the NULL value but you did not provide an indicator variable for that value.

cursor not open

SQL Code	-180
Constant	SQLE_CURSOR_NOT_OPEN
SQL State	24501
ODBC state	34000
Probable cause	You attempted to OPEN a cursor that has not been declared.

cursor already open

SQL Code	-172
Constant	SQLE_CURSOR_ALREADY_OPEN
SQL State	24502
ODBC state	24000
Probable cause	You attempted to OPEN a cursor that is already open.

error opening cursor

SQL Code	-171
Constant	SQLE_OPEN_CURSOR_ERROR
SQL State	07003
ODBC state	24000
Probable cause	You have attempted to open a cursor on a statement that is not a SELECT statement or a CALL statement.

cursor has not been declared

SQL Code	-170
Constant	SQLE_CURSOR_NOT_DECLARED
SQL State	24W01
ODBC state	24000
Probable cause	You attempted to OPEN a cursor that has not been declared.

Cannot outer join a view with a UNION or GROUP BY

SQL Code	-162
Constant	SQLE_CANNOT_OUTER_JOIN
SQL State	52W19
ODBC state	37000

Probable cause A view that contains a UNION or view cannot be used on the right of a LEFT OUTER JOIN or on the left of a RIGHT OUTER JOIN.

invalid type on DESCRIBE statement

SQL Code -161
Constant SQLE_INVALID_DESCRIBE_TYPE
SQL State 07W01
ODBC state (handled by ODBC driver)
Probable cause This is an internal C language interface error. If it occurs, it should be reported to Watcom.

can only describe a SELECT statement

SQL Code -160
Constant SQLE_DESCRIBE_NONSELECT
SQL State 07005
ODBC state (handled by ODBC driver)
Probable cause In the C language interface, you attempted to describe the select list of a statement other than a SELECT statement.

invalid column number

SQL Code -159
Constant SQLE_INVALID_COLUMN_NUMBER
SQL State 42W13
ODBC state S1000
Probable cause The column number in a GET DATA command is invalid.

value %1 too large for destination

SQL Code -158
Constant SQLE_OVERFLOW_ERROR
SQL State 22003
Parameter The value that caused the overflow.
Probable cause A value has been supplied to the database or retrieved from the database that is too large for the destination column or host variable. For example, the value `'10'` may have been supplied for a DECIMAL(3,2) field.

cannot convert %1 to a %2

SQL Code -157
Constant SQLE_CONVERSION_ERROR

SQL State	53018
ODBC state	07006
Parameter	The value that could not be converted.
Parameter	The name of the type for the conversion.
Probable cause	An invalid value has been supplied to or fetched from the database. For example, the value `'12X'` might have been supplied where a number was required.

invalid expression near '%1'

SQL Code	-156
Constant	SQLE_EXPRESSION_ERROR
SQL State	42W08
ODBC state	37000
Parameter	The invalid expression.
Probable cause	You have an expression which the database engine cannot understand. For example, you might have tried to add two dates.

invalid host variable

SQL Code	-155
Constant	SQLE_VARIABLE_INVALID
SQL State	42W07
ODBC state	37000
Probable cause	A host variable supplied to the database using the C language interface as either a host variable or through an SQLDA is invalid.

wrong number of parameters to function '%1'

SQL Code	-154
Constant	SQLE_WRONG_PARAMETER_COUNT
SQL State	37505
ODBC state	37000
Parameter	Name of the function.
Probable cause	You have supplied an incorrect number of parameters to a database function. Refer to "Functions" on page 765 for the correct number of parameters.

SELECT lists in UNION do not match in length

SQL Code	-153
Constant	SQLE_INVALID_UNION
SQL State	53026

ODBC state 37000
Probable cause You have specified a UNION but the SELECT statements involved in the union do not have the same number of columns in the select list.

number in ORDER BY is too large

SQL Code -152
Constant SQLE_INVALID_ORDER
SQL State 53005
ODBC state 37000
Probable cause You have used an integer in an ORDER BY list and the integer is larger than the number of columns in the select list.

subquery allowed only one select list item

SQL Code -151
Constant SQLE_SUBQUERY_SELECT_LIST
SQL State 53023
ODBC state 37000
Probable cause You have entered a subquery which has more than one column in the select list. Change the select list to have only one column.

aggregate functions not allowed on this statement

SQL Code -150
Constant SQLE_AGGREGATES_NOT_ALLOWED
SQL State 42W06
ODBC state 37000
Probable cause An UPDATE statement has used an aggregate function (MIN, MAX, SUM, AVG or COUNT).

function or column reference to '%1' in the select list must also appear in a GROUP BY

SQL Code -149
Constant SQLE_INVALID_GROUP_SELECT
SQL State 53003
ODBC state 37000
Parameter Name of the column referenced directly, or in an expression, that must be in the GROUP BY clause.
Probable cause In a query using GROUP BY, select list items that are not aggregate functions must also appear in the GROUP BY clause. If the select list item is a column reference or an alias, simply add the column name or alias to the GROUP BY

clause. If the select list item is a scalar function, ensure that the function's arguments in the GROUP BY clause match exactly with those in the select list. In some cases, you may want to use the MAX function on the column name (or another aggregate function) instead of adding the column to the GROUP BY clause.

unknown function '%1'

SQL Code	-148
Constant	SQLE_UNKNOWN_FUNC
SQL State	42W05
ODBC state	37000
Parameter	Function name that is not a database function.
Probable cause	You have misspelled the name of a database function (such as MAXIMUM instead of MAX) in a query definition or in a query column name. Refer to "Functions" on page 765 for the correct function name.

there is more than one way to join '%1' to '%2'

SQL Code	-147
Constant	SQLE_AMBIGUOUS_JOIN
SQL State	52W08
ODBC state	37000
Parameter	Name of first table that cannot be joined.
Parameter	Name of second table that cannot be joined.
Probable cause	There are two or more foreign keys relating the two tables and you are attempting to KEY JOIN the two tables. Either there are two foreign keys from the first table to the second table, or each table has a foreign key to the other table. You must use a correlation name for the primary key table which is the same as the role name of the desired foreign key relationship.

there is no way to join '%1' to '%2'

SQL Code	-146
Constant	SQLE_CANNOT_JOIN
SQL State	53W04
ODBC state	37000
Parameter	Name of first table that cannot be joined.
Parameter	Name of second table that cannot be joined.
Probable cause	You have attempted a KEY JOIN between two tables and there is no foreign key on one of the tables that references the primary key of the other table; or you have attempted a

NATURAL JOIN between two tables and the tables have no common column names.

foreign key name '%1' not found

SQL Code	-145
Constant	SQLE_FOREIGN_KEY_NAME_NOT_FOUND
SQL State	52W07
ODBC state	37000
Parameter	Name of the non-existing foreign key.
Probable cause	You have misspelled the name of a foreign key or the foreign key does not exist.

column '%1' found in more than one table -- need a correlation name

SQL Code	-144
Constant	SQLE_COLUMN_AMBIGUOUS
SQL State	52002
ODBC state	SJS01
Parameter	Name of the ambiguous column.
Probable cause	You have not put a correlation name on a column that is found in more than one of the tables referenced in a query. You need to add a correlation name to the reference.

column '%1' not found

SQL Code	-143
Constant	SQLE_COLUMN_NOT_FOUND
SQL State	52003
ODBC state	S0022
Parameter	Name of the column that could not be found.
Probable cause	You have misspelled the name of a column, or the column you are looking for is in a different table.

correlation name '%1' not found

SQL Code	-142
Constant	SQLE_CORRELATION_NAME_NOT_FOUND
SQL State	52W02
ODBC state	S0002
Parameter	Name of the invalid correlation name.
Probable cause	You have misspelled a correlation name, or you have used a table name instead of the correlation name.

table '%1' not found

SQL Code	-141
Constant	SQLE_TABLE_NOT_FOUND
SQL State	52W01
ODBC state	S0002
Parameter	Name of the table that could not be found.
Probable cause	You have misspelled the name of a table, or you have connected with a different userid and forgotten to qualify a table name with a user name. For example, you might have connected to userid `'Teacher'` and tried to refer to **Student** instead of **admin.Student**.

'%1' is an unknown userid

SQL Code	-140
Constant	SQLE_UNKNOWN_USERID
SQL State	08004
ODBC state	28000
Parameter	Name of the userid that could not be found.
Probable cause	The specified userid does not exist.

more than one table is identified as '%1'

SQL Code	-139
Constant	SQLE_CORRELATION_NAME_AMBIGUOUS
SQL State	52012
ODBC state	SG001
Parameter	Ambiguous correlation name.
Probable cause	You have identified two tables in the same FROM clause with the same correlation name.

dbspace '%1' not found

SQL Code	-138
Constant	SQLE_DBSPACE_NOT_FOUND
SQL State	52W13
ODBC state	S0002
Parameter	Name of the dbspace that could not be found.
Probable cause	The named dbspace was not found.

table '%1' requires a unique correlation name

SQL Code	-137
Constant	SQLE_CORRELATION_NAME_NEEDED

SQL State	52W15
ODBC state	37000
Parameter	Name of the table that needs a unique correlation name.
Probable cause	You have specified a join that joins a table to itself. You need to use unique correlation names in order to have multiple instances of a table.

table '%1' is in an outer join cycle

SQL Code	-136
Constant	SQLE_OUTER_JOIN_CYCLE
SQL State	52W14
ODBC state	37000
Parameter	Name of a table in the cycle.
Probable cause	You have specified outer joins that create a cycle of tables.

language extension

SQL Code	-135
Constant	SQLE_LANGUAGE_EXTENSION
SQL State	0AW01
ODBC state	S1000
Probable cause	The requested operation is valid in some versions of SQL, but not in SQL Anywhere.

%1 not implemented

SQL Code	-134
Constant	SQLE_NOT_IMPLEMENTED
SQL State	0A000
ODBC state	S1000
Parameter	The unimplemented feature.
Probable cause	The requested operation or feature is not implemented in SQL Anywhere.

invalid prepared statement type

SQL Code	-133
Constant	SQLE_INVALID_STATEMENT_TYPE
SQL State	07W03
ODBC state	S1000
Probable cause	This is an internal C language interface error. If it occurs, it should be reported to Watcom.

SQL statement error

SQL Code	-132
Constant	SQLE_STATEMENT_ERROR
SQL State	26501
ODBC state	S1000
Probable cause	The statement identifier (generated by PREPARE) passed to the database for a further operation is invalid.

syntax error near '%1'

SQL Code	-131
Constant	SQLE_SYNTAX_ERROR
SQL State	42W04
ODBC state	37000
Parameter	The word or symbol where the syntax error has been detected.
Probable cause	The database engine cannot understand the command you are trying to execute. If you have used a keyword (such as DATE) for a column name, try enclosing the keyword in quotation marks ("DATE").

invalid statement

SQL Code	-130
Constant	SQLE_INVALID_STATEMENT
SQL State	07W02
ODBC state	S1000
Probable cause	The statement identifier (generated by PREPARE) passed to the database for a further operation is invalid.

cannot drop a user that owns tables in runtime engine

SQL Code	-128
Constant	SQLE_USER_OWNS_TABLES
SQL State	55W03
ODBC state	37000
Probable cause	This error is reported by the runtime engine if you attempt to drop a user that owns tables. Because this operation would result in dropping tables, and the runtime engine cannot drop tables, it is not allowed. Use the full engine.

cannot alter a column in an index

SQL Code	-127
Constant	SQLE_COLUMN_IN_INDEX

SQL State 53W05
ODBC state S1000
Probable cause This error is reported if you attempt to delete or modify the definition of a column that is part of a primary or foreign key. This error will also be reported if you attempt to delete a column that has an index on it. DROP the index or key, perform the ALTER command, and then add the index or key again.

table cannot have two primary keys

SQL Code -126
Constant SQLE_PRIMARY_KEY_TWICE
SQL State 52W05
ODBC state 23000
Probable cause You have specified the primary key twice in a CREATE TABLE command.

ALTER clause conflict

SQL Code -125
Constant SQLE_ALTER_CLAUSE_CONFLICT
SQL State 53W01
ODBC state S1000
Probable cause A primary key clause, foreign key clause, or a uniqueness clause must be the only clause of an ALTER TABLE command.

'%1' is not a user group

SQL Code -123
Constant SQLE_NOT_A_GROUP
SQL State 42W03
ODBC state 37000
Parameter Name of user you thought was a group.
Probable cause You have tried to add a member to group, but the group specified has not been granted the GROUP special privilege.

operation would cause a group cycle

SQL Code -122
Constant SQLE_GROUP_CYCLE
SQL State 42W02
ODBC state 37000

Probable cause	You have tried to add a member to group that would result in a member belonging to itself (perhaps indirectly).

do not have permission to %1

SQL Code	-121
Constant	SQLE_PERMISSION_DENIED
SQL State	42501
ODBC state	42001
Parameter	Description of the type of permission lacking.
Probable cause	You have not been granted permission to use a table belonging to another userid.

'%1' already has grant permission

SQL Code	-120
Constant	SQLE_ALREADY_HAS_GRANT_PERMS
SQL State	42W01
ODBC state	37000
Parameter	Name of the userid that already has GRANT permission.
Probable cause	The SQL GRANT command is attempting to give a user GRANT OPTION and that user already has GRANT OPTION.

primary key column '%1' already defined

SQL Code	-119
Constant	SQLE_PRIMARY_KEY_COLUMN_DEFINED
SQL State	52009
ODBC state	23000
Parameter	Name of the column that is already in the primary key.
Probable cause	You have listed the same column name twice in the definition of a primary key.

table '%1' has no primary key

SQL Code	-118
Constant	SQLE_NO_PRIMARY_KEY
SQL State	55008
ODBC state	23000
Parameter	Name of the table that does not have a primary key.
Probable cause	You have attempted to add a foreign key referring to a table that does not have a primary key. You will need to add a primary key to the named table.

table must be empty

SQL Code	-116
Constant	SQLE_TABLE_MUST_BE_EMPTY
SQL State	55W02
ODBC state	S1000
Probable cause	You have attempted to modify a table, and SQL Anywhere can only perform the change if there are no rows in the table.

number of columns does not match SELECT

SQL Code	-114
Constant	SQLE_VIEW_DEFINITION_ERROR
SQL State	53011
ODBC state	21S01
Probable cause	An INSERT command contains a SELECT with a different number of columns than the INSERT.

column %1 in foreign key has a different definition than primary key

SQL Code	-113
Constant	SQLE_INVALID_FOREIGN_KEY_DEF
SQL State	53030
ODBC state	23000
Parameter	Name of the problem column.
Probable cause	The data type of the column in the foreign key is not the same as the data type of the column in the primary key. Change the definition of one of the columns using ALTER TABLE.

table already has a primary key

SQL Code	-112
Constant	SQLE_EXISTING_PRIMARY_KEY
SQL State	55013
ODBC state	23000
Probable cause	You have tried to add a primary key on a table that already has a primary key defined. You must delete the current primary key before adding a new one.

index name '%1' not unique

SQL Code	-111
Constant	SQLE_INDEX_NAME_NOT_UNIQUE
SQL State	52W04
ODBC state	S0011

Parameter	Name of the invalid index.
Probable cause	You have attempted to create an index with a name of an existing index.

'%1' already exists

SQL Code	-110
Constant	SQLE_NAME_NOT_UNIQUE
SQL State	52010
ODBC state	S0001
Parameter	Name of the item that already exists.
Probable cause	You have tried to create a table, view, column, foreign key, or publication with the same name as an existing one.

there are still active database connections

SQL Code	-109
Constant	SQLE_STILL_ACTIVE_CONNECTIONS
SQL State	08W06
ODBC state	S1000
Probable cause	An application has requested SQL Anywhere to shutdown the database using the *db_stop()* function when there are still active connections to the database.

connection not found

SQL Code	-108
Constant	SQLE_CONNECTION_NOT_FOUND
SQL State	08W02
ODBC state	08003
Probable cause	The specified connection name on a DISCONNECT or SET CONNECTION statement is invalid.

error writing to log file

SQL Code	-107
Constant	SQLE_ERROR_WRITING_LOG
SQL State	08W17
ODBC state	S1000
Probable cause	The database engine got an I/O error writing the log file. Perhaps the disk is full or the log file name is invalid.

cannot open log file %1

SQL Code	-106

Constant SQLE_CANNOT_OPEN_LOG
SQL State 08W05
ODBC state 08003
Parameter Name of log file.
Probable cause The database engine was unable to open the transaction log file. Perhaps the log file name specifies an invalid device or directory. If this is the case, you can use the DBLOG utility to find out where the transaction log should be and perhaps change it.

cannot be started -- %1

SQL Code -105
Constant SQLE_UNABLE_TO_CONNECT
SQL State 08001
Parameter Name of database.
Probable cause The specified database environment cannot be found. If it is a database name, then it does not exist, it is not a database, it is corrupt, or it is an older format. If it is a server name, then the server cannot be found.

invalid userid and password on preprocessed module

SQL Code -104
Constant SQLE_INVALID_MODULE_LOGON
SQL State 28W01
ODBC state 28000
Probable cause A userid and password were specified when a module was preprocessed but the userid or password is invalid.

invalid userid or password

SQL Code -103
Constant SQLE_INVALID_LOGON
SQL State 28000
Probable cause The user has supplied an invalid userid or an incorrect password. ISQL will handle this error by presenting a connection dialog to the user.

too many connections to database

SQL Code -102
Constant SQLE_TOO_MANY_CONNECTIONS
SQL State 08W03
ODBC state 08004

Probable cause If you are running the multi-user client, you have exceeded the number of computers allowed to connect to the server by your license agreement. Otherwise, the single user DOS engine is limited to 2 connections, and the Windows engine is restricted to 10 connections.

not connected to SQL database

SQL Code -101
Constant SQLE_NOT_CONNECTED
SQL State 08003
Probable cause You have not connected to the database, or you have executed the DISCONNECT command and have not connected to the database again.

database engine not running

SQL Code -100
Constant SQLE_ENGINE_NOT_RUNNING
SQL State 08W01
ODBC state 08001
Probable cause You have not run the database engine or network requestor or the interface library is unable to find it.

connections to database have been disabled

SQL Code -99
Constant SQLE_CONNECTIONS_DISABLED
SQL State 08W04
ODBC state 08005
Probable cause Connections to the multi-user server have been disabled on the server console. You will receive this error until they have been reenabled on the server console.

authentication violation

SQL Code -98
Constant SQLE_AUTHENTICATION_VIOLATION
SQL State 08W21
ODBC state 08001
Probable cause You have attempted to connect to an engine or server which has been authenticated for exclusive use with a specific application.

database's page size too big

SQL Code	-97
Constant	SQLE_PAGE_SIZE_TOO_BIG
SQL State	08W22
ODBC state	08004
Probable cause	You have attempted to start a database with a page size that exceeds the maximum page size of the running engine. Restart the engine with this database named on the commandline.

database engine already running

SQL Code	-96
Constant	SQLE_ENGINE_ALREADY_RUNNING
SQL State	08W23
ODBC state	S1000
Probable cause	The database engine was not able to start on a *db_start_engine* call because it was already running.

invalid parameter

SQL Code	-95
Constant	SQLE_INVALID_PARAMETER
SQL State	08W24
ODBC state	08004
Probable cause	An error occurred while parsing the string parameter associated with one of the entry points: db_start_engine(), db_start_database(), db_stop_engine(), db_stop_database(), db_string_connect().

database engine not running in multi-user mode

SQL Code	-89
Constant	SQLE_ENGINE_NOT_MULTIUSER
SQL State	08W16
ODBC state	08001
Probable cause	The database was started for bulk loading (the `-b` switch) and cannot be used as a multi-user engine. Stop the database, and start again without `-b`. In the DOS version of Watcom SQL Version 3.0, the database engine was not started in multi-user mode.

client/server communications protocol mismatch

SQL Code	-88

Constant	SQLE_PROTOCOL_MISMATCH
SQL State	08W15
ODBC state	08S01
Probable cause	The multi-user client was unable to start because the protocol versions of the client and the running server do not match. Make sure the client and server software are the same version.

database name required to start engine

SQL Code	-87
Constant	SQLE_DATABASE_NAME_REQUIRED
SQL State	08W14
ODBC state	08001
Probable cause	A database name is required to start the database engine or the multi-user client, but it was not specified.

not enough memory to start

SQL Code	-86
Constant	SQLE_NO_MEMORY
SQL State	08W13
ODBC state	S1001
Probable cause	The database engine or multi-user client executable was loaded but was unable to start because there is not enough memory to run properly.

communication error

SQL Code	-85
Constant	SQLE_COMMUNICATIONS_ERROR
SQL State	08W12
ODBC state	08S01
Probable cause	There is a communication problem between the multi-user client and server. This happens most frequently when the multi-user client was unable to start because a communication error occurred while trying to locate the server.

specified database is invalid

SQL Code	-84
Constant	SQLE_INVALID_DATABASE
SQL State	08W11
ODBC state	08001
Probable cause	The database engine was started but the specified database file is invalid. The engine is stopped.

specified database not found

SQL Code -83
Constant SQLE_DATABASE_NOT_FOUND
SQL State 08W10
ODBC state 08001
Probable cause The database engine or multi-user client was started but was unable to find the specified database or server name. The database file cannot be opened or the specified server cannot be found on the network. The database engine or client is stopped.

unable to start specified database

SQL Code -82
Constant SQLE_UNABLE_TO_START_DATABASE
SQL State 08W09
ODBC state 08001
Probable cause The database engine or multi-user client was started but was unable to find the specified database or server name. No specific reason is known.

invalid database engine command line

SQL Code -81
Constant SQLE_INVALID_COMMAND_LINE
SQL State 08W08
ODBC state 08001
Probable cause It was not possible to start the database engine or multi-user client because the command line was invalid. See "Environment variables" on page 682 for a description of how the engine is started.

unable to start database engine

SQL Code -80
Constant SQLE_UNABLE_TO_START_ENGINE
SQL State 08W07
ODBC state 08001
Probable cause It was not possible to start the database engine or multi-user client. Either there is not enough memory to run the database engine, or the executable cannot be found. See "Environment variables" on page 682 for a description of how the engine is started.

invalid local database switch

SQL Code	-79
Constant	SQLE_INVALID_LOCAL_OPTION
SQL State	08W25
ODBC state	08001
Probable cause	An invalid local database switch was found in the DBS option.

Dynamic memory exhausted!

SQL Code	-78
Constant	SQLE_DYNAMIC_MEMORY_EXHAUSTED
SQL State	08W26
ODBC state	S1001
Probable cause	A failure occurred when trying to allocate dynamic memory.

database name not unique

SQL Code	-77
Constant	SQLE_ALIAS_CLASH
SQL State	08W27
ODBC state	08001
Probable cause	The database cannot be loaded as its name is conflicting with a previously loaded database.

request denied -- no active databases

SQL Code	-76
Constant	SQLE_REQUEST_DENIED_NO_DATABASES
SQL State	08W28
ODBC state	08001
Probable cause	The engine has denied the request as there are currently no loaded databases.

request to start/stop database denied

SQL Code	-75
Constant	SQLE_START_STOP_DATABASE_DENIED
SQL State	08W29
ODBC state	08001
Probable cause	The engine has denied permission to start/stop a database.

the selected database is currently inactive

SQL Code	-74

Constant SQLE_DATABASE_NOT_ACTIVE
SQL State 08W30
ODBC state 08001
Probable cause The selected database is in an inactive state. This state occurs during database initialization and shutdown.

communication buffer underflow

SQL Code -73
Constant SQLE_COMMUNICATIONS_UNDERFLOW
SQL State 08W31
ODBC state 08S01
Probable cause The multi-user client or server has received a network buffer containing insufficient data. If this error occurs, it should be reported to Watcom.

(no message)

SQL Code 0
Constant SQLE_NOERROR
SQL State 00000
Probable cause This code indicates that there was no error or warning.

row not found

SQL Code 100
Constant SQLE_NOTFOUND
SQL State 02000
ODBC state (handled by ODBC driver)
Probable cause You have positioned a cursor beyond the beginning or past the end of the query. There is no row at that position.

value truncated

SQL Code 101
Constant SQLE_TRUNCATED
SQL State 01004
Probable cause You have tried to insert, update, or select a value in the database which is too large to fit in the destination. This warning is also produced if you do a fetch, and the host variable or SQLDA variable is not large enough to receive the value.

using temporary table

SQL Code	102
Constant	SQLE_TEMPORARY_TABLE
SQL State	01W02
ODBC state	(handled by ODBC driver)
Probable cause	A temporary table has been created in order to satisfy the query. It can only occur on an OPEN statement.

invalid data conversion

SQL Code	103
Constant	SQLE_CANNOT_CONVERT
SQL State	01W03
ODBC state	07006
Probable cause	The database could not convert a value to the required type. This is either a value supplied to the database on an insert, update or as a host bind variable, or a value retrieved from the database into a host variable or SQLDA.

row has been updated since last time read

SQL Code	104
Constant	SQLE_ROW_UPDATED_WARNING
SQL State	01W04
ODBC state	(handled by ODBC driver)
Probable cause	A FETCH has retrieved a row from a cursor declared as a SCROLL cursor, and the row was previously fetched from the same cursor, and one or more columns in the row has been updated since the previous fetch. Note that the column(s) updated may or may not be fetched by the cursor; this warning just indicates that the row from the table has been updated. If the cursor involves more than one table, a row from one or more of the tables has been updated.

procedure has completed

SQL Code	105
Constant	SQLE_PROCEDURE_COMPLETE
SQL State	01W05
ODBC state	(handled by ODBC driver)
Probable cause	An OPEN or a RESUME has caused a procedure to execute to completion. There are no more result sets available from this procedure. This warning will also be returned if you attempt to RESUME a cursor on a SELECT statement.

value for column '%1' in table '%2' has changed

SQL Code	106
Constant	SQLE_COLUMN_VALUE_CHANGED
SQL State	01W06
ODBC state	(handled by ODBC driver)
Parameter	Name of the column whose value has changed.
Parameter	Name of the table containing the column.
Probable cause	A replicated UPDATE has found a value in an updated column that does not match the value when the original UPDATE was made.

warning

SQL Code	200
Constant	SQLE_WARNING
SQL State	01000
ODBC state	(handled by ODBC driver)
Probable cause	A warning has occurred. The warning message will indicate the condition that caused the warning.

the supplied buffer was too small to hold all requested query results

SQL Code	400
Constant	SQLE_HLI_MORE_DATA_AVAILABLE
SQL State	01WH1
ODBC state	(handled by ODBC driver)
Probable cause	You attempted to get a query result set using the WSQL HLI function *wsqlquerytomem*. The buffer supplied by the calling application was too small to contain the entire query. The buffer will contain as many rows of the result set as possible, and the cursor will be positioned on the next row of the result set.

39.3.2 Internal errors (assertion failed)

The SQL Anywhere engine has many internal checks that have been designed to detect possible database corruption as soon as possible. If the database engine prints an Assertion Failed message, you should not continue to use it before attempting to determine the cause. You should record the assertion number displayed on the screen and report the error to SQL Anywhere technical support. The DBVALID utility is useful for determining if your database file is corrupt. You may find it necessary to reconstruct your data from backups and transaction logs (see "Backup and Data Recovery" on page 331).

Chapter 40

SQL Preprocessor Error Messages

About this chapter

This chapter presents a list of all SQL preprocessor errors and warnings.

Contents

40.1 SQLPP errors

subscript value %ld too large

Code 2601
Probable Cause You have attempted to index a host variable that is an array with a value too large for the array.

combined pointer and arrays not supported for hosttypes

Code 2602
Probable Cause You have used an array of pointers as a host variable. This is not legal.

only one dimensional arrays supported for char type

Code 2603
Probable Cause You have used an array of arrays of character as a host variable. This is not a legal host variable type.

VARCHAR type must have a length

Code 2604
Probable Cause You have attempted to declare a VARCHAR or BINARY host variable using the DECL_VARCHAR or DECL_BINARY macro but have not specified a size for the array.

arrays of VARCHAR not supported

Code 2605
Probable Cause You have attempted to declare a host variable as an array of VARCHAR or BINARY. This is not a legal host variable type.

VARCHAR host variables cannot be pointers

Code 2606
Probable Cause You have attempted to declare a host variable as a pointer to a VARCHAR or BINARY. This is not a legal host variable type.

initializer not allowed on VARCHAR host variable

Code 2607

Probable Cause You can not specify a C variable initializer for a host variable of type VARCHAR or BINARY. You must initialize this variable in regular C executable code.

FIXCHAR type must have a length

Code 2608
Probable Cause You have used the DECL_FIXCHAR macro to declare a host variable of type FIXCHAR but have not specified a length.

arrays of FIXCHAR not supported

Code 2609
Probable Cause You have attempted to declare a host variable as an array of FIXCHAR. This is not a legal host variable type.

arrays of int not supported

Code 2610
Probable Cause You have attempted to declare a host variable as an array of *ints*. This is not a legal host variable type.

precision must be specified for decimal type

Code 2611
Probable Cause You must specify the precision when declaring a packed decimal host variable using the DECL_DECIMAL macro. The scale is optional.

arrays of decimal not allowed

Code 2612
Probable Cause You have attempted to declare a host variable as an array of DECIMAL. This is not a legal host variable type.

Unknown hostvar type

Code 2613
Probable Cause You declared a host variable of a type not understood by the SQL preprocessor.

invalid integer

Code 2614

Probable Cause An integer was required in an embedded SQL statement (for a fetch offset, or a host variable array index, etc.) and the preprocessor was unable to convert what was supplied into an integer.

'%s' host variable must be a C string type

Code 2615
Probable Cause A C string was required in an embedded SQL statement (for a cursor name, option name etc.) and the value supplied was not a C string.

'%s' symbol already defined

Code 2617
Probable Cause You defined a host variable twice with different definitions.

invalid type for sql statement variable

Code 2618
Probable Cause A host variable used as a statement identifier should be of type *a_sql_statement_number*. You attempted to use a host variable of some other type as a statement identifier.

Cannot find include file '%s'

Code 2619
Probable Cause The specified include file was not found. Note that the preprocessor will use the INCLUDE environment variable to search for include files.

host variable '%s' is unknown

Code 2620
Probable Cause You have used a host variable in a statement and that host variable has not been declared in a declare section.

indicator variable '%s' is unknown

Code 2621
Probable Cause You have used a indicator variable in a statement and that indicator variable has not been declared in a declare section.

invalid type for indicator variable '%s'

Code 2622
Probable Cause Indicator variables must be of type *short int*. You have used a variable of a different type as an indicator variable.

invalid host variable type on '%s'

Code 2623
Probable Cause You have used a host variable that is not a string type in a place where the preprocessor was expecting a host variable of a string type.

host variable '%s' has two different definitions

Code 2625
Probable Cause The same host variable name was defined with two different types within the same module. Note that host variable names are global to a C module.

statement '%s' not previously prepared

Code 2626
Probable Cause An embedded SQL statement name has been used (EXECUTEd) without first being PREPAREd.

cursor '%s' not previously declared

Code 2627
Probable Cause An embedded SQL cursor name has been used (in a FETCH, OPEN, CLOSE etc.) without first being DECLAREd.

unknown statement '%s'

Code 2628
Probable Cause You attempted to drop an embedded SQL statement that doesn't exist.

host variables not allowed for this cursor

Code 2629
Probable Cause Host variables are not allowed on the declare statement for the specified cursor. If the cursor name is provided through a host variable, then you should use full dynamic SQL and prepare

the statement. A prepared statement may have host variables in it.

host variables specified twice - on declare and open

Code 2630
Probable Cause You have specified host variables for a cursor on both the declare and the open statements. In the static case, you should specify the host variables on the declare statement. In the dynamic case, specify them on the open.

must specify a host list or using clause on %s

Code 2631
Probable Cause The specified statement requires host variables to be specified either in a host variable list or from an SQLDA.

no INTO clause on SELECT statement

Code 2633
Probable Cause You specified an embedded static SELECT statement but you did not specify an INTO clause for the results.

incorrect Embedded SQL syntax

Code 2636
Probable Cause An embedded SQL specific statement (OPEN, DECLARE, FETCH etc.) has a syntax error.

missing ending quote of string

Code 2637
Probable Cause You have specified a string constant in an embedded SQL statement, but there is no ending quote before the end of line or end of file.

token too long

Code 2639
Probable Cause The SQL preprocessor has a maximum token length of 2K. Any token longer than 2K will produce this error. For constant strings in embedded SQL commands (the main place this error shows up) use string concatenation to make a longer string.

'%s' host variable must be an integer type

Code 2640
Probable Cause You have used a host variable that is not of integer type in a statement where only an integer type host variable is allowed.

40.2 SQLPP warnings

Into clause not allowed on declare cursor - ignored

Code 2660
Probable Cause You have specified an into clause on a SELECT statement in a declare cursor. Note that the into clause will be ignored.

unrecognized SQL syntax

Code 2661
Probable Cause You have used a SQL statement that will probably cause a syntax error when the statement is sent to the database engine.

unknown sql function '%s'

Code 2662
Probable Cause You have used a SQL function that is unknown to the preprocessor and will probably cause an error when the statement is sent to the database engine.

wrong number of parms to sql function '%s'

Code 2663
Probable Cause You have used a SQL function with the wrong number of parameters. This will likely cause an error when the statement is sent to the database engine.

"static

Code 2664
Probable Cause You have used a static statement name and preprocessed with the -r reentrancy switch. Static statement names cause static variables to be generated that are filled in by the database. If two threads use the same statement, contention arises over this variable. Use a local host variable as the statement identifier instead of a static name.

Chapter 41

Sample Database Command File

The following ISQL commands were used to create the sample database.

```
% Usage:        isql read e:dbfilessample50makesdb5.sql
%
% This command file reloads a database that was unloaded using
"unload".
%
%

SET OPTION Statistics = 3;
SET OPTION Date_order = 'YMD';


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%    Create userids and grant user permissions
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

GRANT CONNECT TO "DBA" IDENTIFIED BY "SQL";
GRANT RESOURCE, DBA, SCHEDULE TO "DBA";
commit work;


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%    Create tables
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

CREATE TABLE "DBA"."sales_order"
(
        "id"            integer NOT NULL,
        "cust_id"               integer NOT NULL,
        "order_date"            date NOT NULL,
        "fin_code_id"           char(2),
        "region"                char(7),
        "sales_rep"             integer NOT NULL,
        PRIMARY KEY ("id"),
);
CREATE TABLE "DBA"."sales_order_items"
(
        "id"            integer NOT NULL,
        "line_id"               smallint NOT NULL,
        "prod_id"               integer NOT NULL,
        "quantity"              integer NOT NULL,
        "ship_date"             date NOT NULL,
        PRIMARY KEY ("id", "line_id"),
);
CREATE TABLE "DBA"."contact"
(
        "id"            integer NOT NULL,
        "last_name"             char(15) NOT NULL,
        "first_name"            char(15) NOT NULL,
        "title"      char(2) NOT NULL,
        "street"                char(30) NOT NULL,
        "city"          char(20) NOT NULL,
        "state"      char(2) NOT NULL,
        "zip"           char(5) NOT NULL,
        "phone"      char(10),
        "fax"           char(10),
        PRIMARY KEY ("id"),
);
CREATE TABLE "DBA"."customer"
(
        "id"            integer NOT NULL,
        "fname"      char(15) NOT NULL,
```

```
        "lname"      char(20) NOT NULL,
        "address"              char(35) NOT NULL,
        "city"          char(20) NOT NULL,
        "state"      char(2) NOT NULL,
        "zip"           char(10) NOT NULL,
        "phone"      char(12) NOT NULL,
        "company_name"         char(35),
        PRIMARY KEY ("id"),
);
CREATE TABLE "DBA"."fin_code"
(
        "code"          char(2) NOT NULL,
        "type"          char(10) NOT NULL,
        "description"          char(50),
        PRIMARY KEY ("code"),
);
CREATE TABLE "DBA"."fin_data"
(
        "year"          char(4) NOT NULL,
        "quarter"              char(2) NOT NULL,
        "code"          char(2) NOT NULL,
        "amount"               numeric(9,0),
        PRIMARY KEY ("year", "quarter", "code"),
);
CREATE TABLE "DBA"."product"
(
        "id"            integer NOT NULL,
        "name"          char(15) NOT NULL,
        "description"          char(30) NOT NULL,
        "size"          char(18) NOT NULL,
        "color"      char(6) NOT NULL,
        "quantity"             integer NOT NULL,
        "unit_price"           numeric(15,2) NOT NULL,
        PRIMARY KEY ("id"),
);
CREATE TABLE "DBA"."department"
(
        "dept_id"              integer NOT NULL,
        "dept_name"            char(40) NOT NULL,
        "dept_head_id"         integer,
        PRIMARY KEY ("dept_id"),
);
CREATE TABLE "DBA"."employee"
(
        "emp_id"               integer NOT NULL,
        "manager_id"           integer,
        "emp_fname"            char(20) NOT NULL,
        "emp_lname"            char(20) NOT NULL,
        "dept_id"              integer NOT NULL,
        "street"               char(40) NOT NULL,
        "city"          char(20) NOT NULL,
        "state"      char(4) NOT NULL,
        "zip_code"             char(9) NOT NULL,
        "phone"      char(10),
        "status"               char(1),
        "ss_number"            char(11) NOT NULL,
        "salary"               numeric(20,3) NOT NULL,
        "start_date"           date NOT NULL,
        "termination_date"             date,
        "birth_date"           date,
        "bene_health_ins"              char(1),
```

```
        "bene_life_ins"      char(1),
        "bene_day_care"      char(1),
        "sex"           char(1),
        PRIMARY KEY ("emp_id"),
);
commit work;


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%    Reload data
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

INPUT INTO "DBA"."sales_order"
        FROM e:dbfilesdocsamplunload117.dat
        FORMAT ASCII
        BY ORDER;

INPUT INTO "DBA"."sales_order_items"
        FROM e:dbfilesdocsamplunload118.dat
        FORMAT ASCII
        BY ORDER;

INPUT INTO "DBA"."contact"
        FROM e:dbfilesdocsamplunload119.dat
        FORMAT ASCII
        BY ORDER;

INPUT INTO "DBA"."customer"
        FROM e:dbfilesdocsamplunload120.dat
        FORMAT ASCII
        BY ORDER;

INPUT INTO "DBA"."fin_code"
        FROM e:dbfilesdocsamplunload121.dat
        FORMAT ASCII
        BY ORDER;

INPUT INTO "DBA"."fin_data"
        FROM e:dbfilesdocsamplunload122.dat
        FORMAT ASCII
        BY ORDER;

INPUT INTO "DBA"."product"
        FROM e:dbfilesdocsamplunload123.dat
        FORMAT ASCII
        BY ORDER;

INPUT INTO "DBA"."department"
        FROM e:dbfilesdocsamplunload124.dat
        FORMAT ASCII
        BY ORDER;

INPUT INTO "DBA"."employee"
        FROM e:dbfilesdocsamplunload125.dat
        FORMAT ASCII
        BY ORDER;

commit work;


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%    Add foreign keys definitions
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


ALTER TABLE "DBA"."sales_order"
        ADD FOREIGN KEY "ky_so_employee_id" ("sales_rep")
REFERENCES "DBA"."employee" ("emp_id");

ALTER TABLE "DBA"."sales_order"
        ADD FOREIGN KEY "ky_so_fincode" ("fin_code_id") REFERENCES
"DBA"."fin_code" ("code") on delete set null;

ALTER TABLE "DBA"."sales_order"
        ADD FOREIGN KEY "ky_so_customer" ("cust_id") REFERENCES
"DBA"."customer" ("id");
CREATE INDEX "ix_sales_cust" ON "DBA"."sales_order"
(
        "cust_id" ASC
);

ALTER TABLE "DBA"."sales_order_items"
        ADD FOREIGN KEY "ky_prod_id" ("prod_id") REFERENCES
"DBA"."product" ("id");

ALTER TABLE "DBA"."sales_order_items"
        ADD FOREIGN KEY "id_fk" ("id") REFERENCES
"DBA"."sales_order" ("id") on delete cascade;
CREATE INDEX "ix_item_prod" ON "DBA"."sales_order_items"
(
        "prod_id" ASC
);
CREATE INDEX "ix_cust_name" ON "DBA"."customer"
(
        "lname" ASC,
        "fname" ASC
);

ALTER TABLE "DBA"."fin_data"
        ADD FOREIGN KEY "ky_code_data" ("code") REFERENCES
"DBA"."fin_code" ("code") on delete cascade;
CREATE INDEX "fin_data_idx" ON "DBA"."fin_data"
(
        "code" ASC
);
CREATE INDEX "ix_prod_name" ON "DBA"."product"
(
        "name" ASC);
CREATE INDEX "ix_prod_desc" ON "DBA"."product"
(
        "description" ASC);
CREATE INDEX "ix_prod_size" ON "DBA"."product"
(
        "size" ASC);
CREATE INDEX "ix_prod_color" ON "DBA"."product"
(
        "color" ASC
);

ALTER TABLE "DBA"."department"
        ADD FOREIGN KEY "ky_dept_head" ("dept_head_id") REFERENCES
"DBA"."employee" ("emp_id") on delete set null;
```

```
ALTER TABLE "DBA"."employee"
        ADD FOREIGN KEY "ky_dept_id" ("dept_id") REFERENCES
"DBA"."department" ("dept_id");
commit work;


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%    Create views
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

commit work;


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%    Set option values
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


SET OPTION Statistics =;
SET OPTION Date_order =;


%
%SQL Option Statements for user
%

SET OPTION "PUBLIC"."Blocking" = 'On';
SET OPTION "PUBLIC"."Checkpoint_time" = '60';
SET OPTION "PUBLIC"."Conversion_error" = 'On';
SET OPTION "PUBLIC"."Timestamp_format" = 'YYYY-MM-DD HH:NN:SS.SSS';
SET OPTION "PUBLIC"."Time_format" = 'HH:NN:SS.SSS';
SET OPTION "PUBLIC"."Date_format" = 'YYYY-MM-DD';
SET OPTION "PUBLIC"."Date_order" = 'YMD';
SET OPTION "PUBLIC"."Isolation_level" = '0';
SET OPTION "PUBLIC"."Precision" = '30';
SET OPTION "PUBLIC"."Recovery_time" = '2';
SET OPTION "PUBLIC"."Row_counts" = 'Off';
SET OPTION "PUBLIC"."Scale" = '6';
SET OPTION "PUBLIC"."Thread_count" = '0';
SET OPTION "PUBLIC"."Wait_for_commit" = 'Off';
SET OPTION "PUBLIC"."Auto_commit" = 'Off';
SET OPTION "PUBLIC"."Auto_refetch" = 'On';
SET OPTION "PUBLIC"."Bell" = 'On';
SET OPTION "PUBLIC"."Commit_on_exit" = 'On';
SET OPTION "PUBLIC"."Echo" = 'On';
SET OPTION "PUBLIC"."Headings" = 'On';
SET OPTION "PUBLIC"."Input_format" = 'ASCII';
SET OPTION "PUBLIC"."ISQL_log" = '';
SET OPTION "PUBLIC"."NULLS" = '(NULL)';
SET OPTION "PUBLIC"."On_error" = 'Prompt';
SET OPTION "PUBLIC"."Output_format" = 'ASCII';
SET OPTION "PUBLIC"."Output_length" = '0';
SET OPTION "PUBLIC"."Screen_format" = 'Text';
SET OPTION "PUBLIC"."Statistics" = '3';
SET OPTION "PUBLIC"."Truncation_length" = '30';
SET OPTION "PUBLIC"."Command_delimiter" = ';';


%
%SQL Option Statements for user
%
```

```
SET OPTION "DBA"."Wait_for_commit" = 'off';
SET OPTION "DBA"."Statistics" = '3';
SET OPTION "DBA"."Date_order" = 'YMD';
commit work;


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%    Create procedures
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

CONNECT "DBA" IDENTIFIED BY "SQL";

SET TEMPORARY OPTION Command_delimiter = ';;';

create procedure
sp_retrieve_contacts()
result(id integer,last_name char(15),first_name char(15),title
char(2),street char(30),city char(20),state char(2),zip
char(5),phone char(10),fax char(10))
begin
  select
id,last_name,first_name,title,street,city,state,zip,phone,fax
    from contact order by contact.id asc
end;;


create procedure
sp_product_info(inout prod_id integer)
result(id integer,name char(15),description char(30),size
char(18),color char(6),quantity integer,unit_price
decimal(15,2),picture_name char(12))
begin
  select
id,name,description,size,color,quantity,unit_price,picture_name
    from product where id=prod_id
end;;


create procedure
sp_customer_list()
result(id integer,company_name char(35))
begin
  select id,company_name from customer
end;;


create procedure
sp_contacts(in action char(1),in contact_id integer,in
contact_old_id integer,in contact_last_name char(15),in
contact_first_name char(15),in contact_title char(2),in
contact_street char(30),in contact_city char(20),in contact_state
char(2),in contact_zip char(5),in contact_phone char(10),in
contact_fax char(10))
begin
  case action when 'I' then
    insert into
contact(id,last_name,first_name,title,street,city,state,zip,
      phone,fax)
values(contact_id,contact_last_name,contact_first_name,
      contact_title,contact_street,contact_city,contact_state,cont-
```

```
act_zip,
      contact_phone,contact_fax) when 'U' then
    update contact set
contact.id=contact_id,contact.last_name=contact_last_name,
      contact.first_name=contact_first_name,contact.title=contact_-
title,
      contact.street=contact_street,contact.city=contact_city,cont-
act.state=contact_state,
      contact.zip=contact_zip,contact.phone=contact_phone,contact.-
fax=contact_fax
      where contact.id=contact_old_id when 'D' then
    delete from contact where contact.id=contact_old_id
  end case
end;;


create procedure
sp_sales_order_items(in ord_id integer,in product integer)
result(line_id integer,prod_id integer,quantity integer,ship_date
date)
begin
  select line_id,prod_id,quantity,ship_date from sales_order_items
where id
    =ord_id
end;;


create procedure
sp_sales_order(in customer_id integer,in product_id integer)
result(id integer,order_date date,fin_code_id char(2),region
char(7),sales_rep integer)
begin
  select s.id,s.order_date,s.fin_code_id,s.region,s.sales_rep
    from sales_order as s,sales_order_items as i where
s.cust_id=customer_id and i.prod_id
    =product_id and s.id=i.id
end;;


create procedure
sp_customer_products(inout customer_id integer)
result(id integer,quantity_ordered integer)
begin
  select product.id,sum(sales_order_items.quantity) from product
    ,sales_order_items,sales_order where
sales_order.cust_id=customer_id
    and sales_order.id=sales_order_items.id and
sales_order_items.prod_id=product.id
    group by product.id
end;;

SET TEMPORARY OPTION Command_delimiter = ';';;

CONNECT "DBA" IDENTIFIED BY "SQL";

commit work;


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%    Create triggers
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

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

SET TEMPORARY OPTION Command_delimiter = ';;';

create trigger
tr_manager before update of dept_head_id on department
referencing old as old_dept new as new_dept
for each row
begin
  update employee set employee.manager_id=new_dept.dept_head_id
    where employee.dept_id=old_dept.dept_id
end;;


create trigger
tr_dept_id after update of dept_id on department
referencing old as old_dept new as new_dept
for each row
begin
  update employee set employee.dept_id=new_dept.dept_id where
employee.dept_id
    =old_dept.dept_id
end;;

SET TEMPORARY OPTION Command_delimiter = ';';;

CONNECT "DBA" IDENTIFIED BY "SQL";

commit work;
```

Chapter 42

Differences from Other SQL Dialects

About this chapter

SQL Anywhere conforms to the ANSI SQL89 standard but has many additional features defined in IBM's DB2 and SAA specification, and in ANSI SQL92.

This chapter describes those features of SQL Anywhere that are not commonly found in SQL implementations. The following SQL Anywhere features are not found in many other SQL implementations.

- Full type conversion is implemented. Any data type can be compared with or used in any expression with any other data type.
- SQL Anywhere has date, time and timestamp types that includes a year, month and day, hour, minutes, seconds and fraction of a second. For insertions or updates to date fields, or comparisons with date fields, a free format date is supported.

  In addition, the following operations are allowed on dates:

  date + *integer*
  : Add the specified number of days to a date.

  date - *integer*
  : Subtract the specified number of days from a date.

  date - *date*
  : Compute the number of days between two dates.

  date + *time*
  : Make a timestamp out of a date and time.

  Also, many functions are provided for manipulating dates and times. See "Functions" on page 765 for a description of these.

- SQL Anywhere supports both entity and referential integrity. This has been implemented via the following two extensions to the CREATE TABLE and ALTER TABLE commands.

```
PRIMARY KEY ( column-name, ... )

[NOT NULL] FOREIGN KEY [role-name] [(column-name, ...)]
           REFERENCES table-name [(column-name, ...)]
           [ CHECK ON COMMIT ]
```

The PRIMARY KEY clause declares the primary key for the relation. SQL Anywhere will then enforce the uniqueness of the primary key, and ensure that no column in the primary key contains the NULL value.

The FOREIGN KEY clause defines a relationship between this table and another table. This relationship is represented by a column (or columns) in this table which must contain values in the primary key of another table. The system will then ensure referential integrity for these columns — whenever these columns are modified or a row is inserted into this table, these columns will be checked to ensure that either they are all NULL or the values match the corresponding columns for some row in the primary key of the other table. For more information, see "CREATE TABLE Statement" on page 859.

- SQL Anywhere allows **automatic joins** between tables. In addition to the NATURAL and OUTER join operators supported in other implementations, SQL Anywhere allows KEY joins between tables based on foreign key relationships. This reduces the complexity of the WHERE clause when performing joins.
- SQL Anywhere allows more than one table to be referenced by the UPDATE command. Views defined on more than one table can also be updated. Many SQL implementations will not allow updates on joined tables.
- The ALTER TABLE command has been extended. In addition to changes for entity and referential integrity, the following types of alterations are allowed:

```
ADD      column  data-type
MODIFY   column  data-type
DELETE   column
RENAME   new-table-name
RENAME   old-column  TO  new-column
```

The MODIFY can be used to change the maximum length of a character column as well as converting from one data type to another. For more information, see "ALTER TABLE Statement" on page 820.

- SQL Anywhere allows subqueries to appear wherever expressions are allowed. Many SQL implementations only allow subqueries on the right side of a comparison operator. For example, the following command is valid in SQL Anywhere but not valid in most other SQL implementations.

```
SELECT   emp_lname,
         emp_birthdate,
         ( SELECT     skill
           FROM       department
           WHERE      emp_id = employee.emp_ID
           AND        dept_id = 200 )
FROM     employee
```

- SQL Anywhere supports several functions not in the ANSI SQL definition. See "Functions" on page 765 for a full list of available functions.
- When using Embedded SQL, cursor positions can be moved arbitrarily on the FETCH statement. Cursors can be moved forward or backward relative to the

current position or a given number of records from the beginning or end of the cursor.

Chapter 43

SQL Anywhere Limitations

SQL Anywhere has been designed with as few limitations as possible. The following limitations do exist, but the memory and disk drive of the microcomputer are more limiting factors in most cases.

database size	2 GB per file, 12 files per database
number of tables per database	32,767
number of tables referenced per transaction	no limit
table size	2 GB—must be in one file *
number of columns per table	999 (using 4K pages)
number of rows per table	limited by table size
row size	limited by table size
number of rows per database	limited by file size
field size	2 GB
number of indexes	32,767 per table
maximum index entry size	no limit

* User-created indexes for the table can be stored separately from the table. The 2 GB limit for table size does not apply to NT databases using the NTFS file system. For this platform only, the maximum table size is the largest file size allowed by the operating system.

Chapter 44

SQL Anywhere Keywords

About this chapter

This chapter lists SQL keywords reserved by SQL Anywhere. If any of these keywords are to be used as an identifier they should be enclosed in quotation marks.

A number of these keywords are not used in the current implementation of SQL Anywhere but are reserved for future enhancements.

absolute	encrypted	names	row
add	end	natural	rows
address	endif	new	save
after	escape	next	savepoint
ahead	escapes	no	schedule
all	every	noholdlock	scroll
allocate	exception	nonclustered	section
alter	exclusive	not	select
and	exec	notfound	send
any	execute	null	sent
array	exists	numeric	set
as	explain	of	setting
asc	external	off	share
at	false	offset	signal
atomic	fetch	old	smallint
autoincrement	first	on	some
before	float	only	sql
begin	for	open	sqlca
between	foreign	optimizer	sqlcode
binary	format	option	sqlerror
bind	found	options	sqlstate
block	from	or	sqlwarning
both	full	order	start
break	function	others	statement
by	get	out	statistics
cache	global	outer	stop
call	go	output	strip
cascade	goto	passthrough	subscribe
case	grant	plan	subscription

cast
char
character
check
checkpoint
close
column
comment
commit
confirm
connect
connection
consolidate
constraint
continue
convert
create
cross
current
cursor
data
database
datatype
date
dba
dbspace
dec
decimal
declare
default
delete
delimited
desc
describe
descriptor
dictionary
disconnect
distinct
do
double
drop
dynamic
each
else
elseif
group
having
hold
holdlock
identified
identity
if
immediate
in
include
index
indicator
inner
inout
input
insensitive
insert
instead
int
integer
into
is
isolation
join
key
labels
last
leave
left
level
like
list
load
local
lock
log
long
loop
match
membership
message
mode
modify
name
named
precision
prepare
preserve
primary
print
prior
privileges
proc
procedure
publication
publish
publisher
purge
put
quotes
raiserror
read
readtext
real
received
recompile
reference
references
referencing
relative
release
remote
rename
reopen
replicate
rereceive
resend
reset
resignal
resolve
resource
restrict
result
resume
return
returns
revoke
right
rollback
subtrans
subtransaction
synchronization
synchronize
table
temporary
textptr
then
time
timestamp
tinyint
to
tran
transaction
trigger
true
truncate
tsequal
type
union
unique
unknown
unload
update
use
user
using
validate
value
values
varbinary
varchar
variable
variables
varying
verify
view
when
whenever
where
while
with
work
writetext

If you are using embedded SQL, the database library function **sql_needs_quotes** can be used to determine if a string requires quotation marks to make it an identifier because it is a reserved word or contains invalid identifier characters.

Chapter 45

SQL Anywhere System Procedures and Functions

About this chapter

This chapter documents the system-supplied catalog stored procedures in SQL Anywhere databases to retrieve system information, and the system-supplied extended procedures, including procedures for sending e-mail messages on a MAPI e-mail system.

Contents

45.1 System procedure overview

SQL Anywhere includes the following kinds of system procedures:

- Catalog stored procedures, for displaying system information in tabular form.
- Extended stored procedures for MAPI e-mail support.
- Transact-SQL system and catalog procedures. For a list of these system procedures see "SQL Server system and catalog procedures" on page 493.
- System functions that are implemented as stored procedures. These are included with other system functions in "System functions" on page 781.

This chapter documents the catalog stored procedures and the extended stored procedures for MAPI e-mail support and other external functions.

45.2 Catalog stored procedures

The following catalog procedures return result sets displaying database engine, database, and connection properties in tabular form.

sa_db_info([database-id])
: Returns a single row containing the Number, Alias, File, ConnCount, PageSize, and LogName for the specified database.

sa_conn_info([connection-id])
: Returns the Number, Name, Userid, DBNumber, LastReqTime, ProcessTime, Port, ReqType, CommLink, NodeAddr, LastIdle, CurrTaskSw, BlockedOn, and UncmtOps properties for each connection. If no connection-id is supplied, information for all current connections to databases on the server is returned.

sa_db_properties([database-id])
: Returns the database id number and the Number, PropNum, PropName, PropDescription, and Value, for each property returned by the **sa_db_info** system procedure.

sa_eng_properties()
: Returns the PropNum, PropName, PropDescription, and Value for each available engine property. For a listing of available engine properties, see "System functions" on page 781.

sa_conn_properties([connection-id])
: Returns the connection id as Number, and the PropNum, PropName, PropDescription, and Value for each available

connection property. For a listing of available connection properties, see "System functions" on page 781.

These system procedures are owned by the "DBO" user ID. The PUBLIC group has EXECUTE permission on these procedures.

45.3 System extended stored procedures

A set of system extended procedures are included into SQL Anywhere databases. These procedures are owned by the DBO user ID that owns the Transact-SQL-compatibility system views.

The following sections describe each of the stored procedures.

45.3.1 MAPI system extended stored procedures

SQL Anywhere includes a set of three system procedures for sending electronic mail using Microsoft's Messaging API standard (MAPI). These system procedures are implemented as Extended Stored Procedures: each of the procedures calls a function in an external DLL.

In order to use the MAPI stored procedures, a MAPI e-mail system must be accessible from the database server machine.

The MAPI stored procedures are:

- xp_startmail Starts a mail session in a specified mail account by logging on the MAPI message system.
- xp_sendmail Sends a mail message to specified users.
- xp_stopmail Closes the mail session.

The following procedure notifies a set of people that a backup has been completed.

```
CREATE PROCEDURE notify_backup()
BEGIN
  CALL xp_startmail( mail_user='ServerAccount',
                     mail_password='ServerPassword'
                   );
  CALL xp_sendmail( recipient='IS Group',
                    subject='Backup',
                    "message"='Backup completed'
                   );
  CALL xp_stopmail( )
END
```

The MAPI system procedures are discussed in the following sections.

xp_startmail system procedure

Syntax

```
[ [ variable = ] CALL ] xp_startmail (
   . . .    [ mail_user = mail-login-name ]
   . . .    [, mail_password = mail-password ]
   . . .    )
```

Purpose

To start an e-mail session.

Usage

Anywhere.

Authorization

None.

Description

xp_startmail is a system stored procedure that starts an e-mail session. It is implemented as a user-defined function.

xp_startmail returns an integer. The return value is zero if the login is successful, and non-zero otherwise.

The *mail-login-name* and *mail-password* values are strings containing the MAPI login name and password to be used in the mail session.

Return codes

The xp_startmail system procedure uses the following return codes:

Return Code	Meaning
0	success
3	failure

xp_sendmail system procedure

Syntax

```
[ [ variable = ] CALL ] xp_sendmail (
    ...     [ recipient = mail-address ]
    ...     [, cc_recipient = mail-address ]
    ...     [, bcc_recipient = mail-address ]
    ...     [, "message" = message-body ]
    ...     [, include-file = file-name ]
    ...     )
```

Purpose

To send an e-mail message.

Usage

Anywhere.

Authorization

Must have executed **xp_startmail** to start an e-mail session.

Description

xp_sendmail is a system stored procedure that sends an e-mail message once a session has been started using **sp_startmail**. It is implemented as a user-defined function.

xp_sendmail returns an integer. The return value is zero if the message is successfully sent, and non-zero otherwise.

The argument values are strings. The *message* parameter name requires double quotes around it as MESSAGE is a keyword.

Return codes

The xp_sendmail system procedure uses the following return codes:

Return Code	Meaning
0	success
5	failure (general)
11	ambiguous recipient
12	attachment not found
13	disk full
14	insufficient memory
15	invalid session
16	text too large
17	too many files
18	too many recipients
19	unknown recipient

Example

The following call sends a message to the user ID **Sales Group** containing the file **prices.doc** as a mail attachment:

```
CALL xp_sendmail(recipient='Sales Group',
                 subject='New Pricing',
                 include_file = 'c:\docs\prices.doc' )
```

xp_stopmail system procedure

Syntax

[variable =] [CALL] xp_stopmail ()

Purpose

To close an e-mail session.

Usage

Anywhere.

Authorization

None.

Description

xp_stopmail is a system stored procedure that starts an e-mail session. It is implemented as a user-defined function.

xp_stopmail returns an integer. The return value is zero if the mail session is successfully closed, and non-zero otherwise.

Return codes

The xp_stopmail system procedure uses the following return codes:

Return Code	Meaning
0	success
2	failure

45.3.2 Other system extended stored procedures

The other system extended stored procedures included are:

xp_cmdshell Executes a system command.

xp_sprintf Takes as input a format string and a set of string arguments, and returns a string.

xp_scanf Takes as input a string and a format string, and returns a set of strings.

The following sections provide more detail on each of these procedures.

xp_cmdshell system procedure

Syntax

[variable = CALL] xp_cmdshell (string)

Purpose

To carry out an operating system command from a procedure.

Usage

Anywhere.

Authorization

None.

Description

xp_cmdshell is a system stored procedure that executes a system command and then returns control to the calling environment,

Example

The following statement lists the files in the current directory in the file **c:\temp.txt**

```
xp_cmdshell('dir > c:\\temp.txt')
```

xp_sprintf system procedure

Syntax

```
[ variable = CALL ] xp_sprintf ( out-string,
    . . .    format-string
    . . .    [ input-string, ... ] )
```

Purpose

To build up a string from a format string and a set of input strings.

Usage

Anywhere.

Authorization

None.

Description

xp_sprintf is a system stored procedure that builds up a string from a format string and a set of input strings. The format-string can contain up to fifty string placeholders (`%s`). These placeholders are filled in by the `input-string` arguments.

All arguments must be strings of less than 254 characters.

Example

The following statements put the string `Hello World!` into the variable **mystring**.

```
CREATE VARIABLE mystring CHAR(254) ;
xp_sprintf( mystring, 'Hello %s', 'World!' )
```

xp_scanf system procedure

Syntax

```
[ variable = CALL ] xp_scanf ( in-string,
   . . .    format-string
   . . .    [ output-string, ... ] )
```

Purpose

To extract substrings from an input string and a format string.

Usage

Anywhere.

Authorization

None.

Description

xp_scanf is a system stored procedure that extracts substrings from an input string and a format string. The format-string can contain up to fifty string placeholders (`%s`). The value of these placeholders in the are filled in by the `output-string` arguments.

All arguments must be strings of less than 254 characters.

Example

The following statements put the string `World!` into the variable **mystring**.

```
CREATE VARIABLE mystring CHAR(254) ;
xp_scanf( 'Hello World!', 'Hello %s', mystring )
```

Chapter 46

SQL Anywhere System Tables

About this chapter

The structure of every SQL Anywhere database is described in a number of *system tables*.

The Entity-Relationship diagram on the next page shows all the system tables and the foreign keys connecting the system tables.

The system tables are owned by the **SYS** user ID. The contents of these tables can only be changed by the database system. Thus, the UPDATE, DELETE, and INSERT commands cannot be used to modify the contents of these tables. Further, the structure of these tables cannot be changed using the ALTER TABLE and DROP commands.

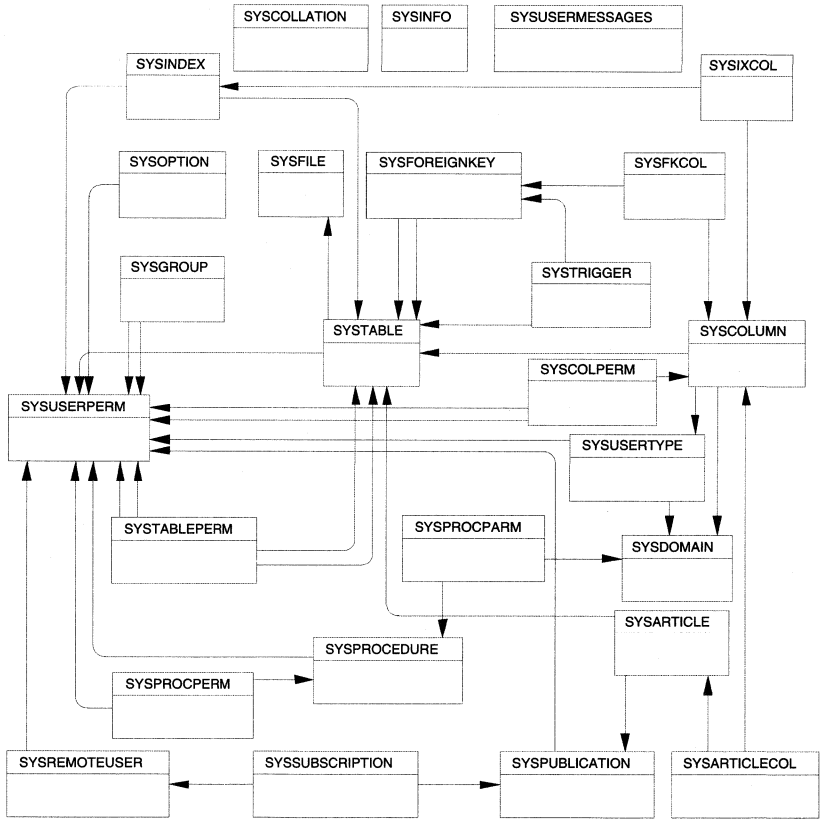
Following a diagram of the system tables, each table is listed alphabetically.

The system tables are described via the CREATE TABLE commands used to create them. They serve as good examples of how tables are created in SQL. Following the CREATE TABLE command, each column is briefly described.

Several of the columns have only two possible values. Usually these values are "Y" and "N" for "yes" and "no" respectively. These columns are designated by "(Y/N)".

46.1 System tables diagram

The SQL Anywhere system tables are shown in the following diagram. Foreign key relations between tables are indicated by arrows: the arrow leads from the foreign table to the primary table.



46.2 DUMMY system table

```
CREATE TABLE SYS.DUMMY (
    dummy_col           INT NOT NULL
)
```

The **DUMMY** table is provided as a table that always has exactly one row. This can be useful for extracting information from the database, as in the following example that gets the current user ID and the current date from the database.

```
SELECT USER, today(*) FROM SYS.DUMMY
```

dummy_col This column is not used. It is present because a table cannot be created with no columns.

46.3 SYSARTICLE system table

```
CREATE TABLE SYS.SYSARTICLE (
    publication_id  SMALLINT        NOT NULL,
    table_id        SMALLINT        NOT NULL,
    where_expr      LONG VARCHAR,
    subscribe_by_expr   LONG VARCHAR,
    PRIMARY KEY     ( publication_id, table_id ),
    FOREIGN KEY     REFERENCES SYS.SYSPUBLICATION,
    FOREIGN KEY     REFERENCES SYS.SYSTABLE
    )
```

Each row of **SYSUSERPERM** describes an article in a SQL Remote publication.

publication_id
The publication of which this article is a part.

table_id Each article consists of columns and rows from a single table. This column contains the table id for this table.

where_expr For articles that contain a subset of rows defined by a WHERE clause, this column contains the search condition.

subscribe_by_expr
For articles that contain a subset of rows defined by a SUBSCRIBE BY expression, this column contains the expression.

46.4 SYSARTICLECOL system table

```
CREATE TABLE SYS.SYSARTICLECOL (
    publication_id  SMALLINT        NOT NULL,
    table_id        SMALLINT        NOT NULL,
    column_id       SMALLINT        NOT NULL,
    PRIMARY KEY     (publication_id, table_id, column_id),
    FOREIGN KEY     REFERENCES SYS.SYSARTICLE,
    FOREIGN KEY     REFERENCES SYS.SYSCOLUMN
    )
```

Each row identifies a column in an article, identifying the column, the table it is in, and the publication it is part of.

publication_id
A unique identifier for the publication of which the column is a part.

table_id The table to which the column belongs.

column_id The column identifier, from the SYSCOLUMN system table.

46.5 SYSCOLLATE system table

```
CREATE TABLE SYS.SYSCOLLATION (
    collation_id        SMALLINT NOT NULL,
    collation_name      CHAR(128) NOT NULL,
    collation_label     CHAR(10) NOT NULL,
    collation_order     BINARY(256) NOT NULL,
    PRIMARY KEY         ( collation_id )
)
```

This table contains the collation sequences available to SQL Anywhere. There is no way to modify the contents of this table.

collation_id A unique number identifying the collation sequence. The collation sequence with **collation_id** equal 2 is the sequence used in previous versions of SQL Anywhere, and is the default when a database is created with DBINIT.

collation_name
The name of the collation sequence.

collation_label
A string identifying each of the available collation sequences. The collation sequence to be used is selected when the database is created by specifying the collation label with the -z option.

collation_order
An array of bytes defining how each of the 256 character codes are treated for comparison purposes. All string comparisons translate each character according to the collation order table before comparing the characters. For the different ASCII code pages, the only difference is how accented characters are sorted. In general, an accented character is sorted as if it were the same as the nonaccented character.

NOTE: This table does not exist in databases created with Watcom SQL Version 3.1 or earlier and only exists if the -z option is specified when the database is created.

46.6 SYSCOLPERM system table

```
CREATE TABLE SYS.SYSCOLPERM (
    table_id            SMALLINT NOT NULL,
    grantee             SMALLINT NOT NULL,
    grantor             SMALLINT NOT NULL,
    column_id           SMALLINT NOT NULL,
    PRIMARY KEY         ( table_id, grantee,
                          grantor, column_id ),
    FOREIGN KEY         grantee ( grantee ) REFERENCES
                        SYS.SYSUSERPERM ( user_id ),
    FOREIGN KEY         grantor ( grantor ) REFERENCES
                        SYS.SYSUSERPERM ( user_id ),
    FOREIGN KEY         REFERENCES SYS.SYSCOLUMN
)
```

The GRANT command can give UPDATE permission to individual columns in a table. Each column with UPDATE permission is recorded in one row of **SYSCOLPERM**.

table_id The table number for the table containing the column.

grantee The user number of the user ID given UPDATE permission on the column. If the **grantee** is the user number for the special **PUBLIC** user ID, the UPDATE permission is given to all user IDs.

grantor The user number of the user ID granting the permission.

column_id This column number, together with the **table_id**, identifies the column for which UPDATE permission has been granted.

46.7 SYSCOLUMN system table

```
CREATE TABLE SYS.SYSCOLUMN (
    table_id            SMALLINT NOT NULL,
    column_id           SMALLINT NOT NULL,
    pkey                CHAR(1) NOT NULL,
    domain_id           SMALLINT NOT NULL,
    nulls               CHAR(1) NOT NULL,
    width               SMALLINT NOT NULL,
    scale               SMALLINT NOT NULL,
    estimate            INT NOT NULL,
    column_name         CHAR(128) NOT NULL,
    remarks             LONG VARCHAR,
    "default"           LONG VARCHAR,
    "check"             LONG VARCHAR,
    user_type           SMALLINT,
    format_str          CHAR(128),
    PRIMARY KEY         ( table_id, column_id ),
    FOREIGN KEY         REFERENCES SYS.SYSTABLE,
    FOREIGN KEY         REFERENCES SYS.SYSDOMAIN
)
```

Each column in every table or view is described by one row in **SYSCOLUMN**.

table_id The table number uniquely identifies the table or view to which this column belongs.

column_id Each table starts numbering columns at 1. The order of column numbers determines the order that columns are displayed in the command select * from table.

pkey (Y/N) Indicate whether this column is part of the primary key for the table.

domain_id Identify the data type for the column by the data type number listed in the **SYSDOMAIN** table.

nulls (Y/N) Indicate whether the NULL value is allowed in this column.

width This column contains the length of string columns, the precision of numeric columns, and the number of bytes of storage for all other data types.

scale The number of digits after the decimal point for numeric data type columns, and zero for all other data types.

estimate A self-tuning parameter for the optimizer. SQL Anywhere will learn from previous queries by adjusting guesses that are made by the optimizer.

column_name
The name of the column.

remarks A comment string.

default The default value for the column. This value is only used when an INSERT statement does not specify a value for the column.

check. Any CHECK condition defined on the column.

user_type If the column is defined on a user-defined data type, the data type is held here.

format_str Currently unused.

46.8 SYSDOMAIN system table

```
CREATE TABLE SYS.SYSDOMAIN (
    domain_id           SMALLINT NOT NULL,
    domain_name         CHAR(128) NOT NULL,
    type_id             SMALLINT,
    precision           SMALLINT,
    PRIMARY KEY         ( domain_id )
)
```

Each of the predefined data types (sometimes called a *domain*) in SQL Anywhere is assigned a unique number. The **SYSDOMAIN** table is provided for informational purposes to show the association between these numbers and the appropriate data type. This table is never changed by SQL Anywhere.

domain_id The unique number assigned to each data type. These numbers cannot be changed.

domain_name
A string containing the data type normally found in the CREATE TABLE command, such as **char** or **integer**.

type_id The ODBC data type. This corresponds to "data_type" in the Transact-SQL-compatibility DBO.SYSTYPES table.

precision

46.9 SYSFILE system table

```
CREATE TABLE SYS.SYSFILE (
    file_id             SMALLINT NOT NULL,
    file_name           CHAR(80) NOT NULL,
    dbspace_name        CHAR(128) NOT NULL,
    PRIMARY KEY         ( file_id )
)
```

Every database consists of one or more operating system files. Each file is recorded in **SYSFILE**.

file_id Each file in a database is assigned a unique number. This file identifier is the primary key for **SYSFILE**. All system tables are stored in **file_id** 0.

file_name The database name is stored when a database is created. This name is for informational purposes only.

dbspace_name
Every file has a *dbspace name* that is unique. It is used in the CREATE TABLE command.

46.10 SYSFKCOL system table

```
CREATE TABLE SYS.SYSFKCOL (
    foreign_table_id    SMALLINT NOT NULL,
    foreign_key_id      SMALLINT NOT NULL,
    foreign_column_id   SMALLINT NOT NULL,
    primary_column_id   SMALLINT NOT NULL,
    PRIMARY KEY         ( foreign_table_id,
                          foreign_key_id, foreign_column_id ),
    FOREIGN KEY         REFERENCES SYS.SYSFOREIGNKEY,
    FOREIGN KEY         ( foreign_table_id,
                          foreign_column_id ) REFERENCES
                        SYS.SYSCOLUMN ( table_id, column_id )
)
```

Each row of **SYSFKCOL** describes the association between a **foreign column** in the foreign table of a relationship and the **primary column** in the primary table.

foreign_table_id
The table number of the foreign table.

foreign_key_id
The key number of the FOREIGN KEY for the foreign table. Together, **foreign_table_id** and **foreign_key_id** uniquely identify

one row in **SYSFOREIGNKEY**, and the table number for the primary table can be obtained from that row.

foreign_column_id
This column number, together with the **foreign_table_id**, identify the foreign column description in **SYSCOLUMN**.

primary_column_id
This column number, together with the **primary_table_id** obtained from **SYSFOREIGNKEY**, identify the primary column description in **SYSCOLUMN**.

46.11 SYSFOREIGNKEY system table

```
CREATE TABLE SYS.SYSFOREIGNKEY (
    foreign_table_id    SMALLINT NOT NULL,
    foreign_key_id      SMALLINT NOT NULL,
    primary_table_id    SMALLINT NOT NULL,
    root                INT NOT NULL,
    check_on_commit     CHAR(1) NOT NULL,
    nulls               CHAR(1) NOT NULL,
    role                CHAR(128) NOT NULL,
    remarks             LONG VARCHAR,
    PRIMARY KEY         ( foreign_table_id, foreign_key_id ),
    UNIQUE              ( role, foreign_table_id ),
    FOREIGN KEY         foreign_table ( foreign_table_id )
                        REFERENCES SYS.SYSTABLE ( table_id ),
    FOREIGN KEY         primary_table ( primary_table_id )
                        REFERENCES SYS.SYSTABLE ( table_id )
)
```

A foreign key is a *relationship* between two tables—the *foreign table* and the *primary table*. Every foreign key is defined by one row in **SYSFOREIGNKEY** and one or more rows in **SYSFKCOL**. **SYSFOREIGNKEY** contains general information about the foreign key while **SYSFKCOL** identifies the columns in the foreign key and associates each column in the foreign key with a column in the primary key of the primary table.

foreign_table_id
The table number of the foreign table.

foreign_key_id
Each foreign key has a **foreign key number** that is unique with respect to:

- The key number of all other foreign keys for the foreign table
- The key number of all foreign keys for the primary table
- The index number of all indexes for the foreign table

primary_table_id
The table number of the primary table.

root Foreign keys are stored in the database as B-trees. The **root** identifies the location of the root of the B-tree in the database file.

nulls (Y/N) Indicate whether the columns in the foreign key are allowed to contain the NULL value. Note that this setting is independent of the **nulls** setting in the columns contained in the foreign key.

check_on_commit (Y/N)
Indicate whether INSERT and UPDATE commands should wait until the next COMMIT command to check if foreign keys are valid. A foreign key is valid if, for each row in the foreign table, the values in the columns of the foreign key either contain the NULL value or match the primary key values in some row of the primary table.

role The name of the relationship between the foreign table and the primary table. Unless otherwise specified, the **role** name will be the same as the name of the primary table. The foreign table cannot have two foreign keys with the same role name.

remarks A comment string.

46.12 SYSGROUP system table

```
CREATE TABLE SYS.SYSGROUP (
    group_id            SMALLINT NOT NULL,
    group_member        SMALLINT NOT NULL,
    PRIMARY KEY         ( group_id, group_member ),
    FOREIGN KEY         group_id ( group_id ) REFERENCES
                        SYS.SYSUSERPERM ( user_id ),
    FOREIGN KEY         group_member ( group_member )
                        REFERENCES SYS.SYSUSERPERM ( user_id )
)
```

There is one row in **SYSGROUP** for every member of every group. This table describes a many-to-many relationship between groups and members. A group may have many members and a user may be a member of many groups.

group_id The user number of group.

group_member
The user number of a member.

46.13 SYSINDEX system table

```
CREATE TABLE SYS.SYSINDEX (
    table_id            SMALLINT NOT NULL,
    index_id            SMALLINT NOT NULL,
    root                INT NOT NULL,
    file_id             SMALLINT NOT NULL,
    "unique"            CHAR(1) NOT NULL,
    creator             SMALLINT NOT NULL,
    index_name          CHAR(128) NOT NULL,
    remarks             LONG VARCHAR,
    PRIMARY KEY         ( table_id, index_id ),
    UNIQUE              ( index_name, creator ),
    FOREIGN KEY         REFERENCES SYS.SYSTABLE,
    FOREIGN KEY         REFERENCES SYS.SYSFILE,
    FOREIGN KEY         ( creator ) REFERENCES
                        SYS.SYSUSERPERM ( user_id )
)
```

Each index in the database is described by one row in **SYSINDEX**. Each column in the index is described by one row in **SYSIXCOL**.

table_id The table number uniquely identifies the table to which this index applies.

index_id Each index for one particular table is assigned a unique index number.

root Indexes are stored in the database as B-trees. The **root** identifies the location of the root of the B-tree in the database file.

file_id The index is completely contained in the file with this **file_id** (see **SYSFILE**). In the current implementation of SQL Anywhere, this file is always the same as the file containing the table.

unique Indicate whether the index is a unique index ("Y"), a non-unique index ("N"), or a unique constraint ("U"). A unique index prevents two rows in the indexed table from having the same values in the index columns.

creator The user number of the creator of the index. In the current implementation of SQL Anywhere, this user is always the same as the creator of the table identified by table_id.

index_name The name of the index. A user ID cannot have two indexes with the same name.

remarks A comment string.

46.14 SYSINFO system table

```
CREATE TABLE SYS.SYSINFO (
    page_size           SMALLINT NOT NULL,
    encryption          CHAR(1) NOT NULL,
    blank_padding       CHAR(1) NOT NULL,
    case_sensitivity    CHAR(1) NOT NULL,
    default_collation   CHAR(10) NOT NULL,
    database_version    SMALLINT NOT NULL
)
```

This table indicates the database characteristics as defined when the database was created using DBINIT. It always contains only one row.

page_size The page size specified to DBINIT. The default value is 1024.

encryption The value "Y" or "N" depending on whether the -e switch was used with DBINIT.

blank_padding
The value "Y" or "N" depending on whether the -b switch was used with DBINIT.

case_sensitivity
The value "Y" or "N" depending on whether the -c switch was used with DBINIT. Note that case sensitivity affects both value comparisons and table and column name comparisons. For example, if case sensitivity is enabled, the system catalog names such as **SYSCATALOG** must be specified in upper case since that is how the name was spelled when it was created.

default_collation
A string corresponding to the **collation_label** in **SYSCOLLATE** corresponding to the collation sequence specified with DBINIT. The default value corresponds to the Multilingual collation sequence (code page 850), which was the default in Watcom SQL Version 3.1. The collation sequence is used for all string comparisons, including searches for character strings as well as column and table name comparison.

database_version
A small integer value indicating the database format. As newer versions of SQL Anywhere become available, new features may require that the format of the database file change. The version number allows SQL Anywhere software to determine if this database was created with a newer version of the software and thus cannot be understood by the software in use.

NOTE: This table does not exist in databases created with Watcom SQL Version 3.1 or earlier.

46.15 SYSIXCOL system table

```
CREATE TABLE SYS.SYSIXCOL (
    table_id            SMALLINT NOT NULL,
    index_id            SMALLINT NOT NULL,
    sequence            SMALLINT NOT NULL,
    column_id           SMALLINT NOT NULL,
    "order"             CHAR(1) NOT NULL,
    PRIMARY KEY         ( table_id, index_id, sequence )
    FOREIGN KEY         REFERENCES SYS.SYSINDEX,
    FOREIGN KEY         REFERENCES SYS.SYSCOLUMN
)
```

Every index has one row in **SYSIXCOL** for each column in the index.

table_id Identifies the table to which the index applies.

index_id Identifies in which index this column is used. Together, **table_id** and **index_id** identify one index described in **SYSINDEX**.

sequence Each column in an index is assigned a unique number starting at 0. The order of these numbers determines the relative significance of the columns in the index. The most important column has **sequence** number 0.

column_id The column number identifies which column is indexed. Together, **table_id** and **column_id** identify one column in **SYSCOLUMN**.

order (A/D) Indicate whether this column in the index is kept in ascending or descending order.

46.16 SYSOPTION system table

```
CREATE TABLE SYS.SYSOPTION (
    user_id             SMALLINT NOT NULL,
    "option"            CHAR(128) NOT NULL,
    "setting"           CHAR(80) NOT NULL,
    PRIMARY KEY         ( user_id, "option" ),
    FOREIGN KEY         REFERENCES SYS.SYSUSERPERM
)
```

Options settings are stored in the **SYSOPTION** table by the SET command. Each user can have their own setting for each option. In addition, settings for

the **PUBLIC** user ID define the default settings to be used for user IDs that do not have their own setting.

user_id The user number to whom this option setting applies.

option The name of the option.

setting The current setting for the named option.

46.17 SYSPROCEDURE system table

```
CREATE TABLE SYS.SYSPROCEDURE (
    proc_id             SMALLINT NOT NULL,
    creator             SMALLINT NOT NULL,
    proc_name           CHAR(128) NOT NULL,
    proc_defn           LONG VARCHAR,
    remarks             LONG VARCHAR,
    replicate           CHAR(1)     NOT NULL,
    PRIMARY KEY         ( proc_id ),
    UNIQUE              ( proc_name, creator ),
    FOREIGN KEY         ( creator ) REFERENCES
                        SYS.SYSUSERPERM ( user_id )
)
```

Each procedure in the database is described by one row in **SYSPROCEDURE**.

proc_id Each procedure is assigned a unique number (the **procedure number**) that is the primary key for **SYSPROCEDURE**.

creator This user number identifies the owner of the procedure. The name of the user can be found by looking in **SYSUSERPERM**.

proc_name The name of the procedure. One creator cannot have two procedures with the same name.

proc_defn The command used to create the procedure.

remarks A comment string.

replicate Holds a Y if the procedure is a primary data source in a Replication Server installation, or an N if not.

46.18 SYSPROCPARM system table

```
CREATE TABLE SYS.SYSPROCPARM (
    proc_id             SMALLINT NOT NULL,
    parm_id             SMALLINT NOT NULL,
    parm_type           SMALLINT NOT NULL,
    parm_mode_in        CHAR(1) NOT NULL,
    parm_mode_out       CHAR(1) NOT NULL,
    domain_id           SMALLINT NOT NULL,
    width               SMALLINT NOT NULL,
    scale               SMALLINT NOT NULL,
    parm_name           CHAR(128) NOT NULL,
    remarks             LONG VARCHAR,
    "default"           LONG VARCHAR,
    PRIMARY KEY         ( proc_id, parm_id ),
    FOREIGN KEY         REFERENCES SYS.SYSPROCEDURE,
    FOREIGN KEY         REFERENCES SYS.SYSDOMAIN
)
```

Each parameter to a procedure in the database is described by one row in **SYSPROCEDURE**.

proc_id The procedure number uniquely identifies the procedure to which this parameter belongs.

parm_id Each procedure starts numbering parameters at 1. The order of parameter numbers corresponds to the order in which they were defined.

parm_type The type of parameter will be one of the following:

- *0 - variable* normal parameter
- *1 - result* result variable - used with procedure that return result sets
- *2 - SQLSTATE* SQLSTATE error value
- *3 - SQLCODE* SQLCODE error value

parm_mode_in (Y/N)
Indicate whether this parameter supplies a value to the procedure (**IN** or **INOUT** parameters).

parm_mode_out (Y/N)
Indicate whether this parameter returns a value from the procedure (**OUT** or **INOUT** parameters).

domain_id Identify the data type for the parameter by the data type number listed in the **SYSDOMAIN** table.

width This column contains the length of string parameters, the precision of numeric parameters, and the number of bytes of storage for all other data types.

scale The number of digits after the decimal point for numeric data type parameters, and zero for all other data types.

parm_name The name of the parameter.

remarks A comment string.

default The default value for the parameter, held as a string.

46.19 SYSPROCPERM system table

```
CREATE TABLE SYS.SYSPROCPERM (
    proc_id             SMALLINT NOT NULL,
    grantee             SMALLINT NOT NULL,
    PRIMARY KEY         ( proc_id, grantee )
    FOREIGN KEY         ( grantee ) REFERENCES
                        SYS.SYSUSERPERM ( user_id ),
    FOREIGN KEY         REFERENCES SYS.SYSPROCEDURE
)
```

Only users who have been granted permission can call a procedure. Each row of the **SYSPROCPERM** table corresponds to one user granted permission to call one procedure.

proc_id The procedure number uniquely identifies the procedure for which permission has been granted.

grantee The user number of the user ID receiving the permission.

46.20 SYSPUBLICATION system table

```
CREATE TABLE SYS.SYSPUBLICATION (
    publication_id      SMALLINT    NOT NULL,
    creator             SMALLINT    NOT NULL,
    publication_name    CHAR(128)   NOT NULL,
    remarks             LONG VARCHAR,
    PRIMARY KEY         ( publication_id ),
    FOREIGN KEY         ( creator )
                        REFERENCES SYS.SYSUSERPERM (user_id )
    )
```

Each row describes a SQL Remote publication.

publication_id
A unique identifying number for the publication.

creator The owner of the publication.

publication_name
The name of the publication, which must be a valid identifier.

remarks Descriptive comments.

46.21 SYSREMOTEUSER system table

```
CREATE TABLE SYS.SYSREMOTEUSER (
    user_id          SMALLINT          NOT NULL,
    user_type        CHAR(1)           NOT NULL,
    address_type     CHAR(128)         NOT NULL,
    address          LONG BINARY       NOT NULL,
    time_sent        TIMESTAMP,
    log_sent         NUMERIC(20,0)     NOT NULL,
    confirm_sent     NUMERIC(20,0)     NOT NULL,
    resend_count     INTEGER           NOT NULL,
    time_received    TIMESTAMP,
    log_received     NUMERIC(20,0)     NOT NULL,
    confirm_received    NUMERIC(20,0),
    rereceive_count INTEGER      NOT NULL,
    PRIMARY KEY      (user_id),
    FOREIGN KEY      REFERENCES SYS.SYSUSERPERM
    )
```

Each row describes a userid with REMOTE permissions (a subscriber), together with the status of SQL Remote messages sent to and from that user.

user_id The user ID of the user with REMOTE permissions.

user_type The column contains either an R to indicate a user granted REMOTE permissions, or a C to indicate a user granted CONSOLIDATE permissions.

address_type Identifies which of the of the message systems supported by SQL Remote is to be used to send messages to this user.

address The address to which SQL Remote messages are to be sent. The address must be appropriate for the **address_type**

time_sent The time the most recent message was sent to this subscriber.

log_sent The log offset for the most recently sent operation.

confirm_sent The log offset for the most recently confirmed operation from this subscriber.

resend_count Counter to ensure messages are applied only once at the subscriber database.

time_received
The time the most recent message was received from this subscriber.

log_received The log offset in the subscriber's database for the operation most recently received at the current database.

confirm_received
The log offset in the subscriber's database for the operation most recently received at the current database.

rereceive_count
Counter to ensure messages are applied only once at the current database.

46.22 SYSSUBSCRIPTION system table

```
CREATE TABLE SYS.SYSSUBSCRIPTION (
    publication_id  SMALLINT        NOT NULL,
    user_id         SMALLINT        NOT NULL,
    subscribe_by    CHAR(128)       NOT NULL,
    created         NUMERIC(20,0)   NOT NULL,
    started         NUMERIC(20,0),
    PRIMARY KEY     (publication_id, user_id, subscribe_by),
    FOREIGN KEY     REFERENCES SYS.SYSPUBLICATION,
    FOREIGN KEY     REFERENCES SYS.SYSREMOTEUSER
    );
```

Each row describes a subscription from one user ID (which must have REMOTE permissions) to one publication.

publication_id
The identifier for the publication to which the user ID is subscribed.

user_id The user ID that is subscribed to the publication.

subscribe_by For publications with a SUBSCRIBE BY expression, this column holds the matching value for this subscription.

created The offset in the transaction log at which the subscription was created.

started The offset in the transaction log at which the subscription was started.

46.23 SYSTABLE system table

```
CREATE TABLE SYS.SYSTABLE (
    table_id            SMALLINT NOT NULL,
    file_id             SMALLINT NOT NULL,
    count               INTEGER NOT NULL,
    first_page          INT NOT NULL,
    last_page           INT NOT NULL,
    primary_root        INT NOT NULL,
    creator             SMALLINT NOT NULL,
    table_name          CHAR(128) NOT NULL,
    table_type          CHAR(10) NOT NULL,
    view_def            LONG VARCHAR,
    remarks             LONG VARCHAR,
    replicate           CHAR(1) NOT NULL,
    PRIMARY KEY         ( table_id ),
    UNIQUE              ( table_name, creator ),
    FOREIGN KEY         ( creator ) REFERENCES
                        SYS.SYSUSERPERM ( user_id ),
    FOREIGN KEY         REFERENCES SYS.SYSFILE
)
```

Each row of **SYSTABLE** describes one table or view in the database.

table_id Each table or view is assigned a unique number (the **table number**) that is the primary key for **SYSTABLE**.

file_id The file number indicates which database file contains the table. The **file_id** is a FOREIGN KEY for **SYSFILE**.

count The number of rows in the table is updated during each successful CHECKPOINT. This number is used by SQL Anywhere when optimizing database access. The **count** is always 0 for a view.

first_page Each SQL Anywhere database is divided into a number of fixed size pages. This value identifies the first page containing information for this table, and is used internally to find the start of this table. The **first_page** is always 0 for a view.

last_page The last page containing information for this table. The **last_page** is always 0 for a view.

primary_root Primary keys are stored in the database as B-trees. The **primary_root** locates the root of the B-tree for the primary key for the table. It will be 0 for a view and for a table with no primary key.

creator This user number identifies the owner of the table or view. The name of the user can be found by looking in **SYSUSERPERM**.

table_name The name of the table or view. One creator cannot have two tables or views with the same name.

table_type This column will be "BASE" for base tables and "VIEW" for views. It will be "GBL TEMP" for global temporary tables and "LCL TEMP" for local temporary tables.

view_def For a view, this column contains the CREATE VIEW command used to create the view. For a table, this column will contain any CHECK constraints for the table.

remarks A comment string.

replicate Holds a Y if the table is a primary data source in a Replication Server installation, or an N if not.

46.24 SYSTABLEPERM system table

```
CREATE TABLE SYS.SYSTABLEPERM (
    stable_id           SMALLINT NOT NULL,
    grantee             SMALLINT NOT NULL,
    grantor             SMALLINT NOT NULL,
    ttable_id           SMALLINT NOT NULL,
    selectauth          CHAR(1) NOT NULL,
    insertauth          CHAR(1) NOT NULL,
    deleteauth          CHAR(1) NOT NULL,
    updateauth          CHAR(1) NOT NULL,
    updatecols          CHAR(1) NOT NULL,
    alterauth           CHAR(1) NOT NULL,
    referenceauth       CHAR(1) NOT NULL,
    PRIMARY KEY         ( stable_id, grantee, grantor ),
    FOREIGN KEY         ( stable_id )
                        REFERENCES SYS.SYSTABLE ( table_id ),
    FOREIGN KEY         future ( ttable_id )
                        REFERENCES SYS.SYSTABLE ( table_id ),
    FOREIGN KEY         grantee ( grantee ) REFERENCES
                        SYS.SYSUSERPERM ( user_id ),
    FOREIGN KEY         grantor ( grantor )
                        REFERENCES SYS.SYSUSERPERM ( user_id )
)
```

Permissions given by the GRANT command are stored in **SYSTABLEPERM**. Each row in this table corresponds to one table, one user ID granting the permission (*grantor*) and one user ID granted the permission (*grantee*).

There are several types of permission that can be granted. Each permission can have one of the following three values.

N No, the grantee has not been granted this permission by the grantor.

Y Yes, the grantee has been given this permission by the grantor.

G The grantee has been given this permission. In addition, the grantee can grant the same permission to another user.

NOTE: The grantee might have been given permission for the same table by another grantor. If so, this information would be recorded in a different row of **SYSTABLEPERM**.

stable_id The table number of the table or view to which the permissions apply.

grantor The user number of the user ID granting the permission.

grantee The user number of the user ID receiving the permission.

ttable_id In the current version of SQL Anywhere, this table number is always the same as **stable_id**.

selectauth (Y/N/G)
Indicate whether SELECT permission has been granted.

insertauth (Y/N/G)
Indicate whether INSERT permission has been granted.

deleteauth (Y/N/G)
Indicate whether DELETE permission has been granted.

updateauth (Y/N/G)
Indicate whether UPDATE permission has been granted for all columns in the table. (Only UPDATE permission can be given on individual columns. All other permissions are for all columns in a table.)

updatecols (Y/N)
Indicates whether UPDATE permission has only been granted for

some of the columns in the table. If **updatecols** has the value Y, there will be one or more rows in **SYSCOLPERM** granting update permission for the columns in this table.

alterauth (Y/N/G)
Indicate whether ALTER permission has been granted.

referenceauth (Y/N/G)
Indicate whether REFERENCE permission has been granted.

46.25 SYSTRIGGER system table

```
CREATE TABLE SYS.SYSTRIGGER (
    trigger_id          SMALLINT NOT NULL,
    table_id            SMALLINT NOT NULL,
    event               CHAR(1) NOT NULL,
    trigger_time        CHAR(1) NOT NULL,
    trigger_order       SMALLINT,
    foreign_table_id    SMALLINT,
    foreign_key_id      SMALLINT,
    referential_action  CHAR(1),
    trigger_name        CHAR(128),
    trigger_defn        LONG VARCHAR NOT NULL,
    remarks             LONG VARCHAR,
    PRIMARY KEY         ( trigger_id ),
    UNIQUE              ( trigger_name ),
    UNIQUE              ( table_id, event,
                        trigger_time, trigger_order ),
    UNIQUE              ( table_id, foreign_table_id,
                        foreign_key_id, event ),
    FOREIGN KEY         REFERENCES SYS.SYSTABLE,
    FOREIGN KEY         REFERENCES SYS.SYSFOREIGNKEY
)
```

Each trigger in the database is described by one row in **SYSTRIGGER**. The table also contains triggers automatically created by the database for foreign key definitions which have a referential triggered action (such as ON DELETE CASCADE).

trigger_id Each trigger is assigned a unique number (the **trigger number**) that is the primary key for **SYSTRIGGER**.

table_id The table number uniquely identifies the table to which this trigger belongs.

event The event that will cause the trigger to fire. This single character value corresponds to the trigger event that was specified when the trigger was created.

D	DELETE
I	INSERT
U	UPDATE
C	UPDATE OF column-list

trigger_time The time at which the trigger will fire. This single character value corresponds to the trigger time that was specified when the trigger was created.

A	AFTER
B	BEFORE

trigger_order
The order in which the trigger will fire. This determines the order that triggers are fired when there are triggers of the same type (insert, update, or delete) that fire at the same time (before or after).

foreign_table_id
The foreign table number identifies the table containing a foreign key definition which has a referential triggered action (such as ON DELETE CASCADE).

foreign_key_id
The foreign key number identifies the foreign key for the table referenced by **foreign_table_id**.

referential_action
The action defined by a foreign key. This single character value corresponds to the action that was specified when the foreign key was created.

C	CASCADE
D	SET DEFAULT
N	SET NULL
R	RESTRICT

trigger_name The name of the trigger. One table cannot have two triggers with the same name.

trigger_defn The command used to create the trigger.

remarks A comment string.

46.26 SYSUSERMESSAGES system table

```
CREATE TABLE SYS.SYSUSERMESSAGES (
    error           INT             NOT NULL,
    uid             SMALLINT        NOT NULL,
    description     VARCHAR(255)    NOT NULL,
    langid          SMALLINT        NOT NULL,
    UNIQUE          ( error,langid )
    )
```

Each row holds a user-defined message for an error condition.

error A unique identifying number for the error condition.

uid The user ID defining the message.

description The message corresponding to the error condition.

langid Reserved.

46.27 SYSUSERPERM system table

```
CREATE TABLE SYS.SYSUSERPERM (
    user_id             SMALLINT NOT NULL,
    user_name           CHAR(128) NOT NULL UNIQUE,
    password            CHAR(128),
    resourceauth        CHAR(1) NOT NULL,
    dbaauth             CHAR(1) NOT NULL,
    scheduleauth        CHAR(1) NOT NULL,
    user_group          CHAR(1) NOT NULL,
    remarks             LONG VARCHAR,
    PRIMARY KEY         ( user_id )
)
```

Each row of **SYSUSERPERM** describes one user ID.

user_id Each new user ID is assigned a unique number (the **user number**) that is the primary key for **SYSUSERPERM**.

user_name A string containing the name for the user ID. Each userid must have a unique name.

password The password for the user ID. The password contains the NULL value for the special userids **SYS** and **PUBLIC**, preventing anyone from connecting to these user IDs.

resourceauth (Y/N)
Indicate whether the user has RESOURCE authority. Resource authority is required to create tables.

dbaauth (Y/N)
Indicate whether the user has DBA (database administrator) authority. DBA authority is very powerful, and should be restricted to as few user IDs as possible for security purposes.

scheduleauth (Y/N)
Indicate whether the user has SCHEDULE authority. This is currently not used by SQL Anywhere.

user_group (Y/N)
Indicate whether the user is a group.

remarks A comment string.

When a database is initialized, three user IDs are created:

- **SYS** The creator of all the system tables.
- **PUBLIC** A special user ID used to record PUBLIC permissions.
- **DBA** The database administrator user ID is the only usable user ID in an initialized system. The initial password is SQL.

There is no way to connect to the **SYS** or **PUBLIC** user IDs.

46.28 SYSUSERTYPE system table

```
CREATE TABLE SYS.SYSUSERTYPE (
    type_id         SMALLINT    NOT NULL,
    creator         SMALLINT    NOT NULL,
    domain_id       SMALLINT    NOT NULL,
    nulls           CHAR(1)     NOT NULL,
    width           SMALLINT    NOT NULL,
    scale           SMALLINT    NOT NULL,
    type_name       CHAR(128)   NOT NULL,
    "default"       LONG VARCHAR    NULL,
    "check"         LONG VARCHAR    NULL,
    format_str      CHAR(128),
    UNIQUE          ( type_name ),
    PRIMARY KEY     ( type_id ),
    FOREIGN KEY     ( creator )
                    REFERENCES SYS.SYSUSERPERM ( user_id ),
    FOREIGN KEY     REFERENCES SYS.SYSDOMAIN
    )
```

Each row holds a description of a user-defined data type.

type_id	A unique identifying number for the user-defined data type.
creator	The owner of the data type.
domain_id	Identifies the data type for the column by the data type number listed in the **SYSDOMAIN** table.
nulls	A `Y` indicates that the user-defined data type does allow nulls. A `N` indicates that the data type does not allow nulls.
width	This column contains the length of string columns, the precision of numeric columns, and the number of bytes of storage for all other data types.
scale	The number of digits after the decimal point for numeric data type columns, and zero for all other data types.
type_name	The name for the data type, which must be a valid identifier.
"default"	The default value for the data type.
"check"	The CHECK condition for the data type.
format_str	Currently unused.

Chapter 47

SQL Anywhere System Views

About this chapter

This chapter lists predefined views for the SQL Anywhere system tables.

The system tables described in "SQL Anywhere System Tables" on page 1131 use numbers to identify tables, userids, and so forth. While this is efficient for internal use by SQL Anywhere, it makes these tables difficult for people to interpret. A number of predefined system views are provided that present the information in the system tables in a more readable format.

The definitions for the system views are included with their descriptions. Some of these definitions are complicated, but need not be understood to use the views. They serve as good examples of what can be accomplished using the SELECT command and views.

47.1 SYS.SYSCATALOG

```
CREATE VIEW SYS.SYSCATALOG ( creator, tname, dbspacename,
          tabletype, ncols, primary_key, "check", remarks )
AS SELECT ( SELECT user_name FROM SYS.SYSUSERPERM
            WHERE user_id = SYSTABLE.creator ),
          table_name,
          ( SELECT dbspace_name from SYS.SYSFILE
            WHERE file_id = SYSTABLE.file_id ),
          IF table_type='BASE' THEN 'TABLE'
          ELSE table_type ENDIF,
          ( SELECT count(*) FROM SYS.SYSCOLUMN
            WHERE table_id = SYSTABLE.table_id ),
          IF primary_root = 0 THEN 'N' ELSE 'Y' ENDIF,
          IF table_type <> VIEW'
            THEN view_def ENDIF,
          remarks
FROM SYS.SYSTABLE
```

Lists all the tables and views from **SYSTABLE** in a readable format.

47.2 SYS.SYSCOLAUTH

```
CREATE VIEW SYS.SYSCOLAUTH ( grantor, grantee, creator,
          tname, colname )
AS SELECT ( SELECT user_name FROM SYS.SYSUSERPERM
            WHERE user_id = SYSCOLPERM.grantor ),
          ( SELECT user_name FROM SYS.SYSUSERPERM
            WHERE user_id = SYSCOLPERM.grantee ),
          ( SELECT user_name
            FROM SYS.SYSUSERPERM == SYS.SYSTABLE
            WHERE table_id = SYSCOLPERM.table_id ),
          ( SELECT table_name FROM SYS.SYSTABLE
            WHERE table_id = SYSCOLPERM.table_id ),
          ( SELECT column_name FROM SYS.SYSCOLUMN
            WHERE table_id = SYSCOLPERM.table_id
            AND column_id = SYSCOLPERM.column_id )
FROM SYS.SYSCOLPERM
```

Presents column update permission information in **SYSCOLPERM** in a more readable format.

47.3 SYS.SYSCOLUMNS

```
CREATE VIEW SYS.SYSCOLUMNS ( creator, cname, tname,
          coltype, nulls, length, syslength,
          in_primary_key, "colno", default_value, remarks )
AS SELECT ( SELECT user_name FROM SYS.SYSUSERPERM
            WHERE user_id = SYSTABLE.creator ),
          column_name, table_name,
          ( SELECT domain_name FROM SYS.SYSDOMAIN
            WHERE domain_id = SYSCOLUMN.domain_id ),
          nulls, width, scale, pkey, column_id,
          "default", SYSCOLUMN.remarks
FROM SYS.SYSCOLUMN == SYS.SYSTABLE
```

Presents a readable version of the table **SYSCOLUMN**. (Note the S at the end of the view name that distinguishes it from the **SYSCOLUMN** table.)

47.4 SYS.SYSFOREIGNKEYS

```
CREATE VIEW SYS.SYSFOREIGNKEYS ( foreign_creator,
          foreign_tname, primary_creator,
          primary_tname, role, columns )
AS SELECT ( SELECT user_name FROM
            SYS.SYSUSERPERM == SYS.SYSTABLE
            WHERE table_id = foreign_table_id ),
          ( SELECT table_name FROM SYS.SYSTABLE
            WHERE table_id = foreign_table_id ),
          ( SELECT user_name
            FROM SYS.SYSUSERPERM == SYS.SYSTABLE
            WHERE table_id = primary_table_id ),
          ( SELECT table_name FROM SYS.SYSTABLE
            WHERE table_id = primary_table_id ), role,
          ( SELECT list( string( FK.column_name,
                                 ' IS ', PK.column_name ) )
            FROM SYS.SYSFKCOL KEY JOIN
              SYS.SYSCOLUMN FK, SYS.SYSCOLUMN PK
            WHERE foreign_table_id =
              SYSFOREIGNKEY.foreign_table_id
            AND foreign_key_id = SYSFOREIGNKEY.foreign_key_id
            AND PK.table_id = SYSFOREIGNKEY.primary_table_id
            AND PK.column_id = SYSFKCOL.primary_column_id )
FROM SYS.SYSFOREIGNKEY
```

Presents foreign key information from **SYSFOREIGNKEY** and **SYSFKCOL** in a more readable format.

47.5 SYS.SYSGROUPS

```
CREATE VIEW SYS.SYSGROUPS ( group_name, member_name )
AS SELECT g.user_name, u.user_name
FROM SYS.SYSGROUP, SYS.SYSUSERPERM g, SYS.SYSUSERPERM u
WHERE group_id = g.user_id AND group_member = u.user_id
```

Presents group information from SYSGROUP in a more readable format.

47.6 SYS.SYSINDEXES

```
CREATE VIEW SYS.SYSINDEXES ( icreator, iname, fname, creator,
          tname, indextype, colnames, interval, level )
AS SELECT ( SELECT user_name FROM SYS.SYSUSERPERM
            WHERE user_id = SYSINDEX.creator ),
          index_name,
          ( SELECT file_name FROM SYS.SYSFILE
            WHERE file_id = SYSINDEX.file_id ),
          ( SELECT user_name FROM SYS.SYSUSERPERM
            WHERE user_id = SYSINDEX.creator ),
          table_name,
          IF "unique" = 'Y' THEN 'Unique'
                            ELSE 'Non-unique' ENDIF,
          ( SELECT list( string( column_name,
                  IF "order" = 'A' THEN ' ASC' i
                                   ELSE ' DESC' ENDIF ) )
          FROM SYS.SYSIXCOL == SYS.SYSCOLUMN
          WHERE index_id = SYSINDEX.index_id ), 0, 0
FROM SYS.SYSTABLE KEY JOIN SYS.SYSINDEX
```

Presents index information from **SYSINDEX** and **SYSIXCOL** in a more readable format.

47.7 SYS.SYSOPTIONS

```
CREATE VIEW SYS.SYSOPTIONS ( user_name, "option", "setting" )
AS SELECT ( SELECT user_name FROM SYS.SYSUSERPERM
            WHERE user_id = SYSOPTION.user_id ),
            "option", "setting"
FROM SYS.SYSOPTION
```

Displays option settings contained in the table **SYSOPTION** in a more readable format.

47.8 SYS.SYSPROCPARMS

```
CREATE VIEW SYS.SYSPROCPARMS ( creator, parmname, procname,
          parmtype, parmmode, parmdomain, length, remarks )
AS SELECT ( SELECT user_name FROM SYS.SYSUSERPERM
            WHERE user_id = SYSPROCEDURE.creator ),
          parm_name, proc_name, parm_type,
          IF parm_mode_in = 'Y' AND
             parm_mode_out = 'N' THEN 'IN'
          ELSE IF parm_mode_in = 'N'
            AND parm_mode_out = 'Y' THEN 'OUT'
          ELSE 'INOUT' ENDIF ENDIF,
          ( SELECT domain_name FROM SYS.SYSDOMAIN
            WHERE domain_id = SYSPROCPARM.domain_id ),
          width, SYSPROCPARM.remarks
FROM SYS.SYSPROCPARM == SYS.SYSPROCEDURE
```

Lists all the procedure parameters from **SYSPROCPARM** in a readable format.

47.9 SYS.SYSTABAUTH

```
CREATE VIEW SYS.SYSTABAUTH ( grantor, grantee,
          screator, stname, tcreator, ttname,
          selectauth, insertauth, deleteauth,
          updateauth, updatecols, alterauth, referenceauth )
AS SELECT ( SELECT user_name FROM SYS.SYSUSERPERM
            WHERE user_id = SYSTABLEPERM.grantor ),
          ( SELECT user_name FROM SYS.SYSUSERPERM
            WHERE user_id = SYSTABLEPERM.grantee ),
          ( SELECT user_name
            FROM SYS.SYSUSERPERM == SYS.SYSTABLE
            WHERE table_id = SYSTABLEPERM.stable_id ),
          ( SELECT table_name FROM SYS.SYSTABLE
            WHERE table_id = SYSTABLEPERM.stable_id ),
          ( SELECT user_name FROM
            SYS.SYSUSERPERM == SYS.SYSTABLE
            WHERE table_id = SYSTABLEPERM.ttable_id ),
          ( SELECT table_name FROM SYS.SYSTABLE
            WHERE table_id = SYSTABLEPERM.ttable_id ),
          selectauth, insertauth, deleteauth,
          updateauth, updatecols,
          alterauth, referenceauthauth
FROM SYS.SYSTABLEPERM
```

Presents table permission information in **SYSTABLEPERM** in a more readable format.

47.10 SYS.SYSTRIGGERS

```
CREATE VIEW SYS.SYSTRIGGERS ( owner, trigname, tname,
          event, trigtime, trigdefn )
AS SELECT ( SELECT user_name FROM SYS.SYSUSERPERM
            WHERE user_id = SYSTABLE.creator ),
          trigger_name, table_name,
          IF event = 'I' THEN 'INSERT'
          ELSE IF event = 'U' THEN 'UPDATE'
          ELSE IF event = 'C' THEN 'UPDATE'
          ELSE 'DELETE' ENDIF ENDIF ENDIF,
          IF trigger_time = 'B' THEN 'BEFORE'
          ELSE 'AFTER' ENDIF,
          trigger_defn
FROM SYS.SYSTRIGGER == SYS.SYSTABLE
WHERE foreign_table_id IS NULL
```

Lists all the triggers from **SYSTRIGGER** in a readable format.

47.11 SYS.SYSUSERAUTH

```
CREATE VIEW SYS.SYSUSERAUTH ( name, password, resourceauth,
          dbaauth, scheduleauth, user_group )
AS SELECT user_name, password, resourceauth,
          dbaauth, scheduleauth, user_group
FROM SYS.SYSUSERPERM
```

Displays all the information in the table **SYSUSERPERM** except for user numbers. Since this view shows passwords, this system view does not have **PUBLIC** select permission. (All other system views have **PUBLIC** select permission.)

47.12 SYS.SYSUSERLIST

```
CREATE VIEW SYS.SYSUSERLIST ( name, resourceauth,
          dbaauth, scheduleauth, user_group )
AS SELECT user_name, resourceauth,
          dbaauth, scheduleauth, user_group
FROM SYS.SYSUSERPERM
```

Presents all information in **SYSUSERAUTH** except for passwords.

47.13 SYS.SYSUSEROPTIONS

```
CREATE VIEW SYS.SYSUSEROPTIONS ( "user_name",
          "option", "setting" )
as SELECT u.name, "option",
          isnull( ( SELECT "setting" FROM sys.sysoptions s
                    WHERE s.user_name = u.name
                    AND s."option" = o."option" ),
          "setting" )
FROM SYS.SYSOPTIONS o, SYS.SYSUSERAUTH u
WHERE o.user_name = 'PUBLIC'
```

Display effective permanent option settings for each user. If a user has no setting for an option, this view will display the public setting for the option.

47.14 SYS.SYSUSERPERMS

```
CREATE VIEW SYS.SYSUSERPERMS
AS SELECT user_id, user_name, resourceauth, dbaauth,
          scheduleauth, user_group, remarks
FROM SYS.SYSUSERPERM
```

Contains exactly the same information as the table SYS.SYSUSERPERM except the password is omitted. All users have read access to this view, but only the DBA has access to the underlying table (SYS.SYSUSERPERM).

47.15 SYS.SYSVIEWS

```
CREATE VIEW SYS.SYSVIEWS ( vcreator, viewname, viewtext )
AS SELECT user_name, table_name, view_def
FROM SYS.SYSTABLE KEY JOIN SYS.SYSUSERPERM
WHERE table_type = 'VIEW'
```

Lists views along with their definitions.

Chapter 48

Glossary

About the glossary

This glossary contains words used in the SQL Anywhere User's Guide and industry-wide terms concerning SQL databases and related technologies.

address In SQL Remote replication, the destination of replication messages sent by a given message system. Publishers and remote users each have their own address. For a FILE message type, the address is a subdirectory name relative to the directory defined by the SQLREMOTE environment variable. This is typically on a volume that is shared between the replicating databases. SQL Remote stores its messages as replication files in this subdirectory. For a MAPI message type, the address is any valid MAPI address.

article In SQL Remote replication, a database object that represents a whole table, or a subset of the rows and columns in a table. Articles are grouped together in publications.

authority Determines what structural actions a user can perform in a database. While most users will have no special authorities, a user with DBA authority can grant other users resource authority, DBA authority, or remote DBA authority.

autocommit An ISQL option. If autocommit is set to TRUE, then a database COMMIT is performed after each successful command and a ROLLBACK after each failed command. The ODBC interface has a setting similar to autocommit. Users of applications communicating with SQL Anywhere through ODBC should check their application documentation for how to set this option. Developers programming directly to the ODBC interface should consult the ODBC documentation for information about implementing autocommit.

backup It is important to make regular backups of your database files in case of media failure. A backup is a copy of the database file. You can make backups using the SQL Anywhere DBBACKUP utility or using other archiving software of your choice.

base data type One of the intrinsic (simple) data types included in SQL Anywhere (such as INTEGER). User-defined data types (including those supplied with SQL Anywhere) are built on base data types.

base table The tables that permanently hold the data in the database are sometimes called **base tables** to distinguish them from temporary tables and from views.

batch A batch is a set of SQL statements sent together to the database engine by an application.

buffer An area of memory reserved for some dedicated use, such as storing incoming messages for processing.

business rules The rules to which the data in a database must conform are often called business rules, as they reflect organizations' operating practices.

For example, a business rule limiting customers' credit can be enforced by a trigger in a sales order table. A rule requiring library books to be returned within two weeks can be checked by constraints, so that reminders can be issued or fines calculated.

cache To avoid having to access a hard disk every time it needs to retrieve or write information to the database, SQL Anywhere keeps data it may need to access again in the computer's memory, where access is much quicker. The area of memory set aside for this information is called a cache. See also checkpoint log.

checkpoint A time at which all dirty pages held in cache are written to disk is called a checkpoint. Once all the dirty pages are written to disk, the checkpoint log is deleted. A checkpoint may occur for one of several reasons, at times specified by the user or decided internally by the database engine.

checkpoint log A SQL Anywhere database file is composed of pages. Before a page is updated (made dirty), a copy of the original is always made. The copied pages are the checkpoint log. The checkpoint log is deleted when a checkpoint occurs.

client Client is a widely-used term with several meanings. It refers to the user's side of a client/server arrangement: for example, an application that addresses a database, typically held elsewhere on a network, is called a client application.

code page A code page is a character set. SQL Anywhere supports many different national languages and character sets, as long as they are available under the user's operating system. See also: collation.

collation SQL Anywhere supports many different national languages by providing a choice of collations. Each collation specifies a code page (character set) and a sorting and comparison order for that code page. SQL Anywhere also supports custom collations. See also: code page.

collation sequence A collation sequence is an ordering of characters used for sorting and comparing strings. Each national language employs a character set or code page, and a collation sequence for that set.

column All data in relational databases such as SQL Anywhere is held in tables, composed of rows and columns. Each column holds a particular type of information. See also: table, row.

command file A text file containing SQL statements. Command files can be built by yourself (manually) or by database utilities (automatically). The DBUNLOAD utility, for example, creates a command file consisting of the SQL statements necessary to recreate a given database.

command window The ISQL command window is an edit control for entering SQL statements for execution.

commit When a user sends the SQL command COMMIT, all the work done to that point is applied to the database itself, and can no longer be undone. A commit marks the end of a transaction. See also: transaction, rollback.

compound statement A compound statement is a set of SQL statements treated as a unit. The body of a procedure or trigger consists of a compound statement. A compound statement starts with BEGIN and finishes with END. See also: stored procedure, trigger.

compressed database file A database file that has been compressed to a smaller physical size using SQL Anywhere's database compression utility (DBSHRINK). To make changes to a compressed database file, you must use an associated write file. Compressed database files can be re-expanded into normal database files using SQL Anywhere's database uncompression utility (DBEXPAND).

concurrency Multiuser versions of SQL Anywhere support concurrent applications: separate connections which may address the same data in the database, running at the same time. SQL Anywhere provides transaction processing and automatic row-level locking to ensure that information remains consistent and that each concurrent application sees a consistent set of data. See also: transaction, locking.

conflict trigger In SQL Remote replication, a trigger that is fired when an update conflict is detected, before the update is applied. Specifically, conflict triggers are fired by the failure of values in the VERIFY clause of an UPDATE statement to match the values in the database before the update. They are fired before each row is updated.

connection When a client application connects to a database, it specifies several parameters that govern all aspects of the connection once it is established. A user ID, a password, the name of the database to attach to, are all parameters that specify the connection. All exchange of information between the client application and the database to which it is connected is governed by the connection. See also: user ID, password, named connection.

connection parameters When a client application connects to a database, the connection parameters specify the characteristics of the connection. See connection.

connection string When a client application connects to a database, it uses connection parameters to specify the characteristics of the connection. These connection parameters are collected together as a connection string.

See also: connection parameters.

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

constraint When tables and columns are created they may have constraints assigned to them. A constraint ensures that all entries in the database object to which it applies satisfy a particular condition. For example, a column may have a UNIQUE constraint, which requires that all values in the column be different. A table may

have a foreign key constraint, which specifies how the information in the table relates to that in some other table.

See also: integrity, foreign key constraint, primary key constraint, column, table. unique constraint, check constraint.

container In a graphical user interface, a container is an object that contains other objects. Containers can be expanded by double-clicking them.

cursor A cursor is a handle, or identifier, for a particular SQL query and for the position within the result set that is being accessed. Cursors allow each row of a query that returns more than one row to be processed by a client application individually.

cursor stability Cursor stability is a concurrency condition, guaranteed by choosing an isolation level of 1, 2, or 3. Cursor stability guarantees that no row fetched through a cursor yields uncommitted data.

daemon A background process on a computer, running periodically or all the time, which manages a particular function such as printing services or network communication services.

data dictionary See system tables.

data source Databases available to ODBC applications are defined through data sources.

For a detailed description of ODBC data sources, see "Working with ODBC Data Sources" on page 136.

data type Each of *column* in a table is associated with a particular data type. Integers, character strings, and dates are examples of data types. For information about the supported data type, see "Data types" on page 755.

database A relational database is a collection of tables, related by primary and foreign keys. The tables hold the information in the database, and the tables and keys together define the structure of the database. A database may be stored in one or more database files, on one or more devices.

database administrator The database administrator (DBA) is a person responsible for maintaining the database. The DBA is generally

responsible for all changes to a database schema, and for managing users and user groups.

The role of database administrator is built in to SQL Anywhere databases as a user ID. When a database is initialized, a DBA user ID is created. The DBA user ID has authority to carry out any activity within the database.

database connection All exchange of information between client applications and the database takes place in a particular connection. A valid user ID and password are required to establish a connection, and the actions that can be carried out during the connection are defined by the privileges granted to the user ID.

database engine All access to information in a SQL Anywhere database goes through a SQL Anywhere engine. The specific SQL Anywhere engine you are using will depend on your operating system. Requests for information from a database are sent to the database engine, which carries out the instructions.

database file A database is held in one or more distinct database files. The user does not have to be concerned with the organization of a database into files: requests are issued to the database engine about a database, and the engine knows in which file to look for each piece of required information. Database administrators can create new database files for a database using the "CREATE DBSPACE Statement" on page 845 command.

Each table, together with its associated indexes, must be contained in a single database file.

database name When a database is loaded by an engine, it is assigned a database name. Client applications can connect to a database by specifying its database name.

The default database name is the root of the database file.

database object A database is made up of tables, indexes, views, procedures, and triggers. Each of these is a database object.

database owner The user ID that creates a database is the owner of that database, and has the authority to carry out any changes to that database. The database owner is also referred to as the database administrator, or DBA. A database owner can grant permission to other users to have access to the database and to carry out different operations on the database, such as creating tables or stored procedures.

datagram Communications across a network may take place in a session, (also called a connection, or virtual circuit) or in a connectionless manner. In the connectionless case, the independent packets of information are called datagrams, in analogy with telegrams.

Connectionless communications require routing decisions for all packets, and are not guaranteed. Multi-user editions of SQL Anywhere database server use datagrams for their TCP/IP, IPX, and NetDG communication links.

DBA An abbreviation for database administrator, also called the database owner. When a database is first created, using the DBINIT utility, it is created with the single user ID **DBA**, with password **SQL**.

dbspace A SQL Anywhere database can be held in multiple files, called dbspaces. The SQL command CREATE DBSPACE adds a new file to the database.

Each table, together with its associated indexes, must be contained in a single database file. See also: database file, "CREATE DBSPACE Statement" on page 845.

DDE Dynamic data exchange (DDE) is a method for Windows applications to communicate with each other. In any DDE conversation, one application is the client, or destination, while the other application is the server, or source.

default value Also known as a "column default" or just "default", this is a value that is automatically assigned to particular columns when a new row is entered into a database table, without any action on the part of the client application, as long as no value is specified by the client application. If the client application does specify a value for the column, it overrides the column's default value. Default values can be user-defined (a string or number) or pre-defined (e.g. a timestamp supplied by the system).

device A device is a disk drive, a tape drive, or other information storage medium.

DLL A dynamic link library, or DLL, is a collection of compiled functions that can be addressed by a running application at run time. DLL's allow a single set of functions to be shared by many applications, and can be updated without updating every application that depends on them. The Windows, Windows NT, and OS/2 operating systems support DLLs.

driver A piece of software that manages low-level functions on a computer, such as a communication with a network card or a printer.

Embedded SQL The native programming interface to SQL Anywhere from C programs. SQL Anywhere embedded SQL is an implementation of the ANSI and IBM standard.

entity integrity Each row in every table in a database must be uniquely identifiable in order to be accessed by the database engine. Each row is identified by its primary key. This requirement of identifiability is called entity integrity, and is automatically ensured by SQL Anywhere through a checking of the primary key. See also: primary key, referential integrity.

engine name When a database engine is started it is assigned an engine name. Client applications specify the engine to which they want to connect by using the engine name.

The default engine name is the first database name.

environment variable The DOS and OS/2 operating systems allow users to set environment variables that can be used by applications to identify system-specific information. Windows applications have access to DOS environment variables. SQL Anywhere uses the SQLCONNECT and SQLANY environment variables to identify default connection parameters and home directory for SQL Anywhere.

entity In Entity-Relationship design of databases, a first step is to identify the entities in the information you will incorporate into your database. Entities become tables in the final implementation.

Entity-Relationship design Entity-Relationship design is a systematic approach to designing databases, based on a top-down analysis of the tasks you need to perform.

See also: the chapter "Designing Your Database" on page 143.

erase Erasing a database deletes all tables and data from disk, including the transaction log that records alterations to the database.

ethernet Ethernet is a term associated with network cards, a protocol, and a network topology. In a network topology context, an ethernet is commonly associated with a bus network.

exception handler In procedures and triggers, an exception handler is defined in the EXCEPTION part of a compound statement, and is code that is executed if an error is encountered in the compound statement.

extraction In SQL Remote replication, the act of synchronizing a remote database with its consolidated database by unloading the appropriate structure and data from the consolidated database, then reloading it into the remote database. Extraction uses direct manipulation of ordinary files—it does not use the SQL Remote message system.

FILE In SQL Remote replication, a message system that uses shared files for exchanging replication messages. This is useful for testing and for installations without an explicit message-transport system (such as MAPI).

foreign key Tables are related to each other by using foreign keys. A foreign key in one table (the foreign table) contains a value corresponding to the primary key of another table (the primary table). This relates the information in the foreign table to that in the primary table.

See also: primary key, referential integrity.

forward log See transaction log.

full backup In a full backup, a copy is made of the entire database file itself, and optionally of the transaction log. A full backup contains all the information in the database. See also: incremental backup.

function Also called a "user-defined function", this is a type of procedure that returns a single value to the calling environment. A function can be used, subject to permissions, in any place that a built-in non-aggregate function is used.

group A user group is a database user ID that has been given the permission to have members. User groups are used to make the assignment of database permissions simpler. Rather than assign permissions to each user ID, a user ID is assigned to a particular group, and takes on the permissions assigned to that group. See also: permissions, DBA, user ID.

identifier An identifier is any string composed of the characters A through Z, a through z, 0 through 9, underscore (_), at sign (@), number sign (#), or dollar sign ($). The first character must be a letter.

Alternatively, any string of characters can be used as an identifier by enclosing it in double quotes.

incremental backup An incremental backup is a copy of the transaction log. This log contains all the information needed to restore the database to its present state from its state when the transaction log was started.

index An index on one or more columns of a database table allows fast lookup of the information in these columns, and so can greatly speed up database queries. Specifically, indexes assist WHERE clauses in SELECT statements.

inner join An inner join is one kind of JOIN. operation allowed in the FROM clause of SELECT queries. INNER JOIN is the default type of join.

An inner join includes only those rows of the table on each side of the expression that has matching rows in the other table.

For a full description, see "FROM Clause" on page 915. See also: outer join.

integrity Integrity of information in a database ensures that each row in the database can be uniquely identified (entity integrity) and that relations between rows in different tables are properly maintained (referential integrity). SQL Anywhere provides several tools for maintaining the integrity of information in databases. See also: entity integrity, referential integrity.

IPX IPX is a network-level protocol by Novell.

isolation levels The isolation level for instructions within a transaction determines the extent to which other transactions may share the data addressed by the transaction. The isolation level can be set by the user: different isolation levels are appropriate for different kinds of transaction. See also: locking, transaction.

ISQL ISQL (Interactive SQL) is a SQL Anywhere database administration and browsing utility.

join The JOIN clause in a SELECT query enables a single database query to obtain information from several related tables.

keys Database tables may contain two types of key: a *primary key*, and a *foreign key*.

LAN See local area network.

local area network A local area network (LAN) is a collection of networked computers characterized by two attributes:

- A diameter of not more than a few kilometres.
- Ownership by a single organization.

See also: protocol.

locking SQL Anywhere places a lock on a row that is being addressed by a transaction. The lock prevents other transactions from having access to the row in a way that could make the data in the database or the data seen by users of client applications inconsistent, while allowing concurrent transactions. See also: concurrency, transaction.

log files SQL Anywhere maintains a set of three log files to ensure that the data in the database is recoverable in the event of a system or media failure, and to assist database performance. See also: checkpoint log, transaction log, rollback log.

MAPI Microsoft's Message Application Programming Interface, a message system used in several popular e-mail systems such as Microsoft Mail.

media failure A media failure occurs when the information on a medium (typically a hard disk drive) becomes unusable. A media failure may occur as a result of damage to the file, the file system, or the actual device. SQL Anywhere includes tools for backup and recovery to minimize the effects of media failure. See also: system failure, backup.

Message Agent In SQL Remote replication, a program (DBREMOTE) that sends and receives replication messages on behalf of its database. A consolidated database's message agent typically runs continuously, receiving replication messages continuously and sending replication messages periodically, while a remote database's agent typically runs on demand, receiving and sending all pending messages.

message system In SQL Remote replication, a protocol for exchanging messages between the consolidated database and a remote database. SQL Anywhere includes support for a FILE message system (using shared files) and the MAPI message system. In most cases, a

consolidated database and a remote database(s) will send and receive messages using the same message system.

message type In SQL Remote replication, a database object that specifies how remote users communicate with the publisher of a consolidated database. A consolidated database may have several message types defined for it; this allows different remote users to communicate with it using different message systems. A message type is named after a message system (e.g. MAPI), and includes the publisher address for that message system (e.g. a valid MAPI address).

messages Message based communication between applications or computers does not require a direct connection. Instead, a message sent at one time by an application can be received at another time by another application at a later time.

named connection Any connection from ISQL or and embedded SQL application to a database may optionally be given a name. If a client application has several connections in place simultaneously, the user may switch from one connection to another by using the SET CONNECTION command. See also: connection.

named pipes Named Pipes are an interprocess communication mechanism implemented by a number of leading operating system vendors. Named Pipes are usually implemented atop some transport-level protocol.

NDIS NDIS (Network Driver Interface Specification) is a datalink-level protocol jointly defined by Microsoft and IBM.

NetBIOS NetBIOS is a transport-level interface defined by IBM. See also: protocol.

NetBEUI NetBEUI is a transport-level protocol. See also: protocol.

NetWare A widely-used network operating system by Novell. NetWare generally employs the IPX protocol, although the TCP/IP protocol may also be used.

network adapter A network adapter is the physical attachment to a computer allowing network communication. Also called a network card.

See also: network driver

network architecture A network architecture is the physical interconnection of computers on a network. Typical architectures include bus, where all computers connect to a single carrier, and ring, where one computer is directly connected to another to form a closed ring.

network card A network card is the physical attachment to a computer allowing network communication. Also called a network adapter.

See also: network driver

network driver A network driver is the software logically located between the operating system and the *network card*, which allows applications to communicate to the network card. Network cards are usually bundled with a number of network drivers for various operating systems and network protocols.

network operating system An operating system optimized for server deployment, such as Novell NetWare and Microsoft Windows NT Server Edition.

NULL The NULL value is a special value for a database entry, different from any other valid value for any data type. The NULL value represents missing or inapplicable information. See also: column, constraint.

nullability Determines whether a column allows a NULL value to be assigned to it. Columns are typically allowed to be NULL if their values are optional or not always available, and are not required for the data in the database be correct.

ODBC The Open Database Connectivity (ODBC) interface, defined by Microsoft Corporation, is a standard interface to database management systems in the Windows and Windows NT environments. ODBC is one of several interfaces supported by SQL Anywhere.

ODI ODI (Open Datalink Interface) is a data link-level interface defined by Novell.

optimizer The SQL Anywhere database engine contains an optimizer which attempts to pick the best strategy for executing each query. The optimizer uses educated guesses about the occurrence of particular elements in the database to select a strategy. The user can provide explicit estimates in order to help tune an execution strategy. See also: index, query.

outer join An outer join is one kind of JOIN operation allowed in the FROM clause of SELECT queries.

An outer join may be either a LEFT OUTER or a RIGHT OUTER join. An outer join includes all rows of the table on the LEFT (RIGHT) of the expression whether or not there are matching rows in the other table.

For a full description, see "FROM Clause" on page 915. See also inner join.

owner Each object of a database is owned by the user ID that created it. The owner of a database object has rights to do anything with that object. See also: DBA.

packet A packet is a communication entity between processes. Messages between two processes are usually fragmented into a number of packets for data transmission from the source and then reassembled at the target host into a single message.

page A SQL Anywhere database file is composed of pages. Before a page is updated (made dirty), a copy of the original is always made in memory. The copied pages are the checkpoint log. The page-size of a database file is specified when the database is created. See also: cache, checkpoint log.

passthrough In SQL Remote replication, a mode by which the publisher of the consolidated database can directly change remote databases with SQL statements. Passthrough is set up for specific remote users (you can specify all remote users, individual users, or those users who subscribe to given publications). In normal passthrough mode, all database changes made at the consolidated database are passed through to the selected remote databases. In "passthrough only" mode, the changes are made at the remote databases, but not at the consolidated database.

password Whenever a user connects to a database, a password must be specified. The passwords are stored in the SYS.SYSUSERPERM system table, to which only the DBA has access.

performance statistics Values that reflect the performance of the database system with respect to disk and memory usage. The CURRREAD statistic, for example, represents the number of file reads issued by the engine which have not yet completed.

permissions Each user has a set of permissions that govern the actions they may take while connected to a database. Permissions are assigned by the DBA or by the owner of a particular database object. See also: DBA, database object, owner.

phantom lock A phantom lock is one of three types of lock supported by SQL Anywhere to support concurrency, as part of SQL Anywhere's transaction processing capabilities.

A phantom lock is a type of read lock employed at isolation level 3. No other transaction can read or update the row once a phantom lock has been acquired.

See also: read lock, write lock, locking. For a full description, see the chapter "How locking works" on page 204.

primary copy In a replication installation, the primary copy of any data is considered to hold the original of the data. All other places in which the piece of data is held are copies of this original.

protocol The rules and conventions used to communicate between computers are collectively known as a protocol. Instances of protocols include IPX, NetBEUI, and IP. The SQL Anywhere client and server can communicate using a number of different protocols, including NetBIOS, IPX, NamedPipes and TCP/IP.

See also: IPX, NetBIOS, Named Pipes, TCP/IP.

protocol stack Network communications between communications take place according to a set of protocols. These protocols are logically organized in layers, each with a different function. The set of layers forms a protocol stack.

primary key Each table in a relational database must be assigned a primary key. The primary key is a column, or set of columns, whose values uniquely identify every row in the table. See also: entity integrity.

programming interface Programs may connect to SQL Anywhere using one of the interfaces supported by SQL Anywhere. Embedded SQL is SQL Anywhere's native interface. ODBC is another low-level interface. WSQL DDE and WSQL HLI provide higher level interfaces.

publication In SQL Remote replication, a database object that describes data to be replicated. A publication consists of articles (tables or subsets of tables). Periodically, the changes made to each

publication in a database are replicated to all subscribers to that publication as publication updates.

publication update In SQL Remote replication, a periodic batch of changes made to one or more publications in one database. A publication update is sent as part of a replication message to the remote database(s).

publisher In SQL Remote replication, the single user in a database that can exchange replication messages with other replicating databases.

remote database In SQL Remote replication, a database that exchanges replication messages with a consolidated database. Remote databases may contain all or some of the data in the consolidated database.

remote permission In SQL Remote replication, the permission to exchange replication messages with the publishing database. Granting remote permissions to a SQL Anywhere user make them a remote user. This requires you to specify a message type, an appropriate remote address, and a replication frequency. In general terms, remote permissions can also refer to any user involved in SQL Remote replication (for example, the consolidated publisher and remote publisher).

qualifier The name of a database object, such as a table or a view, may be preceded by a qualifier to indicate its owner and hence uniquely identify the database object. For example, a table named **some_table** created by a user **A_User** can be uniquely identified as **A_User.some_table**.

Columns in tables or views may likewise be qualified by the table name (and owner) to identify them uniquely. For example, a column **this_column** may have the fully qualified name **User.some_table.this_column**.

Use of fully qualified names in SQL queries helps to avoid ambiguities.

query Information is retrieved from SQL Anywhere databases by submitting a query. A query is an SQL SELECT statement, the clauses of which specify the exact information the database engine must return tot he client application. See also: SQL, subquery.

read lock A read lock is one of three types of lock supported by SQL Anywhere to support concurrency, as part of SQL Anywhere's transaction processing capabilities.

When a transaction reads a row, employing one of the more secure isolation levels available (level 2 or 3), other transactions can read the row, but cannot update it.

See also: phantom lock, write lock, locking. For a full description, see the chapter "How locking works" on page 204.

referential integrity The tables of a relational database are related to each other by foreign keys. SQL Anywhere provides tools that maintain the referential integrity of the database: that is, ensure that the relations between the rows in different tables remain valid. See also: foreign key, entity integrity.

relationship A step in the Entity-Relationship design of databases, is to identify the relationships between the entities that you have identified.

One-to-many and one-to-one relationships become foreign key relationships in the final implementation, while a many-to-many relationship becomes a table. See also: entity. For a full description, see the chapter "Designing Your Database" on page 143.

remote DBA authority The Message Agent should be run using a user ID with REMOTE DBA authority, to ensure that actions can be carried out, without creating security loopholes.

remote user In SQL Remote replication, a SQL Anywhere user who has been granted remote permissions in a replication setup. When the remote database is extracted from the consolidated database, the remote user becomes the publisher of the remote database, able to exchange publication updates with the consolidated database. While SQL Anywhere groups can also be granted remote permissions, note that users in these "remote groups" do not inherit remote permissions from their group.

replication For databases, a process by which the changes to data in one database (including creation, updating, and deletion of records) are also applied to the corresponding records in other databases. SQL Anywhere supports replication using SQL Remote or Sybase Replication Server.

replication frequency In SQL Remote replication, a setting for each remote user that determines how often the publisher's message agent should send replication messages to that remote user. The frequency can be specified as on-demand, every given interval, or at a certain time of day.

replication message In SQL Remote replication, a discrete communication that is sent from a publishing database to a subscribing database. Messages can contain a mixture of publication updates and passthrough statements (manual SQL statements such as DDL).

resource authority Resource authority is the permission to create and modify objects of a database schema. Resource authority can be granted only by the DBA. See also: DBA, permissions.

role name Each foreign keyis assigned a name, called a role name, to distinguish it from other foreign keys in the same table. If no role name is specified by the user, the role name is set to the name of the primary table.

See also: "CREATE TABLE Statement" on page 859

rollback When a user sends a rollback SQL statement to the database engine, all work since the last savepoint or since the beginning of the current transaction is undone, and no changes are made to the database by any of the instructions. See also: commit, transaction, rollback log.

rollback log A log kept in order to cancel changes made to database tables. The rollback log is needed in the event of a ROLLBACK request or a system failure. There is a separate rollback log for each transaction. When a transaction is complete, its rollback log is deleted.

row All data in relational databases such as SQL Anywhere is held in tables, composed of rows and columns. Each row holds a separate occurrence of each column. In a table of employee information, for example, each row contains information about a particular employee. See also: table, column.

runtime database engine The SQL Anywhere Desktop Runtime engine is a royalty-free redistributable database engine. The runtime database engine supports all the database manipulation language features of the full SQL Anywhere, with the exception of procedures and triggers. Also, the runtime database engine does not employ a transaction log.

savepoint Within a transaction, a SAVEPOINT statement allows flexibility in committing and rolling back work. Work since the most recent savepoint can be undone using ROLLBACK TO SAVEPOINT. The RELEASE SAVEPOINT disallows any future rollbacks to the most recent savepoint. Before SQL Anywhere version 4.0,

savepoints were called subtransactions. See also: transaction, commit, rollback.

schema The structure of a database is called the schema. The schema is held in the system tables.

The schema includes the complete definitions of each database object in the database, including tables, indexes, views, procedures, and triggers.

search condition In SQL, a search condition is a WHERE clause in a SELECT statement. For a full description of valid search conditions, see "Search conditions" on page 803.

session Communications across a network may take place in a session, (also called a connection, or virtual circuit) or in a connectionless manner, using a datagram method. When a connection is established, a route from the source computer to the destination computer is part of the session setup, and routing decisions are not required for every packet.

Network server editions of SQL Anywhere use sessions for their NetBIOS and local Named Pipes communication links.

SQL Structured Query Language (SQL) is the language used to communicate to SQL Anywhere databases. SQL is very widely used in database applications, and in order to ensure compatibility among databases, SQL is the subject of standards set by several standards bodies.

SQLCA A SQL Connection Area (SQLCA) is a segment of database client application code that manages the connection parameters governing the connections between the client application and a database. See also: programming interface.

SQL Remote An asynchronous message-based replication system for two-way server-to-laptop, server-to-desktop, and server-to-server replication between SQL Anywhere databases.

server See database server.

statement SQL allows several kinds of statement. Some statements modify the data in a database (commands), others request information from the database (queries), and others modify the database schema itself. See also: SQL, query.

store-and-forward Store-and-forward exchange of information is typical of message based systems, and allows information to be exchanged without a direct connection between applications.

stored procedure Stored procedures are procedures kept in the database itself, which can be called from client applications. Stored procedures provide a way of providing uniform access to important functions automatically, as the procedure is held in the database, not in each client application. See also: trigger.

subquery A subquery is a component of a SQL query (SELECT statement) that is itself a query. Subqueries are important tools in constructing complex queries to databases. See also: SQL. query.

submission In a SQL Remote installation, changes made to a remote database and sent to the consolidated database are a submission. If and only if the changes are successfully applied at the consolidated database they will be replicated to other databases.

subscriber In SQL Remote replication, a remote user who is subscribed to one or more of a database's publications.

subscribing In a Replication Server or SQL Remote installation, a database that has subscribed to a replication or publication receives updates of changes to the data in that replication or publication.

subscription In SQL Remote replication, a link between a publication and a remote user, allowing the user to exchange updates on that publication with the consolidated database. The user's subscription may include an argument (value) for the publication's SUBSCRIBE BY paramater (if any).

subtransaction See *savepoint.*

synchronization In SQL Remote replication, the process by which SQL Remote deletes all existing rows from those tables of a remote database that form part of a publication, and copies the publication's entire contents from the consolidated database to the remote database. Synchronization is performed during the initial extraction of the remote database from the consolidated database, and may also be necessary later if a remote database becomes corrupt or gets out of step with the consolidated database (and cannot be repaired using passthrough mode). Synchronization can be accomplished by bulk extraction (the recommended method), by manually loading from

files, or by sending synchronization messages through the message system.

system failure A system failure occurs when a power failure or some other failure causes a computer to fail while there are partially completed transactions. See also: media failure, transaction, backup.

system catalog The system catalog is the list of all tables and views in the database.

The SQL Anywhere system view SYS.SYSCATALOG presents this information.

system object In a database, a table, view, stored procedure, or user-defined data type that is pre-defined by SQL Anywhere. System tables store information about the database itself, while system views, procedures, and user-defined data types largely support Sybase Transact-SQL compatibility.

system tables Every SQL Anywhere database includes a set of tables called the system tables, which hold information about the database structure itself: descriptions of the tables, users and their permissions, and so on.

The system tables are created and maintained automatically by the database engine. They are owned by the special user ID **SYS**, and cannot be modified by database users.

system views Every SQL Anywhere database includes a set of views, which present the information held in the system tables in a more easily understood format.

table All data in relational databases is stored in tables. Each table consists of rows and columns. Each column carries a particular kind of information (a phone number, a name, and so on), while each row specifies a particular entry. Each row in a relational database table must be uniquely identifiable by a primary key. See also: database, row, column, primary key, database object, owner.

TCP/IP Transmission Control Protocol/Internet Protocol (TCP/IP) is a network protocol supported by SQL Anywhere See also: protocol, IPX, NetBIOS.

temporary table Complex queries can require that large amounts of intermediary information be held. SQL Anywhere holds this information in temporary tables. It can build indexes on temporary tables, and sort them, just like permanent tables. See also: query.

token ring A token ring is a specification of a particular network architecture and protocol. A ring architecture is a topology is which every computer on the ring has a wire in and a wire out and forms a closed loop. A token is passed around the ring to arbitrate between computers that wish to communicate at the same time.

topics Each separate panel in an online help system is a topic.

Transact-SQL The SQL dialect used in Sybase SQL Server. SQL Anywhere supports a large subset of Transact-SQL.

transaction A transaction is a logical unit of work that should be processed in its entirety by the database (though not necessarily at once) or not at all. SQL Anywhere supports transaction processing, with locking features built in to allow concurrent transactions to access the database without corrupting the data. Transactions begin following a COMMIT or ROLLBACK statement and end either with a COMMIT statement, which makes all the changes to the database required by the transaction permanent, or a ROLLBACK statement, which undoes all the changes made by the transaction. See also: locking, concurrency.

transaction blocking A transaction becomes blocked when it must wait for another transaction to finish before it can carry out its task. Proper design of transactions by application developers can minimize the occurrence of transaction blocks, which slow down database operation. See also: transaction, locking.

transaction log A log storing all changes made to a database, in the order in which they are made. In the event of a media failure on a database file, the transaction log is essential for database recovery. The transaction log should therefore be kept on a different device from the database files for optimal security.

The SQL Anywhere Desktop Runtime System does not employ a transaction log.

transaction processing A database engine that supports transaction processing is capable of ensuring that the results of each *transaction* are either stored in the database in their entirety or not at all. Transaction

processing is a key element in promoting secure and reliable databases.

translation driver A translation driver is a part of the data link layer of a protocol stack. Data link layers conform to either ODI or NDIS specifications, and translation drivers supplied by networking software vendors enable multiple protocol stacks, requiring both ODI and NDIS drivers, to operate with a single network adapter.

trigger A trigger is a procedure stored in the database that is executed automatically by the database engine whenever a particular action occurs, such as a row being updated. Triggers are used to enforce complex forms of referential integrity, or to log activity on database tables. See also: integrity, stored procedure.

two-phase commit A mechanism to coordinate transactions across multiple database servers; only needed for distributed databases. See also: commit, transaction.

unload Unloading a database dumps the structure and/or data of the database to text files (command files for the structure, ASCII comma-delimited files for the data). This may be useful for creating extractions, creating a backup of your database, or building new copies of your database with the same or slightly modified structure. You can also unload the data (but not the structure) of a particular table.

updates In replication, each set of changes sent from one database to another is an update to a publication or replication. :glosssary.upgrade A major release of SQL Anywhere (formerly Watcom SQL) generally means a revised internal database format (to support new database features). When you upgrade your SQL Anywhere software to a major revision (for example, moving from 4.0 to 5.0), you should upgrade your existing databases to the new database format (after making backups of them).

user-defined data type A named combination of base data type, default value, check condition, and nullability. Defining similar columns using the same user-defined data type encourages consistency throughout the database.

user account Every connection with a database requires a user account. The permissions that a user has are tied to their user account. A user account consists of a user ID and password.

user ID A string of characters that identifies the user when connecting to a particular database. The user ID, together with a password, constitute a user account. See also: permissions, connection, DBA.

validate When the information in a database, or a database table, is checked for integrity it is validated. See also: entity integrity, referential integrity.

view A view is a computed table. Every time a user uses a view of a particular table, or combination of tables, it is recomputed from the information stored in those tables. Views can be useful for security purposes, and to tailor the appearance of database information to make data access straightforward. As a permanent part of the database schema, a view is a database object. See also: table, database object.

VIM Vendor-Independent Messaging, a message system used in cc:Mail and Lotus Notes.

Watcom-SQL The SQL dialect used in SQL Anywhere. Watcom-SQL conforms closely to ANSI SQL.

Winsock Winsock is a specification of the socket library with extensions. The socket library is a collection of functions used to interface to a number of different transport-level protocols. The socket library was initially specified by the University of California at Berkeley for the TCP/IP protocol.

write file If a database is used with a write file, all changes made to the database do not modify the database itself, but instead are made to the write file. Write files are useful in applications development, so the developer can have access to the database without interfering with it. Also, write files are used in conjunction with compressed databases and other read-only databases.

write lock A write lock is one of three types of lock supported by SQL Anywhere to support concurrency, as part of SQL Anywhere's transaction processing capabilities.

When a transaction updates or inserts a row, it acquires a write lock on the row. No other transaction can acquire a lock on the same row.

See also: read lock, phantom lock, locking. For a full description, see the chapter "How locking works" on page 204.

WSQL DDE A high level programming interface to SQL Anywhere for Windows applications.

WSQL HLI A high level programming interface to SQL Anywhere.

@

A

B

C

E

F

G

J

K

L

M

N

O

S

T

U

V

W

X

Y