Powersoft

Open Tools from Sybase, Inc.

PowerBuilder

SQL Anywhere

User's Guide
Volume 1: Using SQL Anywhere

Version 6

PowerBuilder®

AA0521

October 1997

SQL Anywhere for PowerBuilder

This *SQL Anywhere User's Guide* describes all the features of the SQL Anywhere product. The version of SQL Anywhere that is included in PowerBuilder does not have all the features of the full SQL Anywhere product. The following features, documented in the *SQL Anywhere User's Guide* are not available to PowerBuilder users:

- **Embedded SQL, HLI, Open Client, and DDE interfaces** PowerBuilder users access SQL Anywhere through ODBC. The chapters on the SQL Anywhere programming interfaces can be ignored, and the Open Server Gateway to enable Open Client connections can also be ignored.
- **SQL Preprocessor** The SQL preprocessor is required only for Embedded SQL, and is not needed by or provided for PowerBuilder users.
- **SQL Remote** The SQL Remote replication technology is not included with PowerBuilder.
- **Multi-user network server** Only a standalone database engine is included with PowerBuilder. The multi-user network server is not included in PowerBuilder.
- **Operating systems** Only the version of SQL Anywhere for the operating system you are using is included in PowerBuilder. The *SQL Anywhere User's Guide* describes all platforms for which SQL Anywhere is available.

In addition, note that the sample database is referred to throughout by the file name SADEMO.DB. This sample database is the same as the Powersoft demo database (PSDEMODB.DB), and all examples in this book will work against the Powersoft demo database.

Contents

About This Manual v

PART ONE **Introduction to SQL Anywhere**

1 **Introduction** 3
- About SQL Anywhere 4
- Upgrading databases to SQL Anywhere 7
- About this manual 8

2 **New Features in SQL Anywhere 5** 13
- What's in a name? 14
- New features overview 15
- New features in the Watcom-SQL language 18
- New sample database 21

3 **Overview of SQL Anywhere** 23
- The SQL Anywhere engine and the SQL Anywhere server 24
- Running SQL Anywhere on a single computer 25
- Running SQL Anywhere on a network 29
- Running mixed operating systems on a single computer 32
- SQL Anywhere programming interfaces 34
- The SQL Anywhere programs 36

PART TWO **Tutorials**

4 **Managing Databases with Sybase Central** 45
- Sybase Central and database management 46
- Navigating the main Sybase Central window 47
- Adding a table to a database 52
- Viewing and editing procedures 57
- Managing users and groups 60

Backing up a database using Sybase Central 63
Using the Sybase Central online help 65

5 **Using ISQL 67**
The SQL Anywhere program group 68
Starting SQL Anywhere 69
Connecting to the sample database from ISQL 70
Accessing Help from ISQL 71
The ISQL command window 72
Leaving ISQL 73
Displaying data in ISQL 74
Command recall in ISQL 76
Function keys 78
Canceling an ISQL command 79
What's next? 80

6 **Using Character-Mode ISQL 81**
Tutorial files 82
Starting the SQL Anywhere software 83
Connecting to the sample database from ISQL 84
ISQL menu selection 85
Obtaining help from ISQL 86
The ISQL command window 87
Leaving ISQL 88
Displaying data in ISQL 89
Command window keys in ISQL 91
Scrolling the data window 93
Command recall in ISQL 94
Function keys 95
Aborting an ISQL command 96
What next? 97

7 **Selecting Data from Database Tables 99**
Looking at the information in a table 101
Ordering query results 103
Selecting columns from a table 104
Selecting rows from a table 105
Comparing dates in queries 106
Compound search conditions in the WHERE clause 107
Pattern matching in search conditions 108
Matching rows by sound 109
Short cuts for typing search conditions 110

8 **Joining Tables 111**
Displaying a list of tables 112
Joining tables with the cross product 114
Restricting a join 115
How tables are related 117
Join operators 118

9 **Obtaining Aggregate Data 121**
A first look at aggregate functions 122
Using aggregate functions to obtain grouped data 123
Restricting groups 125

10 **Updating the Database 127**
Adding rows to a table 128
Modifying rows in a table 129
Canceling changes 130
Making changes permanent 131
Deleting rows 132
Validity checking 133

11 **Introduction to Views 137**
Defining a view 138
Using views for security 140

12 **Introduction to Subqueries 141**
Preparing to use subqueries 142
A simple subquery 143
Comparisons using subqueries 145
Using subqueries instead of joins 148

13 **Command Files 151**
Entering multiple statements in the ISQL Command window 152
Saving statements as command files 153
Command files with parameters 154

14 **System Tables 155**
The SYSCATALOG table 156
The SYSCOLUMNS table 157
Other system tables 158

PART THREE Using SQL Anywhere

15 **Connecting to a Database 161**
Connection overview 162
Connecting from the SQL Anywhere utilities 168
Connecting from an ODBC-enabled application 169

16 **Designing Your Database 181**
Relational database concepts 182
Planning the database 185
The design process 187
Designing the database table properties 199

17 **Working with Database Objects 203**
Using Sybase Central to work with database objects 204
Using ISQL to work with database objects 205
Working with databases 206
Working with tables 211
Working with views 217
Working with indexes 223

18 **Ensuring Data Integrity 225**
Data integrity overview 226
Using column defaults 230
Using table and column constraints 235
Enforcing entity and referential integrity 239
Integrity rules in the system tables 244

19 **Using Transactions and Locks 245**
An overview of transactions 246
How locking works 250
Isolation levels and consistency 251
How SQL Anywhere handles locking conflicts 254
Choosing an isolation level 256
Savepoints within transactions 258
Particular concurrency issues 259
Transactions and portable computers 262

20 **Using Procedures, Triggers, and Batches 265**
Procedure and trigger overview 266
Benefits of procedures and triggers 267

Introduction to procedures 268
Introduction to user-defined functions 273
Introduction to triggers 276
Introduction to batches 281
Control statements 283
The structure of procedures and triggers 286
Returning results from procedures 290
Using cursors in procedures and triggers 295
Errors and warnings in procedures and triggers 300
Using the EXECUTE IMMEDIATE statement in procedures 307
Transactions and savepoints in procedures and triggers 308
Some hints for writing procedures 309
Statements allowed in batches 312
Calling external libraries from stored procedures 314

21 **Monitoring and Improving Performance 319**
Factors affecting database performance 320
Using keys to improve query performance 322
Using indexes to improve query performance 326
Search strategies for queries from more than one table 328
Sorting query results 331
Temporary tables used in query processing 332
How the optimizer works 333
Monitoring database performance 336

22 **Database Collations 347**
Collation overview 348
Support for multibyte character sets 352
Choosing a character set 354
Creating custom collations 357
The collation file format 358

23 **Importing and Exporting Data 363**
Import and export overview 364
Exporting data from a database 366
Importing data into a database 370
Tuning bulk operations 373

24 **Managing User IDs and Permissions 375**
An overview of database permissions 376
Managing individual user IDs and permissions 380
Managing groups 386
Database object names and prefixes 391

Using views and procedures for extra security 393
How SQL Anywhere assesses user permissions 396
Users and permissions in the system tables 397

25 **Backup and Data Recovery 399**
System and media failures 400
The SQL Anywhere logs 401
Using a transaction log mirror 406
Backing up your database 411
Recovery from system failure 414
Recovery from media failure 416

26 **Introduction to SQL Remote Replication 419**
Introduction to data replication 420
SQL Remote concepts 422
SQL Remote features 427
Message systems supported by SQL Remote 429
Tutorial: setting up SQL Remote using Sybase Central 430
Set up the consolidated database in Sybase Central 433
Set up the remote database in Sybase Central 438
Tutorial: setting up SQL Remote using ISQL and DBXTRACT 439
Set up the consolidated database 442
Set up the remote database 445
Start replicating data 447
A sample publication 450
Some sample SQL Remote setups 451

27 **SQL Remote Administration 455**
SQL Remote administration overview 456
SQL Remote message types 457
Managing SQL Remote permissions 463
Setting up publications 470
Designing publications 475
Setting up subscriptions 483
Synchronizing databases 484
How statements are replicated by SQL Remote 489
Managing a running SQL Remote setup: overview 494
Running the SQL Remote Message Agent 496
The SQL Remote message tracking system 500
Transaction log and backup management for SQL Remote 503
Error reporting and conflict resolution in SQL Remote 507
Using passthrough mode for administration 513

28 **Running Programs as Services** **517**
Introduction to services 518
Managing services 519
Adding and removing SQL Anywhere services 520
Configuring SQL Anywhere services 522
Starting and stopping services 526
Running more than one service 528
Monitoring a SQL Anywhere network server service 531
The Windows NT Control Panel Service Manager 532

PART FOUR **Transact-SQL Compatibility**

29 **Using Transact-SQL with SQL Anywhere** **535**
An overview of SQL Anywhere support for Transact-SQL 536
SQL Server and SQL Anywhere architectures 539
General guidelines for writing portable SQL 543
Configuring SQL Anywhere for Transact-SQL compatibility 544
Using compatible data types 549
Local and global variables 556
Building compatible expressions 561
Using compatible functions 565
Building compatible search conditions 576
Other language elements 580
Transact-SQL statement reference 581
Compatible system catalog information 598
SQL Server system and catalog procedures 601
Implicit data type conversion 603

30 **Transact-SQL Procedure Language** **605**
Transact-SQL procedure language overview 606
Automatic translation of SQL statements 608
Transact-SQL stored procedure overview 610
Transact-SQL trigger overview 611
Transact-SQL batch overview 613
Supported Transact-SQL procedure language statements 614
Returning result sets from Transact-SQL procedures 624
Variable and cursor declarations 625
Error handling in Transact-SQL procedures 627

31 **Using the Open Server Gateway** **629**
Open Server Gateway overview 630
Open Server Gateway architecture 632

Data type mappings 634
Setting up the Open Server Gateway 642
Events handled by Open Server Gateway 646
Using SQL Anywhere with OmniCONNECT 649

PART FIVE The SQL Anywhere Programming Interfaces

32 Programming Interfaces 653

33 The Embedded SQL Interface 655
The C language SQL preprocessor 656
Embedded SQL interface data types 664
Host variables 667
The SQL communication area (SQLCA) 673
Fetching data 677
Static vs dynamic SQL 682
The SQL descriptor area (SQLDA) 690
SQL procedures in Embedded SQL 696
Library functions 701
Interface library DLL dynamic loading 723
Embedded SQL commands 727
Database examples 730
SQLDEF.H header file 747

34 ODBC Programming 751
ODBC C language programming 752
ODBC programming for the Macintosh 762

35 The WSQL DDE Server 765
DDE concepts 766
Using WSQL DDE Server 768
Excel and WSQL DDE Server 772
Word and WSQL DDE Server 774
Visual Basic and WSQL DDE Server 775

36 The WSQL HLI Interface 779
DLL concepts 780
Using WSQL HLI 781
Host variables with WSQL HLI 782
WSQL HLI functions 783
wsqlexec command strings 790

WSQL HLI and Visual Basic 798
WSQL HLI and REXX 802

PART SIX **SQL Anywhere Reference**

37 **SQL Anywhere Components 807**
SQL Anywhere components overview 809
Registry entries and environment variables 810
Software component return codes 813
The database engine 814
The Backup utility 824
The Collation utility 829
The Compression utility 832
The Erase utility 834
The Information utility 837
The Initialization utility 839
The ISQL utility 846
The Log Translation utility 849
The Open Server Gateway 853
The Open Server Information utility 855
The Open Server Stop utility 856
The REBUILD batch or command file 857
The SQL Remote Database Extraction utility 858
The SQL Remote Message Agent 863
The Stop utility 867
The Transaction Log utility 869
The Uncompression utility 873
The Unload utility 876
The Upgrade utility 882
The Validation utility 885
The Write File utility 888
The SQL Preprocessor 892

38 **Watcom-SQL Language Reference 895**
Syntax conventions 896
Watcom-SQL language elements 897
Expressions 899
Search conditions 907
Comments in Watcom-SQL 915

39 **SQL Anywhere Data Types 917**
Character data types 918

Numeric data types 920
Date and time data types 922
Binary data types 926
User-defined data types 927
Data type conversions 929
Year 2000 compliance 930

40 **Watcom-SQL Functions 935**
Aggregate functions 936
Numeric functions 938
String functions 941
Date and time functions 945
Data type conversion functions 951
System functions 953
Miscellaneous functions 965

41 **Watcom-SQL Statements 969**
ALLOCATE DESCRIPTOR statement 970
ALTER DBSPACE statement 972
ALTER PROCEDURE statement 974
ALTER PUBLICATION statement 975
ALTER REMOTE MESSAGE TYPE statement 976
ALTER TABLE statement 977
ALTER TRIGGER statement 982
ALTER VIEW statement 983
CALL statement 984
CASE statement 986
CHECKPOINT statement 988
CLOSE statement 989
COMMENT statement 991
COMMIT statement 993
Compound statements 995
CONFIGURE statement 998
CONNECT statement 999
CREATE DATATYPE statement 1002
CREATE DBSPACE statement 1004
CREATE FUNCTION statement 1005
CREATE INDEX statement 1007
CREATE PROCEDURE statement 1009
CREATE PUBLICATION statement 1013
CREATE REMOTE MESSAGE TYPE statement 1015
CREATE SCHEMA statement 1017
CREATE SUBSCRIPTION statement 1019
CREATE TABLE statement 1020

CREATE TRIGGER statement 1028
CREATE VARIABLE statement 1031
CREATE VIEW statement 1033
DBTOOL statement 1035
Declaration section 1038
DECLARE CURSOR statement 1039
DECLARE TEMPORARY TABLE statement 1043
DEALLOCATE DESCRIPTOR statement 1044
DELETE statement 1045
DELETE (positioned) statement 1047
DESCRIBE statement 1049
DISCONNECT statement 1052
DROP statement 1053
DROP CONNECTION statement 1055
DROP OPTIMIZER STATISTICS statement 1056
DROP PUBLICATION statement 1057
DROP REMOTE MESSAGE TYPE statement 1058
DROP STATEMENT statement 1059
DROP VARIABLE statement 1060
DROP SUBSCRIPTION statement 1061
EXECUTE statement 1062
EXECUTE IMMEDIATE statement 1064
EXIT statement 1065
EXPLAIN statement 1066
FETCH statement 1068
FOR statement 1073
FROM clause 1074
GET DATA statement 1081
GET DESCRIPTOR statement 1083
GET OPTION statement 1084
GRANT statement 1085
GRANT CONSOLIDATE statement 1089
GRANT PUBLISH statement 1091
GRANT REMOTE statement 1092
HELP statement 1094
IF statement 1095
INCLUDE statement 1097
INPUT statement 1098
INSERT statement 1102
LEAVE statement 1104
LOAD TABLE statement 1105
LOOP statement 1108
MESSAGE statement 1109
NULL value 1110
OPEN statement 1112
OUTPUT statement 1115

PARAMETERS statement 1118
PASSTHROUGH statement 1119
PREPARE statement 1120
PREPARE TO COMMIT statement 1123
PUT statement 1124
READ statement 1126
RELEASE SAVEPOINT statement 1127
RESIGNAL statement 1128
RESUME statement 1129
RETURN statement 1130
REVOKE statement 1132
REVOKE CONSOLIDATE statement 1134
REVOKE PUBLISH statement 1135
REVOKE REMOTE statement 1136
ROLLBACK statement 1137
ROLLBACK TO SAVEPOINT statement 1138
ROLLBACK TRIGGER statement 1139
SAVEPOINT statement 1140
SELECT statement 1141
SET statement 1145
SET CONNECTION statement 1147
SET DESCRIPTOR statement 1148
SET OPTION statement 1149
SET SQLCA statement 1170
SIGNAL statement 1171
START DATABASE statement 1172
START ENGINE statement 1173
START SUBSCRIPTION statement 1174
STOP DATABASE statement 1175
STOP ENGINE statement 1176
STOP SUBSCRIPTION statement 1177
SYNCHRONIZE SUBSCRIPTION statement 1178
SYSTEM statement 1179
TRUNCATE TABLE statement 1180
UNION operation 1181
UNLOAD TABLE statement 1182
UPDATE statement 1183
UPDATE (positioned) statement 1186
VALIDATE TABLE statement 1187
WHENEVER statement 1188

42 **SQL Anywhere Database Error Messages 1191**
Error message index by SQLCODE 1192
Error messages index by SQLSTATE 1201
Alphabetic list of error messages 1210

Internal errors (assertion failed) 1296

43 **SQL Preprocessor Error Messages 1297**
SQLPP errors 1298
SQLPP warnings 1307

44 **Differences from Other SQL Dialects 1309**
SQL Anywhere features 1310

45 **SQL Anywhere Limitations 1313**
Size and number limitations 1314

46 **SQL Anywhere Keywords 1315**
Alphabetical list of keywords 1316

47 **SQL Anywhere System Procedures and Functions 1319**
System procedure overview 1320
Catalog stored procedures 1321
System extended stored procedures 1323

48 **SQL Anywhere System Tables 1329**
System tables diagram 1330
Alphabetical list of system tables 1331

49 **SQL Anywhere System Views 1355**
Alphabetical list of views 1356

50 **Glossary 1363**

About This Manual

Subject

This manual describes how to use the Sybase SQL Anywhere database management system, and provides tutorial, task-based, and reference material to assist you in learning all aspects of the product.

Audience

This manual is for anyone using SQL Anywhere, whether as a developer or as an administrator.

Where to find information

SQL Anywhere is a multi-platform DBMS with many different uses. The table lists some of the starting points for using the *SQL Anywhere User's Guide*.

If you want to...	Go to...
Get an overview of the SQL Anywhere database management system	Part One: Introduction to SQL Anywhere
Learn how to create and manage databases in a Windows 95 or Windows NT environment.	"Managing Databases with Sybase Central"
Learn about the SQL language for querying data.	Part Two: Tutorials
Learn how to create and manage databases in an OS/2 or command-line environment.	The chapter "Working with Database Objects"
Learn how to connect to a SQL Anywhere database from a client application.	The chapter "Connecting to a Database"

PART ONE

Introduction to SQL Anywhere

This part provides an overview of the Sybase SQL Anywhere database management system, previously called Watcom SQL. The part includes a description of the new features in this release.

CHAPTER 1

Introduction

About this chapter

This chapter introduces the SQL Anywhere and the SQL Anywhere User's Guide.

Contents

Topic	Page
About SQL Anywhere	4
Upgrading databases to SQL Anywhere	7
About this manual	8
Product installation	11
Contact information	11

About SQL Anywhere

A full SQL database management system

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 fewer resources (memory space, disk space, and CPU speed) than other database management systems.

SQL Anywhere is available for Windows NT, Windows 95, Windows 3.x, OS/2, NetWare, Solaris/Sparc, HP-UX, AIX, DOS, and QNX operating systems.

SQL Anywhere can be used as a standalone database management system or (on non-UNIX operating systems) as a network database server in a client/server environment.

Database files are compatible between versions and across all operating systems. The operating systems on which you are entitled to run SQL Anywhere depends on your license agreement.

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

ODBC support

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.

Supplied components

Sybase SQL Anywhere is a multi-platform database management system that is available in both Standard and Professional Editions, and which is included as a part of several other products, including Powersoft PowerBuilder, Optima++, and S-Designor. Depending on which operating system you are using, and which particular product you have purchased, you may have access to some or all of the features described in this manual.

The components available to you depend on the operating system you are running, and on the particular product you have purchased. This section describes the contents of the SQL Anywhere Standard Edition. Additional tools and components are included in SQL Anywhere Professional. If you received SQL Anywhere as part of another product, consult the product documentation to see which components are available to you.

SQL Anywhere components

The SQL Anywhere Standard Edition CD-ROM includes the following components. Your use of these components is subject to the software license agreement.

- **Standalone database engine** For same-machine database access.
- **Network server and client** For client/server database access across a network.
- **SQL Remote** Message-based replication technology.
- **Sybase Central** Graphical administration utility
- **ISQL** Interactive SQL utility.
- **Command-line administration utilities** For database management tasks.

A DBMS for many platforms and environments

SQL Anywhere is available for the following operating systems:

- **Windows and Windows NT** The standalone database engine and multi-user network server are available for each of Windows 3.x, Windows 95, and Windows NT. The Sybase Central administration utility is available for Windows 95 and Windows NT only.
- **OS/2** The standalone database engine and multi-user network server are available for OS/2. These include a Presentation Manager version of ISQL, and command-line administration utilities. Sybase Central is not available for OS/2.
- **NetWare** The network server is available as a NetWare Loadable Module (NLM). The ISQL utility is also available as an NLM.
- **UNIX** The standalone engine is available for Sun Microsystems Solaris/Sparc, Hewlett Packard HPUX, and IBM AIX. This includes a character-mode ISQL and command-line administration utilities.
- **QNX** The network server only is available for QNX. This includes a character-mode ISQL and command-line administration utilities.

- **DOS** The standalone database engine and multi-user network server are available for DOS. This includes a character-mode ISQL and command-line administration utilities.

For information about supported releases of these operating systems, see your *Read Me First.*

Upgrading databases to SQL Anywhere

Which databases need to be updated?

You do not need to upgrade your SQL Anywhere 5.0 databases to use SQL Anywhere 5.5.

If you have databases created with Watcom SQL, you need to upgrade your database to SQL Anywhere 5.0 or 5.5 if you wish to use some of the features introduced in SQL Anywhere. The following summary describes the version you must have in order to use some features.

- To use procedures and triggers, you must be using a database that is at least release 4.0.
- To use SQL Remote on existing databases, you must upgrade your database to release 5.0.
- To access the Transact-SQL system procedures and system views, you must upgrade your database to release 5.0.

How to upgrade a database

The Upgrade utility is provided to upgrade Watcom SQL 4.0 databases to SQL Anywhere format. The Upgrade utility is accessible either from Sybase Central or as the DBUPGRAD command-line utility. The following statement at the command line upgrades a 4.0 database file named TEST.DB to a SQL Anywhere format:

```
dbupgrad -c "dbf=test.db;uid=dba;pwd=sql"
```

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.

About this manual

This manual describes how to build, run, and maintain databases using SQL Anywhere, and how to build database applications that work with the SQL Anywhere standalone database engine and network database server.

The manual includes the following sections.

- **Tutorials**
- **Using SQL Anywhere**
- **Transact-SQL Compatibility**
- **The SQL Anywhere Programming Interfaces**
- **Reference**

Platforms and products covered

This documentation applies to all supported platforms of SQL Anywhere. Not all features are available on all platforms. The operating systems and features you are entitled to use depends on the product you have purchased and your license agreement.

If you have obtained SQL Anywhere as part of another product, such as Powersoft PowerBuilder, S-Designor, or Optima++, you may not have access to all features described in this manual.

Documentation overview

Tutorials

The Tutorials part provides a step by step introduction to the following:

- Managing databases with Sybase Central
- Using the Interactive SQL (ISQL) utility
- Structured Query Language (SQL).

The chapter "Using ISQL" shows you how to use ISQL, and the rest of the tutorial introduces most of the important features of SQL, using a sequence of examples based on the supplied sample database.

Using SQL Anywhere

Each chapter in the part "Using SQL Anywhere" describes a different aspect of the product. Taken together, they provide a guide to building, running, and maintaining databases and database applications with SQL Anywhere.

Transact-SQL Compatibility

Transact-SQL Compatibility is for users wishing to develop applications and databases compatible with Sybase SQL Server and the Transact-SQL dialect of SQL it supports. The part describes the SQL Anywhere support for Transact-SQL.

The SQL Anywhere Programming Interfaces

The SQL Anywhere Programming Interfaces explains how to use the programming interfaces to SQL Anywhere. The bulk of the section describes the Embedded SQL and ODBC interfaces for use by C programmers. Developers using ODBC-enabled application development systems, such as Powersoft PowerBuilder, do not need to use this section: the information in Using SQL Anywhere and Reference, together with your application development system's documentation, cover all the information needed to develop SQL Anywhere applications from ODBC-enabled applications.

Reference

Reference is a complete reference guide to SQL Anywhere programs, supported commands with complete syntax, error messages, and system tables.

Available formats

The contents of this manual are provided in print and as online Help, accessible through Sybase Central, ISQL, and as a desktop icon. The online Help version of the manual is more up to date than the printed documentation.

Document conventions

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

Typographic conventions

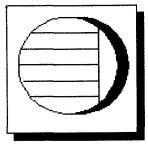
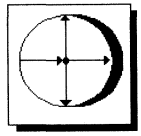
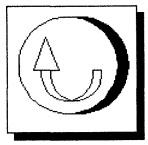
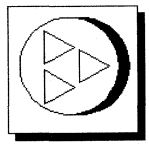
The documentation uses the following typographic conventions:

Item	Description
`Code`	SQL and program code is displayed in a monospaced font
User entry	Text entered by the user is displayed in a bold sans serif typeface
new term	New terms introduced in the text are displayed in a bold serif typeface
emphasis	Emphasized words are displayed in italic
SQL keywords	SQL keywords are displayed in upper case

For a description of syntax-formatting conventions, see "Syntax conventions" in the chapter "Watcom-SQL Language Reference".

Graphic icons

The following icons are used in this documentation:

Icon	Meaning
	A database server, such as Sybase SQL Server or Sybase SQL Anywhere.
	A SQL Anywhere database engine, or a SQL Anywhere network server and Client. All applications that work with a SQL Anywhere standalone engine work identically with a SQL Anywhere network server, via a SQL Anywhere Client.
	A Sybase Replication Server.
	A Replication Agent. A replication agent is required for a database to act as a primary data site in Sybase Replication Server installations.
	A SQL Anywhere Log Transfer Manager.
	A client application.

Product installation

Installation instructions for SQL Anywhere are in the *SQL Anywhere Read Me First.*

Contact information

Contact information for technical support and sales inquiries is provided on the separate Contact Sheet in this package.

CHAPTER 2

New Features in SQL Anywhere 5

About this chapter

This chapter describes features new to SQL Anywhere 5.

Contents

Topic	Page
What's in a name?	14
New features overview	15
New features in the Watcom-SQL language	18
New sample database	21

What's in a name?

Previous releases of this product were named Watcom SQL.

With the Sybase/Powersoft merger of February 1995, Watcom became a part of Sybase, Inc. Watcom SQL is now a part of the Sybase System 11 Server product line, and the new name of the product, SQL Anywhere, reflects that change. The principal dialect of SQL supported by SQL Anywhere is named Watcom-SQL.

The Sybase SQL Anywhere name reflects the versatility of the product, and the emphasis on both scalability and mass deployment. Sybase SQL Anywhere gives you the high performance and reliability of a transaction-processing SQL server, the ability to scale from single users to many users, and the flexibility to deploy on multiple platforms. It's a database designed for the workplace.

The new features outlined in this chapter include data replication and Transact-SQL-compatibility, along with many others. We think you will agree that these new features, built on the proven foundations of Watcom SQL, make SQL Anywhere a powerful workplace complement to Sybase SQL Server in the System 11 Server product line.

New features overview

While there have been many improvements and enhancements, the following are major areas around which the new release has been designed.

Feature	Description	For more information
SQL Remote data replication	SQL Remote is a message-based replication system for replication between SQL Anywhere databases. It is built to be easy to use, centrally administered, and ideal for replication to laptop and other occasionally-connected users' personal databases.	See chapters "Introduction to SQL Remote Replication" and "SQL Remote Administration."
Sybase Central database management tool	Sybase Central is a graphical database administration tool that runs on the Windows 95 and Windows NT 3.51 operating systems. Sybase Central conforms closely to the Windows 95 interface guidelines, making it easy to learn and use.	See the chapter "Managing Databases with Sybase Central"
Transact-SQL compatibility	SQL Anywhere 5.0 includes a set of extensions to the Watcom-SQL language from the Sybase Transact-SQL dialect. This makes the development of compatible applications for SQL Anywhere and SQL Server database servers much more straightforward, and also brings new features to all SQL Anywhere users.	See the chapters "Using Transact-SQL with SQL Anywhere" and "Transact-SQL Procedure Language".
ODBC 2.5 support	SQL Anywhere 5.0 now supports ODBC 2.5 at level 2 for the Windows 95 and Windows NT operating systems, and ODBC 2.1 for the Windows 3.x operating system.	

Feature	Description	For more information
Open Server Gateway	The Open Server Gateway. allows client applications to work with both SQL Server and SQL Anywhere database servers	See the chapter "Using the Open Server Gateway".
Data replication with Sybase Replication Server	The Open Server Gateway allows SQL Anywhere to act as a replicate site database in Sybase Replication Server installations. A separate product, the SQL Anywhere Replication Agent, allows SQL Anywhere databases to act as primary data sites in Replication Server installations	
Performance improvements	Performance enhancements do not require extensive documentation. However, there are major improvements to the speed of many tasks performed by SQL Anywhere since the last major release	
System functions and performance monitoring	An extensive set of system functions allows access to statistics concerning database engine or server performance. These are also available from Sybase Central. Statistics from the Windows NT engine and network server can be viewed in the NT Performance Monitor.	See the chapter "Monitoring and Improving Performance".
SQL extensions	The Watcom-SQL language has been extended in many areas to provide more flexibility and power to SQL Anywhere users	See "New features in the Watcom-SQL language" on page 18.
Transaction log mirroring	SQL Anywhere can optionally maintain two identical transaction logs. Transaction log mirroring provides additional protection against loss of data due to disk failure	For more information on transaction log mirroring, see the chapter "Backup and Data Recovery".
MAPI system procedures	System external procedures are supplied that allow MAPI e-mail calls to be made from stored procedures	See the chapter "Using Procedures, Triggers, and Batches".

Feature	Description	For more information
Calls to external DLLs from procedures	You can now create procedures that call external DLLs	See the chapter "Using Procedures, Triggers, and Batches".

New features in the Watcom-SQL language

Watcom-SQL is the dialect of SQL supported by the SQL Anywhere standalone database engine and network server. The extensions to Watcom-SQL in this release include improvements in several areas.

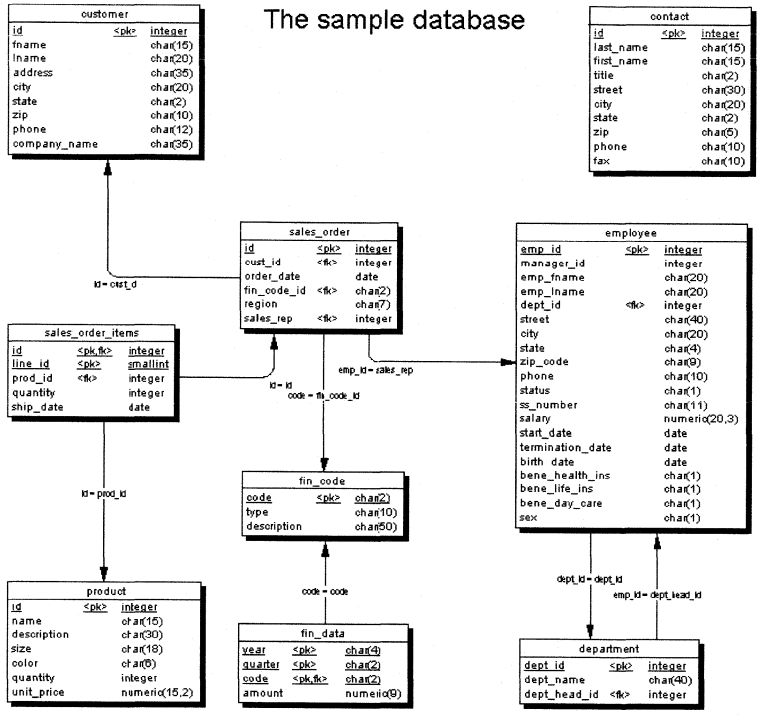
Feature	Description	For more information
Stored procedure extensions	Stored procedures are easier to create, have more flexible parameter declarations (including default values and optional parameters in CALL statements), and can return status information.	See the chapter "Using Procedures, Triggers, and Batches".
User-defined functions	With the new CREATE FUNCTION statement you can use the SQL Anywhere stored procedure language to define new functions, you can use just like other functions.	See the chapter "Using Procedures, Triggers, and Batches".
Batches	This release introduces support for batches of SQL statements. Control statements (IF, LOOP, and so on) are now available in command files as well as in procedures and triggers.	See the chapter "Using Procedures, Triggers, and Batches".
Statement level triggers	Triggers can be made to fire once after each statement, as an alternative to the row-level triggers supported in previous releases	See the chapter "Using Procedures, Triggers, and Batches".
New Watcom-SQL language elements	These include global variables, new operators, and new comment specifiers	See "Watcom-SQL language elements" in the chapter "Watcom-SQL Language Reference".

Feature	Description	For more information
New built-in functions	New numeric, string, date and time, and data type conversion functions have been added, as well as new system functions. New functions include: ♦ PATINDEX for pattern matching ♦ System functions describing database properties	See the section "Watcom-SQL Functions".
User-defined data types	You can define your own data types from the existing base data types supported by SQL Anywhere	See the section "SQL Anywhere Data Types".
Extensions to data manipulation statements	The DELETE, and UPDATE statements have been extended to support criteria based on joins.	See "DELETE statement" in the chapter "Watcom-SQL Statements" and "UPDATE statement" in the chapter "Watcom-SQL Statements".
System stored procedures	SQL Server provides stored procedures for carrying out database management functions. The system tables of SQL Server and SQL Anywhere are different, but several of the system stored procedures are provided to carry out analogous actions on each.	
More flexible column constraints	Before this release, all constraints associated with a table, whether column constraints or table constraints, were held as a single table constraint. Column constraints are now held individually, allowing them to be individually deleted or replaced	See "ALTER TABLE statement" in the chapter "Watcom-SQL Statements"

Feature	Description	For more information
Other new statements	Other new statements have been added: LOAD TABLE, UNLOAD TABLE, TRUNCATE TABLE, MESSAGE, EXECUTE IMMEDIATE, RETURN, and others	See the listing in section "Watcom-SQL language elements" in the chapter "Watcom-SQL Language Reference".

New sample database

The sample database has been changed in this release. The new sample database contains employee, sales, product and financial information for a small company. The database is a subset of the PowerBuilder demo database. The following diagram illustrates the sample database structure.



The sample database is called SADEMO.DB.

CHAPTER 3

Overview of SQL Anywhere

About this chapter

This chapter presents an overview of SQL Anywhere architecture on a single computer and on a network.

An application developed in a standalone environment can be deployed in a multiuser network environment with no alteration to the code whatsoever.

This chapter also introduces terminology for relational database management systems, and describe the database management tools included in your SQL Anywhere package.

Contents

Topic	Page
The SQL Anywhere engine and the SQL Anywhere server	24
Running SQL Anywhere on a single computer	25
Running mixed operating systems on a single computer	32
SQL Anywhere programming interfaces	34
The SQL Anywhere programs	36

The SQL Anywhere engine and the SQL Anywhere server

SQL Anywhere includes two executables for managing databases:

- The network server, for managing databases using a client/server arrangement on networks
- The database engine, for managing databases on a single computer in a standalone mode

PowerBuilder and InfoMaker users
This chapter contains some information about connecting to a SQL Anywhere network server. The SQL Anywhere network server is not included with PowerBuilder and InfoMaker.

The SQL Anywhere server and SQL Anywhere engine manage databases in exactly the same way and are completely compatible. However, the SQL Anywhere engine has no support for network communications.

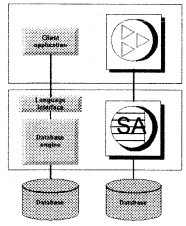
Server and engine
The terms **engine** and **server** are used in this manual to refer to the database engine and database server. In contexts where either the engine or the server could be used, the terms are used interchangeably. When the engine and server need to be distinguished, the terms **standalone engine** and **network server** are sometimes used to emphasize which is meant.

The following sections describes the architecture for running SQL Anywhere on a single computer using the SQL Anywhere engine and for running SQL Anywhere across a network using the SQL Anywhere network server.

For information on running more than one operating system on a single computer, see the section "Running mixed operating systems on a single computer" on page 32.

Running SQL Anywhere on a single computer

The figure shows the architecture of a standalone SQL Anywhere installation, running a single database engine and working with a single database. All more complicated arrangements are elaborations of this basic setup, so you should understand how the basic setup works, even if you are operating a multiuser client/server installation.



Standalone SQL Anywhere components

The components of the basic standalone SQL Anywhere setup are:

- The client application
- The SQL Anywhere interface layer
- The SQL Anywhere database engine
- The database

Database users do not directly manipulate database files. Instead, their client application communicates with the **database engine**, using a programming interface supported by SQL Anywhere, and the database engine handles all manipulation of the actual database.

For information on a client application and a database server *for different operating systems* running on the same computer, see the section "Running mixed operating systems on a single computer" on page 32.

The client application

A client application communicating with the SQL Anywhere database engine must do so using a **programming interface** supported by SQL Anywhere. The client application calls functions from one of the SQL Anywhere programming interfaces.

The client application together with the programming interface layer form the **client side** of the setup.

ODBC-enabled application development systems

If you are building a client application from an ODBC-enabled application development system, such as Powersoft PowerBuilder, or others, you do not need the information in the section "Programming Interfaces". These application development systems already implement an ODBC interface internally. Application development systems form an extra layer that sits between the application developer and the SQL Anywhere interface, and you should consult your application development system's documentation to understand how to communicate with a database engine.

PowerBuilder and InfoMaker use the ODBC interface.

If you are using one of the programming interfaces directly to develop client applications, you should understand the SQL Anywhere programming interface you are using. The programming interfaces are described fully in the section "Programming Interfaces".

The SQL Anywhere database engine

The database engine and the database together form the **server side** of the setup. A client application manipulates a database by sending requests to the database engine.

SQL Statements

Communications between a client application and a database engine take the form of Structured Query Language (**SQL**) statements. For example, a SELECT statement, or query, is used to extract information from a database. An UPDATE statement may be used to modify the contents of one of the database tables.

The client application sends the SQL statements and the database engine processes them and sends the results back to the client application.

Running an application against a SQL Anywhere network server

Running an application against a SQL Anywhere network server generally requires an extra component to handle network communications from the client computer (see "Running SQL Anywhere on a network" on page 29). However, for applications on the same computer the database server can be run in exactly the same manner as the standalone database engine described here using a direct connection to the network server.

If a SQL Anywhere Client is running and the network server is running on the same machine, the Client will not be used by client applications connecting on the same machine; the connection will be direct.

For a discussion of running a client application and a database server for different operating systems on a single computer, see "Running mixed operating systems on a single computer" on page 32.

The database

SQL Anywhere is a relational database system. The database itself is stored on one or more disk drives, and consists of the following objects:

Database objects	Description
Tables	Hold the information in the database
Keys	Relate the information in one table to that in another
Indexes	Allow quick access to information in the database
Views	Are computed tables
Stored procedures	Hold queries and commands that may be executed by any client application (stored procedures are not available in the SQL Anywhere Desktop Runtime system)
Triggers	Assist in maintaining the integrity of the information in the database (triggers are not available in the SQL Anywhere Desktop Runtime system)
System tables	Hold the information about the structure of the database

Multiple databases on a single database engine

A single SQL Anywhere database engine can manage access to several databases simultaneously. You can start and stop databases from database administration tools or client applications, and you may connect to any of the currently running databases on a database engine.

As far as the database user is concerned, interaction with a database engine is always through a **connection**. Each time users connect to a database, supplying valid user ID and password, they are connected to a specific database on a specific database engine. Once a connection is established, it provides a channel through which all communications go. The connection insulates the user from the other components of a running database system such as network sessions and interprocess communication mechanisms.

Multifile databases

SQL Anywhere supports multifile databases. When a SQL Anywhere database is first initialized, it is composed of one file—the **root file**. As tables and other database objects are added to the database, however, they may be stored in different files, which may be on different disk drives from the root file.

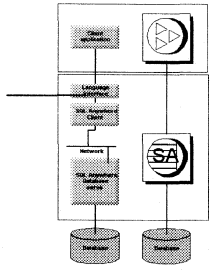
Users of the database (other than the database administrator) need not be aware of the physical location of database files. The database engine handles all access to the files and shields this complexity from the user.

Running SQL Anywhere on a network

SQL Anywhere Server products support connections from many users at a time, over a network. In this case, the database engine runs on one computer (the **database server computer**), while client applications run on other computers (**client computers**).

The SQL Anywhere server supports multiuser network access to SQL Anywhere. The SQL Anywhere standalone engine does not support multiuser access or network communications.

PowerBuilder and InfoMaker users
This chapter contains some information about connecting to a SQL Anywhere network server. The SQL Anywhere network server is not included with PowerBuilder and InfoMaker.



The client side of the SQL Anywhere setup sends SQL queries and commands over the network to the server side, which carries out commands and sends the results of queries back to the client.

The SQL Anywhere Client

The SQL Anywhere Client is a program that handles network communications to the SQL Anywhere database server. The SQL Anywhere Client is a program named DBCLIENT (the Windows 3.x version is called DBCLIENW). The SQL Anywhere Client communicates with the SQL Anywhere network server. The standalone engine cannot handle communications from the SQL Anywhere Client.

For QNX, the database client is different from other operating systems. Instead of being a separate executable, the SQL Anywhere Client is a library named **dbclient**, which is loaded dynamically by client applications.

Standalone setup versus network setup

From the client application's point of view, there is no difference between the standalone setup and the network setup. In the single-user setup, a client application sends requests and commands to the database engine. In the multiuser setup, these requests are sent to the SQL Anywhere Client instead. In each case, the client application has a single point of contact with the DBMS. The additional complexity of handling requests in a multiuser, networked environment is hidden from the client application.

Standalone applications work with the SQL Anywhere network server
Once a client application is developed and working on a standalone SQL Anywhere setup on a single computer, no changes are needed to the application in order for it to work as a client application in a network environment against a SQL Anywhere server.

For more information on the SQL Anywhere network server, see the SQL Anywhere Network Guide.

SQL Anywhere multiplatform support

The SQL Anywhere standalone database engine is available for the Windows 3.x, Windows 95 and Windows NT, OS/2, and DOS operating systems. The SQL Anywhere database server is available for Novell NetWare, Windows 95 and Windows NT, OS/2, Windows 3.x, DOS, and QNX operating systems.

One SQL Anywhere database server can support multiple clients operating on different operating systems, communicating via different network protocols.

Some database terms

With a single database running on a single database engine, the default settings make the engine name, the database name, and the database file the same, apart from the path and extension associated with the file. In this situation there is little ambiguity when talking about *the database*.

In environments with multiple databases, multiple database files, and several database engines operating simultaneously, it is important to distinguish among the different components that make up a running SQL Anywhere DBMS.

Terms used in this documentation

Term	Description
Database file	Even though the tables of a database may be held in several files on several disk drives, each database is identified by a single **root file**. Throughout this book, when reference is made to a **database file**, it is referring to the root file
Database name or alias	A SQL Anywhere database engine or server can run several databases simultaneously, managing access to each of them. When a database is started on a database engine, it is assigned a **database name**, also called the **database alias**. If no database name is explicitly assigned, the database receives the name of the root file with the path and extension removed
Server or engine name	When a database server or engine is started, it is assigned a **server name** or **engine name**. The server or engine name is entirely distinct from the name of the database engine program itself. By default, the server name is the first database name. For example, if a database engine is started with database C:\PB\EX\PSDEMODB.DB and no name is explicitly given, then the name of the engine is PSDEMODB
Clients and servers	Both the client side and the server side of a SQL Anywhere client/server setup consist of several components. The terms **client** and **server** themselves are commonly used to describe not only the computers on which each side of the setup sits, but also the programs that are communicating, and also the collection of software components on each of the computers. Throughout this guide, the terms *client* and *server* are qualified whenever possible to specify which of these meanings is used

Running mixed operating systems on a single computer

SQL Anywhere supports situations in which client applications and the database engine run under different operating systems on the same computer. This support requires a SQL Anywhere Client (DBCLIENT) for the client application operating system. You can think of the mixed operating system setup as a client/server network arrangement, with both client and server sides residing on the same computer.

The situations where this can occur are:

- DOS or Windows client applications with an OS/2 engine
- DOS or Windows 3.x client applications with a Windows 95 or Windows NT engine

Mixed operating systems can occur when running DOS or Windows 3.x client applications with a Windows 95 or Windows NT engine

DOS or Windows client applications on OS/2

The OS/2 standalone engine or network server can be accessed from a DOS or WIN-OS/2 client application on the same machine.

The client application communicates with the SQL Anywhere Client for the client application operating system. (The DOS and Windows 3.x SQL Anywhere Clients are installed with *SQL Anywhere for OS/2* or as a separate install with the DESKTOP RUNTIME SYSTEM FOR OS/2.) The SQL Anywhere Client handles the communication with the OS/2 database engine or server using named pipes or DDE.

Command lines for DOS or Windows 3.x clients

The command line to run the DOS client is:

dbclient ***engine-name***

The command line to run the Windows 3.x client is:

dbclienw ***engine-name***

Other servers

SQL Anywhere also supports access to network servers elsewhere on the network simultaneously via a separate communication link in the SQL Anywhere Client.

For more complete information, see the SQL Anywhere Network Guide.

DOS or Windows 3.x clients on Windows 95 or NT

The Windows 95 or Windows NT database engine or server can be accessed from a DOS or Windows 3.x client application on the same machine.

The client application communicates with the SQL Anywhere Client for the client application operating system. (The DOS and Windows SQL Anywhere Clients are installed with SQL Anywhere.) The SQL Anywhere Client handles the communication with the Windows 95 or Windows NT database engine or server, using named pipes (Windows NT) or DDE (Windows 95).

Command lines for DOS or Windows 3.x SQL Anywhere clients

The command line to run the DOS SQL Anywhere Client is:

dbclient *engine-name*

The command line to run the Windows 3.x SQL Anywhere Client is:

dbclienw *engine-name*

Other servers

SQL Anywhere also supports access to network servers elsewhere on the network simultaneously via a separate communication link in the SQL Anywhere Client.

For more information, see the SQL Anywhere Network Guide.

SQL Anywhere programming interfaces

The conversation between a client application and a database engine or server takes place through a SQL Anywhere **programming interface**. The available programming interfaces are:

- High level interfaces in ODBC-enabled application development systems such as Powersoft PowerBuilder and in applications with ODBC-enabled macro languages
- Low-level full-function interfaces: **ODBC** and **embedded SQL**
- High-level interfaces: **WSQL DDE** and **WSQL HLI**

This section describes the main characteristics of the high and low-level interfaces.

For more detailed information and guidelines to help you choose the most appropriate interface, see "Programming Interfaces".

Low-level programming interfaces

ODBC and embedded SQL provide low-level interfaces to SQL Anywhere.

ODBC is supported by a wide range of DBMS and is implemented for Windows, Windows NT, and OS/2 as a database driver: a dynamic link library (DLL) that an application invokes to gain access to SQL Anywhere databases. On other platforms, the ODBC interface is implemented as a static library.

Embedded SQL is slightly quicker than ODBC and has a wider set of functions. Embedded SQL is implemented as a C/C++ language preprocessor, which translates SQL statements embedded in your code into calls to the interface. The interface is implemented as a DLL for Windows, Windows NT, and OS/2, and as a library on the other platforms.

The ODBC interface

The ODBC interface is the most widely-used interface to SQL Anywhere.

SQL Anywhere supports the Microsoft **Open Database Connectivity** (**ODBC**) interface not only in the Windows and Windows NT environments, but also on DOS, OS/2, the Macintosh, and QNX. This low-level interface provides almost all the functionality of embedded SQL with, (in the SQL Anywhere implementation), only a small performance penalty. Client applications using the ODBC interface can work with many different RDBMS.

SQL Anywhere supports all of the ODBC Version 2.1 API functions (Core, Level 1, and Level 2).

For a full description of ODBC programming, see the chapter "ODBC Programming".

The embedded SQL interface

The native programming interface of SQL Anywhere is embedded SQL. SQL Anywhere comes with dynamic link libraries (normal libraries in DOS, QNX, and NetWare) and a preprocessor to enable development of C and C++ applications using embedded SQL.

For a full description of embedded SQL programming, see the chapter "The Embedded SQL Interface".

High-level programming interfaces

SQL Anywhere also provides two higher-level programming interfaces, DDE (Windows and Windows NT only), and HLI (Windows, OS/2, and Windows NT only). High-level interfaces to SQL Anywhere are also provided by ODBC-enabled application development systems and applications.

The DDE interface

The DDE server is a Windows application that enables you to access and alter data in SQL Anywhere databases using **dynamic data exchange (DDE)**. Many Windows applications, including leading spreadsheets and word processors, support the DDE protocol.

The HLI interface

The HLI is provided as a DLL for Windows, OS/2 and Windows NT and can be used from any application or environment that can call DLLs. It is simpler to use than ODBC, but also slower and supports less functionality.

The SQL Anywhere programs

SQL Anywhere includes a set of database administration tools, as well as the SQL Anywhere database engine itself.

Each of the database tools is a SQL Anywhere client application and communicates with the database engine using the embedded SQL interface.

The Windows 3.x version of many of the programs has a slightly different name (ending with a W) so that Windows 3.x applications can coexist in the system path with Windows 95, DOS, OS/2, or Windows NT applications.

For reference information about each of the utilities in your SQL Anywhere package see the chapter "SQL Anywhere Components".

The SQL Anywhere database engine and server

At the core of SQL Anywhere is the database engine or database server, which handles all requests from client applications and carries out all manipulations of the database.

The name of the database engine or server executable depends on the operating system, as follows:

Operating system	Database Engine or Server Executable
SQL Anywhere, for Windows 3.x	DBENG50W.EXE is a 32-bit database engine, and DBENG50S.EXE is a 16-bit database engine provided for computers that can run Windows 3.x only in standard mode
SQL Anywhere, for operating systems other than Windows 3.x	DBENG50.EXE
SQL Anywhere Desktop Runtime System for Windows 3.x	RTDSK50W.EXE is a 32-bit database engine, and RTDSK50S.EXE is a 16-bit database engine provided for computers that can run Windows 3.x only in standard mode. The runtime database engines are distributable royalty-free on purchase of the SQL Anywhere Desktop Runtime System

Operating system	Database Engine or Server Executable
SQL Anywhere Desktop Runtime System, for operating systems other than Windows 3.x	RTDSK50.EXE. The runtime database engine is distributable royalty-free on purchase of the SQL Anywhere Desktop Runtime System
SQL Anywhere database server for Windows 3.x	DBSRV50W.EXE is a 32-bit database engine
SQL Anywhere database server, for operating systems other than Windows 3.x	DBSRV50.EXE, DBSRV50.NLM for the NetWare Loadable Module and DBSERVER for QNX

The SQL Anywhere Client, DBCLIENT.EXE, enables client applications to communicate with the network database server. For Windows 3.x, the SQL Anywhere Client executable is named DBCLIENW.EXE.

For QNX, the SQL Anywhere Client is a library named DBCLIENT, which is loaded at run time.

The SQL Anywhere Desktop Runtime database engine

The SQL Anywhere Desktop Runtime System contains a runtime SQL Anywhere database engine that supports database manipulation language (DML) SQL commands.

What is it?

The runtime version of the database engine:

- Is a transaction-processing database engine
- Fully supports the DML subset of SQL commands, such as SELECT, INSERT, UPDATE, and DELETE
- Provides full referential integrity, including cascading updates and deletes
- Supports declared temporary tables
- Allows users to be added to a database
- Works with existing databases

Runtime and standalone database engine differences

The functions supported by the SQL Anywhere standalone database engine that are not fully supported in the runtime database engine are as follows:

Standalone database engine function	How it operates in the runtime database engine
Replication	You cannot replicate data
ALTER statements	You cannot use: ♦ ALTER DBSPACE to modify the characteristics of the main database file or extra dbspace ♦ ALTER TABLE to change table definitions ♦ ALTER PROCEDURE to change procedure definitions
CREATE statements	You can use CREATE VARIABLE to create SQL variables. SQL variables do not form part of the database definition You cannot use the following CREATE statements: ♦ CREATE DATATYPE to create user-defined data types ♦ CREATE DBSPACE to create new database files ♦ CREATE INDEX to create indexes ♦ CREATE PROCEDURE to create procedures ♦ CREATE FUNCTION to create user-defined functions ♦ CREATE TABLE to create tables or temporary tables ♦ CREATE TRIGGER to create triggers ♦ CREATE VIEW to create views
COMMENT statement	You cannot add comments to the system table
DROP statement	You cannot use the DROP statement to drop a DATATYPE, DBSPACE, FUNCTION, INDEX, TABLE, VIEW, PROCEDURE, or TRIGGER from the system tables
CALL statement	You cannot invoke procedures

Standalone database engine function	How it operates in the runtime database engine
GRANT statement	You can grant the special user permissions (CONNECT, DBA, RESOURCE, GROUP, MEMBERSHIP IN GROUP) using GRANT, but you cannot change permissions on tables and views
REVOKE statement	You can revoke the special user permissions using REVOKE, but you cannot change permissions on tables and views
Triggers	Triggers will not be fired by the runtime database engine
No transaction log	To simplify database administration, the runtime database engine does not employ a transaction log, Users of the runtime database engine should backup their databases by making backup copies of their database file itself
The database engine options that refer to transaction logs (-a,-f)	Not applicable to the runtime database engine
The DBBACKUP options described in the User's Guide that refer to transaction logs (-r, -t, -x)	Not applicable to the runtime database engine

The Sybase Central database management program

Sybase Central is a graphical database management tool for Windows 95 or for Windows NT 3.51 or later. Sybase Central is the recommended tool for carrying out the tasks described in the following sections, including the following:

- Creating and maintaining databases
- Backing up databases
- Managing transaction log files
- Working with compressed databases
- Working with read-only databases

- Compacting databases by unloading and reloading

SQL Anywhere administration utilities

For a complete description of each of the utilities that come with your SQL Anywhere package, see the chapter "SQL Anywhere Components".

You can access the administrative utilities from Sybase Central, which is the recommended tool for managing SQL Anywhere databases, from ISQL, or as command-line utilities.

Action	Utility	Description
Stopping a database server	**stop** utility	Stops a running database server or SQL Anywhere Client
Creating and maintaining databases	**initialization** utility	Creates a new database. If you wish to use a customized sorting and comparison order (collation) for a new database, you need to use the collation utility.
Modifying	**erase** utility	Erases a database file, log file, or write file
Inspecting database files	**information** utility	Provides information about a database file or write file
Modifying the transaction log	**transaction log** utility	Displays or changes the name of the transaction log. Transaction logs are a record of all changes made to a database, and are an important part of database backups. Transaction logs can be translated into ASCII SQL command file form by the log translation utility as part of recovery from some forms of database failure, and applied to a copy of the database made at the time the transaction log was started to restore the database
Backing up databases	**backup** utility	Performs full or incremental online backup of databases
Validating the database	**validation** utility	Checks the validity of all indexes on a database table, and should be used in conjunction with backups to maintain the integrity of information in a database

Action	Utility	Description
Working with compressed databases	**database compression** utility	Compresses a database file (compressed database files are usually 40 to 60 percent of their original size)
	uncompression utility	Expands a compressed file Compressed databases are useful in situations where file space is limited. The SQL Anywhere database engine is not able to update compressed database files directly, however, and so compressed databases need to be used in conjunction with a **write file** created with the write file utility.
Working with read-only databases	**write file** utility	Manages write files and provides a method of working with read-only databases. There are some situations where it is necessary to work with databases that SQL Anywhere cannot modify directly: read-only databases. Database provided on CD-ROM are a common class of read-only database. Compressed databases are read-only databases. If you wish to test new applications without modifying a production database, a write file is a useful alternative. All changes made to a read-only database are made instead to a write file, leaving the master database unchanged. The SQL Anywhere standalone engine and network server also produce a transaction log, which maintains records of all changes made to the write file. If, at a later point, you wish to apply the changes in the write file to the master database, you can do this by applying the transaction log to the original database using the log translation utility.

Action	Utility	Description
Unloading and reloading data	**unload** utility	Unloads the database. This program creates a text command file named RELOAD.SQL. This command file may be run from ISQL to rebuild the database from scratch. The size of a database file does not decrease when rows are deleted (although future insertions will use the space freed by the deleted rows). You can **rebuild** a database in order to reclaim the disk space associated with deleted rows. The REBUILD batch or command file (not available in the Windows 3.x version of SQL Anywhere) automates the process of rebuilding a database.
Upgrading databases	**upgrade** utility	Upgrades databases made using earlier versions of SQL Anywhere to SQL Anywhere version 5.0 format. For information about upgrading databases, see "The Upgrade utility" in the chapter "SQL Anywhere Components".

PART TWO

Tutorials

This part introduces you to the basics of the SQL language used to query and modify databases. If you are already familiar with SQL, you will not need to read this part.

CHAPTER 4

Managing Databases with Sybase Central

About this chapter

This chapter introduces the Sybase Central database management tool and is a brief introduction to how Sybase Central can be used for managing database properties.

Detailed instructions on using Sybase Central are available in the Sybase Central online Help. The online Help is arranged as a set of how to windows leading you step by step through database management tasks. In addition, context sensitive Help is available to explain the different interface elements.

Contents

Topic	Page
Sybase Central and database management	46
Navigating the main Sybase Central window	47
Adding a table to a database	52
Viewing and editing procedures	57
Managing users and groups	60
Backing up a database using Sybase Central	63
Using the Sybase Central online help	65

Before you begin

Sybase Central requires Windows 95, or Windows NT 3.51 or later.

Sybase Central and database management

Sybase Central is a database management tool that exposes SQL Anywhere database settings, properties, and utilities in a graphical user interface.

Database administration tasks typically fall into two categories:

- Tasks carried out by sending SQL statements to the database engine.
- Tasks carried out by SQL Anywhere utilities.

Sybase Central provides an easy-to-use interface for both kinds of tasks.

Using Windows 95 or NT

For users with access to Windows 95 or NT, Sybase Central will make your database administration tasks easier and more efficient.

Other systems

For SQL Anywhere users not running Windows 95 or Windows NT 3.5.1 or later, all the tasks you can carry out with Sybase Central can be carried out using ISQL to send SQL statements to the database engine or server, and using the command-line versions of the SQL Anywhere utilities.

For more information about SQL Anywhere utilities, see the chapter "SQL Anywhere Components".

Navigating the main Sybase Central window

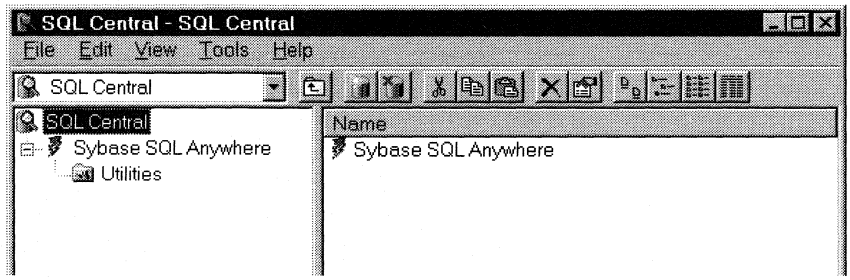
This tutorial introduces the Sybase Central user interface. It also describes how to start Sybase Central, how to connect to a database, and how to view a database schema in Sybase Central.

After completing the tutorial you should feel comfortable exploring Sybase Central's capabilities by yourself.

Start Sybase Central

- To start Sybase Central in Windows 95 or Windows NT 4.0, select Start➤Programs➤SQL Anywhere➤Sybase Central.
- To start Sybase Central from Windows NT prior to Version 4.0, double-click Sybase Central in the SQL Anywhere Program Manager group.

The main Sybase Central window appears.



The Sybase Central window is modeled after the Explorer in Windows 95 and Windows NT 4.0. The main window is split into two vertically-aligned panels. The left panel displays a hierarchical view of database objects or **containers** in a tree-like structure. A container is a database object that can hold other database objects, including other containers. Sybase Central is at the root of the tree. Plug-ins for Sybase Central, such as the Sybase SQL Anywhere database management system, occupy the first level after the root level.

The right panel displays the contents of the container that has been selected in the left panel. The contents of a container can be viewed in the right panel in several ways: as large icons, small icons, as a list, and alongside their associated details. You can switch between these views by clicking the buttons on the Tool Bar immediately below the Window menu.

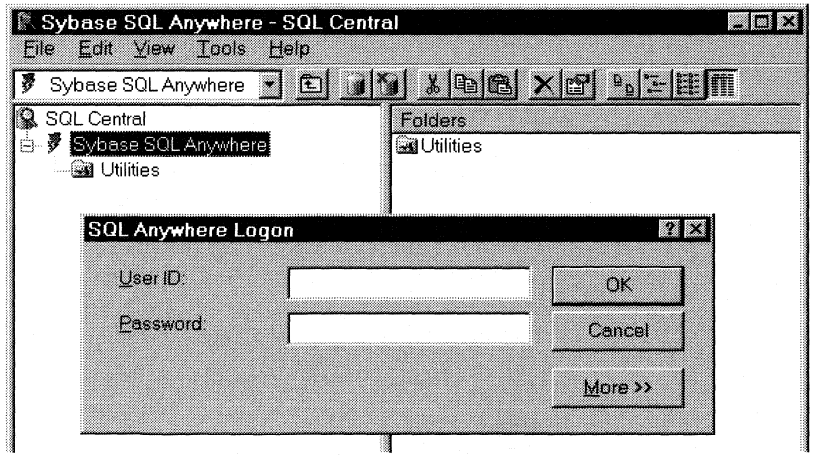
Connecting to a database from Sybase Central

This section describes how to connect to the sample database using the user ID **DBA** and the password **SQL**.

> **Logging on**
> The sample database contains the user ID "DBA", a user with full administration and resource creation rights. The password for this user is "SQL".
>
> By default, all newly-created SQL Anywhere databases contain this user ID and password. It is the responsibility of the database administrator to provide the desired level of security by changing passwords and creating other user IDs.

❖ **To connect to the sample database:**

1. Start the sample database (SADEMO.DB) by selecting the Standalone Sample Database menu item from the Sybase SQL Anywhere 5.0 menu. Select Start>Programs>SQL Anywhere 5.0>Standalone Sample Database.
   The SQL Anywhere engine and sample database must be running for any of the following steps to work properly.
2. From the same Sybase SQL Anywhere 5.0 menu, start Sybase Central by clicking the Sybase Central menu item.
3. On the Sybase Central toolbar, click the Connect toolbar button or select Connect from the Tools menu on the menu bar.
4. Enter the user ID **DBA** and the password **SQL**, and click OK.



You can save connection parameters as **connection profiles** to avoid retyping often-used connection parameters. For information about connection profiles, search the Sybase Central online help index for "Connection Profiles".

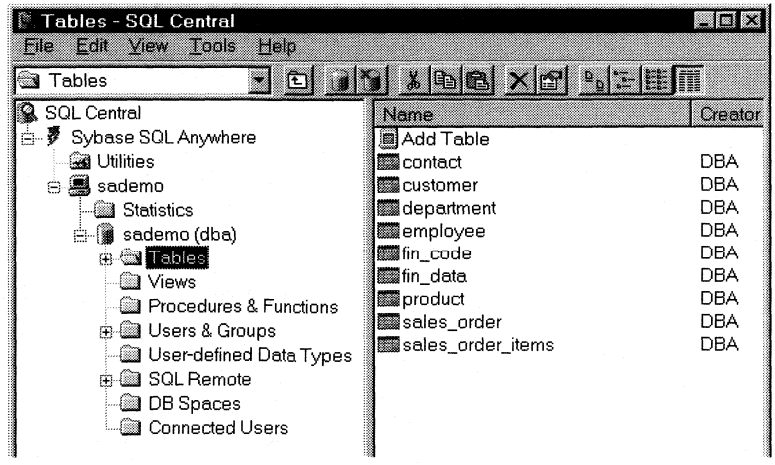
Viewing a database schema

A database **schema** is the collection of all objects in the database. Sybase Central displays a database schema as a hierarchy of containers and their contents. This section describes how to view the schema of a database.

Expanding a database container

There are a variety of methods for viewing the objects in a database, including the following.

- Click a container in the left panel to select that container. The right panel then shows the contents of the selected container. Among the contents of the server container are all the databases you've attached to, including ones you have not yet connected to by entering a user ID and password.
- Click once on the plus or minus sign next to containers in the left panel to expand or collapse the hierarchical tree of objects. This allows you to view database objects at levels below the level of the currently selected container. If no plus or minus sign appears next to a container it contains no objects extending beyond the level of that container.
- Press the plus or minus keys on the numeric keypad when the selected container has a plus or minus sign beside it. This expands or contracts the container.



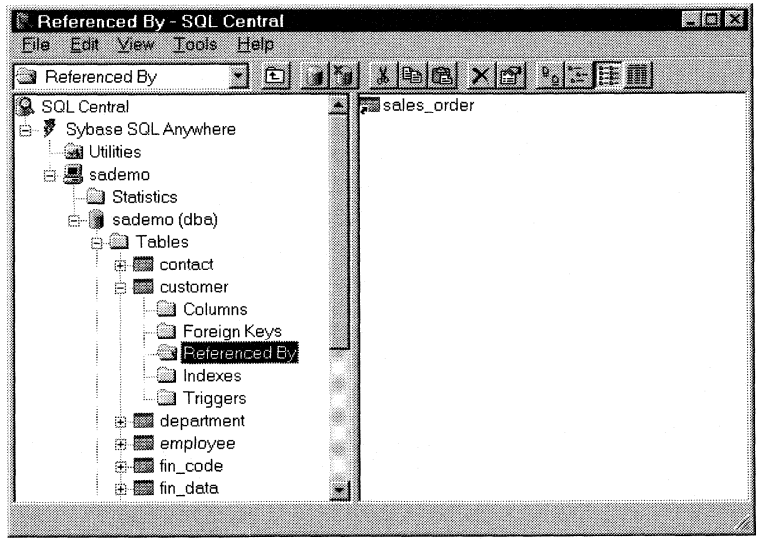
The left panel displays container objects only. The right panel displays the contents of the container object selected in the left panel.

Viewing the tables in a database

The following illustrates the steps taken to examine the contents of a table folder in a database container.

❖ **To examine the tables in a database:**

1. Select the Tables folder. You may have to expand the server and database objects in the hierarchical tree in the left panel in order to view the Tables folder.
2. Each table in the Tables folder is itself a container. Select a table in the left panel to reveal the contents of that table in the right panel. Each table object contains folders for columns, foreign keys, relations, indexes, and triggers.



3. Expand the table object in the left panel to reveal its contents in the hierarchical tree. Select each object in the table container by clicking on it once in the left panel. The right panel will display the contents of the table object selected in the left panel.

Viewing database object properties

The properties of database objects such as a database or stored procedures, can be viewed using any of the following methods. In all cases, a tabbed dialog box called a property sheet will be displayed, revealing all editable and non-editable properties.

- Using the right mouse button, click on a database schema. From the pop-up menu choose Properties.

- While holding the Alt key, double-click database schema in the right panel.
- With a database schema selected in the right panel, press the Alt and Enter keys simultaneously.
- With a database object selected in either the left or right panel, choose Properties from the File menu.

You can navigate a database by clicking or double-clicking in either panel. Explore the contents of the other folders in the database. Every SQL Anywhere database contains individual folders for the following:

- **Tables** Base tables stored in the database.
- **Views** Computed tables, stored in the database as a query and evaluated when accessed.
- **Procedures & Functions** for using a module-based language consisting of SQL procedures.
- **User & Groups** for administering who is permitted to use the database.
- **User-defined Data Types** for creating non-standard data types.
- **SQL Remote** for administering replication of data in the database.
- **DB Spaces** for creating more than one .DB file for the database.

You should explore the sample database until you are comfortable locating database objects in the Sybase Central main window.

Adding a table to a database

This tutorial takes you through the steps adding a table to the sample database. This task includes adding columns to an existing table. The tutorial covers the following interface skills:

- Using the table editor to add tables and columns
- Using property sheets to add new object properties
- Using drag-and-drop techniques to add new objects

To add a table to your database you define the name and properties of your table its columns using a tool called the table editor. The table editor displays the table columns and their properties in a spreadsheet-like grid. When these definitions are saved, the table is created.

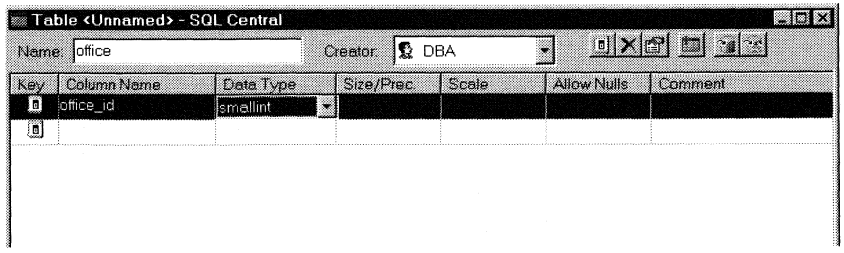
Create a table and adding columns in Sybase Central

Tables are added or edited using the table editor, which displays column properties in a grid: column properties are displayed along the top of the grid while the columns are identified by rows, one for every column in the table.

In this tutorial we create a table in the sample database. The new table will describe different offices for the sample database.

❖ To create a table and columns:

1. Select the Tables folder in the left panel. This can be done by first expanding the server and database containers, then clicking on the Tables folder.
2. Double-click the Add Tables object in the right panel, causing the Table Editor to appear.

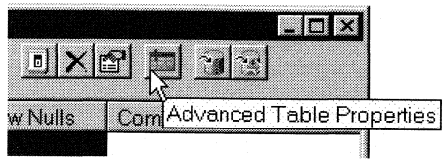


3 Type the new table name in the Name text box in the upper left corner of the table editor. In this case, use the name office.

4 Click in the Column Name column of the first row to enter the name of the first column in the table. Use the name office_id.

5 Under the Data Type column choose smallint from the drop down list of standard data types.

6 Do not make any entries under the Size/Precision, Scale and Comments columns. Ensure the entry under the Allow Nulls column is left unchecked so this column can later be used as a primary key.



7 Click the Advanced Table Properties button on the toolbar and enter the comment "Company offices" in the Comment text box. Comments are an optional property.

8 Finish creating the table by clicking the Save and Close toolbar button. The table editor disappears and the office table is added as another table in the Tables folder. It can be viewed in the right panel by selecting the Tables folder in the left panel.

The following section describes how to edit an existing column and add a primary key to the table using the table editor.

Editing existing tables using the table editor

This section describes how to edit an existing table. The following steps will add a primary key to a table created in the previous section named office. This will be accomplished by turning a column named office_id into a primary key. This column holds a numerical ID for each office.

❖ **To edit an existing table using the table editor:**

1 In the left panel, select the office table. You may have to expand the server, database and the Tables containers to view the office table.

2 Using the right mouse button, click on the selected office table to display the pop-up menu for the table object and select the Edit Columns menu item (the same menu item is also available under File menu on the menu bar). This causes the table editor to appear, displaying one row for every column in the table.

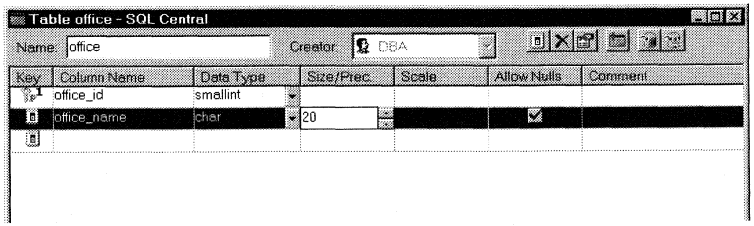
3 Click once on the office_id row, under the Key column. The icon changes to show the column named office_id is now also the primary key for the table named office.

> **Primary key conditions**
> If a check box appears under Allow Nulls column, or if duplicate values are stored in the office_id column, the column cannot be changed to a primary key.

4 Click the Save toolbar button to commit the changes to the office table. Pausing the mouse above a toolbar button causes a ToolTip to appear to more easily identify toolbar buttons.

5 Click the Add Column toolbar button to create another row in the table editor grid. This adds another column in the office table.

6 Edit the properties of the new column so that it's name is office_name, its type is char, its size is 20 and its value can be equal to null.



7 Click the Save and Close toolbar button to commit the changes and close the table editor.

The column is now present in the database, although it has no data. Sybase Central does not include facilities for data entry.

Dragging a column to a new table using Sybase Central

This section describes how to add a column to the office table created in a previous section to hold the address of each office. The customer table in the same sample database already has an address column, which can be copied and added to the office table. These steps will copy the attributes, including the name, of a column in one table to another table, but will not copy data from one table to another.

❖ **To drag a column to a table:**

1. In the left panel, click and expand the customer table to expose the Columns folder. You may have to expand the database container and the Tables folder to do this.
2. Select the Columns folder. The right panel will adjust to display the columns in the customer table.
3. Click the address column in the right panel, and drag it to the office table in the left panel. The Copy Column Definition window is displayed.
4. Click OK to add the column to the office table. It will have the same attributes as the address column of the customer table but will contain no data.
5. To see the new column, select and expand the office table in the left panel, then select the Columns folder. The right panel will display the columns of the office table. The newly-added address column should be visible in the right panel.
6. To see the attributes of the address column, double-click the address column in the right panel. The address properties sheet appears.
7. Change the name of the column to office_address by typing this in the Name text box. This modification applies to the address column of the office table only. Once a column has been copied there is no longer any connection between the original column and the new column.

Notes

- Drag and drop is available for several kinds of database management task. For example, you can add users to user groups by dragging, or drag a table to another table to create a foreign key relationship. For a complete list of drag-and-drop operations, look up drag and drop summary in the Sybase Central online help.

- Add Object icons are used to add other objects besides tables to a database. Stored procedures, triggers, indexes, users, user groups, and columns are among the database objects that you can add with an Add Object icon.

Deleting tables using Sybase Central

Tables can be deleted, or dropped, from a database. The office table created in a previous section can be dropped from the sample database, restoring the database to its original state.

❖ **To delete the office table from the sample database:**

1. Using the right mouse button, click the office table.
2. Select Delete from the pop-up menu.

Viewing and editing procedures

Stored procedures are kept in a folder within the database container object. This tutorial shows how to view and alter the contents of a procedure, and how to create new procedures using the Sybase Central code editor.

The Sybase Central code editor is a separate window for displaying and editing the code of triggers, procedures, and views.

Beyond text-editing functions, it provides:

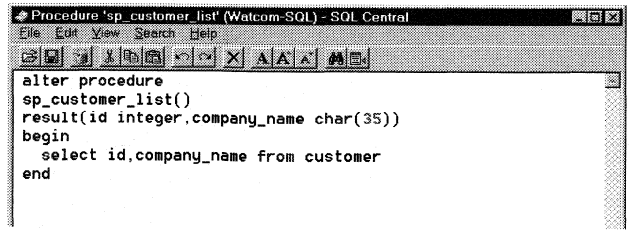
- Automatic syntax highlighting.
- Automatic formulation of DDL statements.
- Unlimited Undo and Redo.
- Ability to find and replace text, and to jump to specific line numbers.
- Ability to execute the DDL code against the database.

Viewing stored procedure code with Sybase Central

There are several stored procedures included in the sample database. The following steps explain how to view and edit stored procedures.

❖ To view the contents of a procedure:

1. Select the Procedures & Functions folder from the sample database in the left panel. You may have to expand the database container to do this.
2. In the right panel, double-click sp_customer_list in the right panel. The Sybase Central code editor appears, displaying the text of the procedure.



3. A call to this procedure returns a set of customer IDs and company names from the Customer table.

Viewing procedures in Watcom-SQL or Transact-SQL

SQL Anywhere supports two alternative syntaxes for stored procedures. The native SQL Anywhere syntax (Watcom-SQL) is based on the ISO draft standard. SQL Anywhere also supports the Sybase Transact-SQL syntax. You can enter procedures in either syntax, and SQL Anywhere can automatically translate between the two syntaxes.

Not all procedure statements may translate. Untranslated statements appear as comments in the translated procedure.

❖ **To view the alternative syntaxes of a procedure:**

1. With the right mouse button, click on the sp_customer_list procedure. A pop-up menu appears.
2. Click Open to view the procedure in the syntax in which it was entered. This is the syntax in which the procedure is stored in the database.
3. Click Open as Watcom-SQL to view the procedure in Watcom-SQL syntax.
4. Click Open as Transact-SQL to view the procedure in Transact-SQL syntax.

Notes

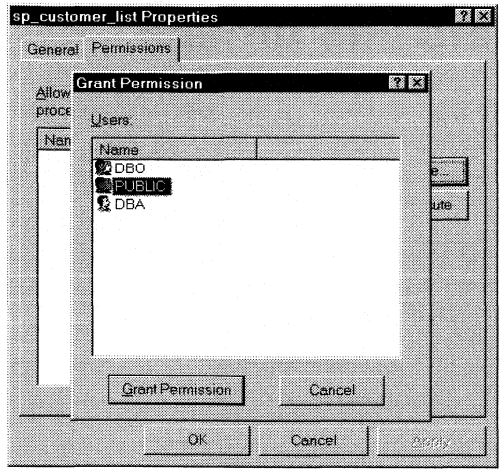
- The Sybase Central code editor is used to display and create views and triggers as well as stored procedures.

Setting permissions on procedures

Stored procedures have permissions associated with them. In order to execute a procedure you either need to be granted permission to execute a procedure, or you need to be a member of a user group that has permission to execute the procedure.

❖ **To view and alter the permissions on a procedure:**

1. Using the right mouse button, click on the sp_customer_list procedure. A pop-up menu appears.
2. Select Properties from this menu. The properties sheet for sp_customer_list appears.
3. Click the Permissions tab to see which user IDs have been granted permissions on this procedure. Currently none do, as the only user for the sample database is DBA, who automatically has execute permissions as owner of the procedure.



4 Click the Grant Permissions button tab to grant users or groups permission to execute this procedure. Grant permission to execute this procedure to the Public user group by selecting the Public user group icon and clicking the Grant Permission button.

5 Click the OK button to accept the changes to the sp_customer_list permissions or Cancel to undo the changes.

Managing users and groups

In SQL Anywhere, both users and groups are objects within the database. However, groups are also containers, capable of containing users (making users a member of that group) and other groups.

In this structure, permissions granted to a group are inherited by those users and groups which have membership in the group. SQL Anywhere allows you to create users and groups permitted to use a database and grant membership to groups.

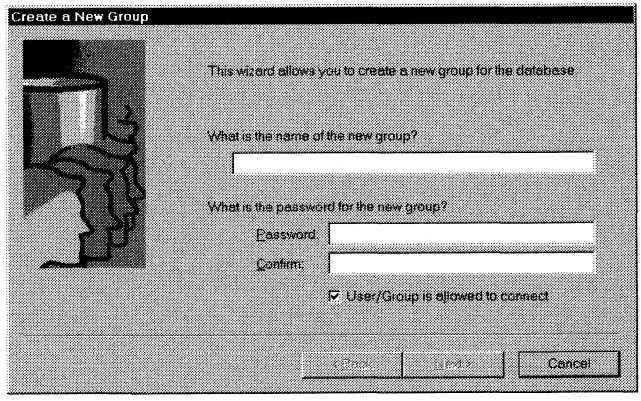
This tutorial demonstrates how to create a group for the database, create an individual user, and make the user a member of the group.

Adding a group to the database

In this section we add a sales group to the sample database.

❖ **To add a group to a database:**

1. Select the Users & Groups folder in the left panel. You may need to expand the sample database container in the left panel to do this.
2. Double-click Add Group in the right panel. The New Group wizard appears.



3. Type the name Sales in the top text box. This is the name of the user group.
4. Enter a password (for example, Sales), confirm it by entering it again, and click Next to show the following page of the wizard.

5 Check the Resource authorities option, and uncheck the DBA and Remote DBA option, for this group. Then click Next to show the following page of the wizard.

6 Click Finish to create the new group.

The sales group appears in both panels. Groups are container objects, and so appear in the left panel.

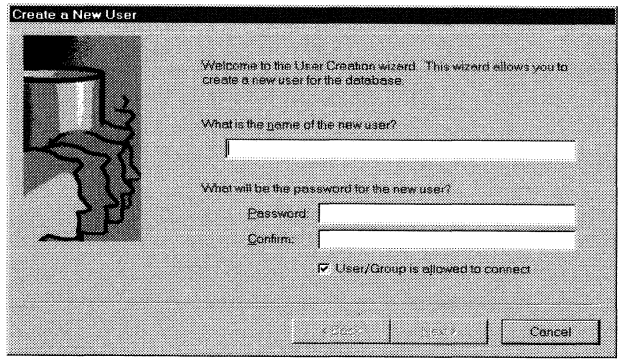
Adding a user to the database

This section describes how to add a user to a database.

❖ **To add a new user to the sample database:**

1 Select the Users & Groups folder in the left panel. You may need to expand the sample database container in the left panel to do this.

2 Double-click Add User in the right panel. The new user wizard appears.



3 Type the name Sandy in the top text box. This is the user ID for the new user.

4 Type a password, and confirm it by retyping it. For example, you could use the password Sandy. Click OK to confirm the entries.

5 Follow the instructions to complete the wizard choosing the default options.

6 An icon appears in the right panel, showing the new user. There is no icon in the left panel, as individual user ID's are not containers.

Adding a user to a group

Sybase Central provides two ways to add a user to a group. You can use the individual user's Properties sheet, which has a Membership tab. Or you can use a drag and drop method.

In this section we add two users to a group using drag and drop.

❖ **To add users to a group:**

1. Ensure that the Sales group is shown in the left panel and that the users DBA and Sandy are shown in the right panel. You may need to select the Users & Groups folder to do this.
2. Click Sandy to select this user.
3. While holding down the Ctrl key, click DBA to select this user as well.
4. Click and drag the users to the Sales group in the left or right panel.
5. Select the Sales group in the left panel to show its members.

Restoring the sample database to its original state

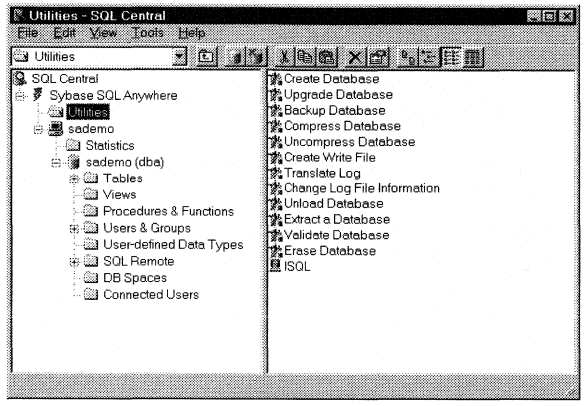
To restore the database to its original state, you can delete the Sales group and the user Sandy. For each of the two icons:

1. Using the right mouse button, click the icon.
2. Select Delete from the pop-up menu.

Backing up a database using Sybase Central

Sybase Central includes a set of database utilities for carrying out common database administration tasks. Wizards are provided to help carry out the task step-by-step.

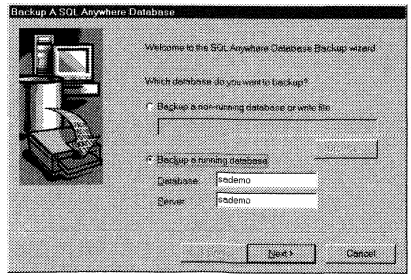
To see all the database utilities provided with Sybase Central, click the Database Utilities folder at the top of the left panel. A list of the utilities appears in the right panel.



Some of the utilities are used on database files, while others can be used with running databases. Those utilities that can be used on a running database are shown on a pop-up menu by right mouse clicking on the database icon. In this lesson, we back up the sample database.

❖ To back up a running database

1 Using the right mouse button, click the sample database and select Backup from the pop-up menu. The Backup Wizard appears.



2 You do not need to alter anything on the first page of the wizard. Click Next to move to the second page.

3 You should already have the user ID DBA filled in, together with the password. Click Next to move to the next page of the wizard.

4 Type a directory in the text box indicating where you wish to back up the database to. As this is a tutorial only, you may wish to choose a temporary file directory such as C:\TEMP.

5 Check the Main Database File and Transaction Log File options, and uncheck the Database Write File option. Then click Next to take you to the next page of the Wizard.

6 Select the bottom option button from the three options presented, and click Next to take you to the next page of the Wizard.

7 Review the choices you have made, and click Finish to back up the database. A window displays the progress of the backup.

Wizards are available for several other database administration tasks. You may wish to try creating a database by selecting the Database Utilities folder in the left panel and then double-clicking the Create Database tool in the right panel.

Using the Sybase Central online help

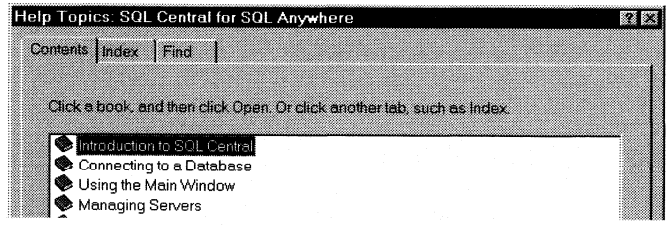
The main documentation for Sybase Central is available as online help separate from the SQL Anywhere User's Manual. This tutorial introduces you to the Sybase Central online help.

The online help is presented as a set of **topics**. You can find topics about a subject of interest using the Contents, the Index, or by searching the text of the online Help.

Using the online help Contents

❖ **To see the Sybase Central online help contents:**

1. Select Help Topics from the Sybase Central Help menu. The Sybase Central online help Help Topics window appears.
2. The Help Topics window has three tabs: Contents, Index, and Find. Click the Contents tab.



The online help Contents are displayed as a set of books. To open a book, click the book and click Open. The following example illustrates the organization of the help topics.

❖ **To find out how to add a group:**

1. With the Contents tab showing on the Help Topics window, click Managing Users & Groups to open the book.
2. Click Creating A New User Or Group to display the topic.

Notes

- Many topics have a Related Topics link at the end of the topic, which can take you to related topics in the Sybase Central online help and in the online version of this book.
- You can use the Browse buttons (with arrows) to take you back and forward through related topics.

- You can configure some aspects of the appearance of the online help by clicking Options.

You should spend some time browsing through the Contents window to familiarize yourself with the online help organization.

Using the online help Index

The Index provides an alternative way to search for information in the online Help. This section shows how to find information about creating users using the index.

❖ **To find a topic using the Index:**

1. Open the Help Topics window, and click the Index tab.
2. Type "users" in the box numbered 1. In box 2, the Users & Groups index entry is highlighted.
3. The index is a two-level index. Click Creating under Users & Groups, and click Display to show the topic.

Notes

If there is more than one topic indexed under the entry you display, a list of topics is displayed.

Searching the text of the online help

If you cannot find the information you are looking for using the Contents or the Index, you may want to try searching the text of the online Help.

❖ **To find a topic by searching text:**

1. Open the Help Topics window, and click the Find tab.
2. When you have built your word list, type "user" in the box numbered 1. As you type, a list of matching words is shown in the box numbered 2.
3. Click user in box 2.
4. Click Creating A New User Or Group in the topic list in box number 3, and click Display to show the topic.

CHAPTER 5

Using ISQL

About this chapter

This chapter discusses how to run ISQL (Interactive SQL) under Microsoft Windows, Microsoft Windows NT, or IBM OS/2. Ideally, you should run the software on your computer as you work through this chapter.

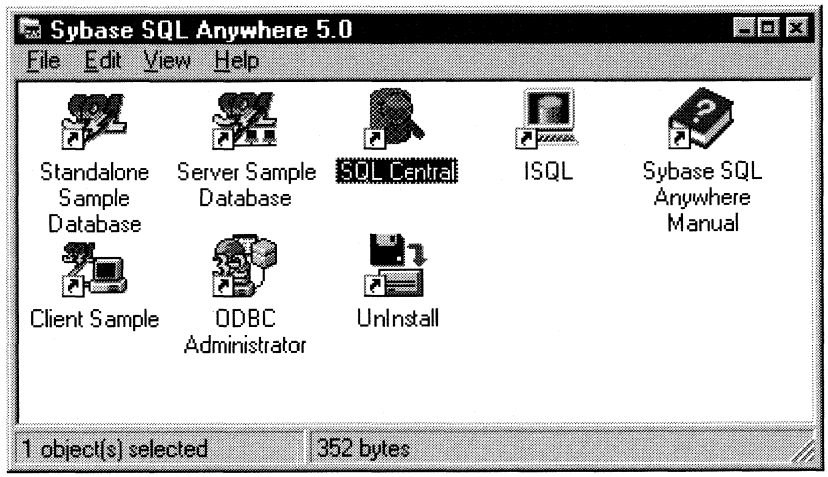
This chapter presents the various facilities offered by the ISQL environment. The chapter "Selecting Data from Database Tables" starts an in-depth tutorial of the SQL language and the SQL Anywhere database engine.

Contents

Topic	Page
The SQL Anywhere program group	68
Starting SQL Anywhere	69
Connecting to the sample database from ISQL	70
Accessing Help from ISQL	71
The ISQL command window	72
Leaving ISQL	73
Displaying data in ISQL	74
Command recall in ISQL	76
Function keys	78
Canceling an ISQL command	79
What's next?	80

The SQL Anywhere program group

After the software is installed, you will have a SQL Anywhere program group, containing icons for the components of the SQL Anywhere software (the SQL Anywhere client installation gives a subset of these icons). The program group for Windows 95 is displayed in the figure:

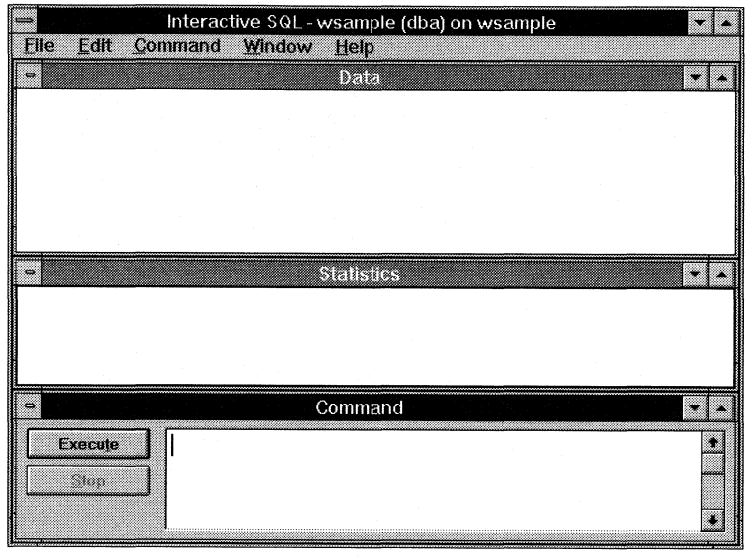


Starting SQL Anywhere

Start the database engine on the sample database by double clicking on the SQL Anywhere icon labeled Standalone Sample Database.

The SQL Anywhere window appears displaying some startup information. After a few seconds, the window automatically reduces to an icon on the bottom of the screen.

Once the database engine is running, start ISQL by double clicking on the ISQL icon. The ISQL window appears.

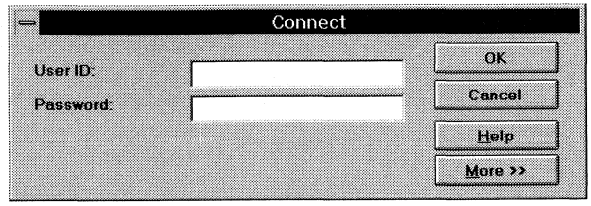


Connecting to the sample database from ISQL

When you start ISQL, it is not connected to any database. When a database is running, ISQL displays the SQL Anywhere Log On window.

1 You must enter a number of connection parameters in the Connection window.

- **User ID** For the sample database, use the user ID dba.
- **Password** The password for the dba user ID is sql.



The password does not appear when you type it. This prevents someone else from seeing your password

- The connect dialog contains a button labeled More>>. Click More>> to reveal a larger dialog box which contains more options for connecting to databases. These options are not needed for connecting to a database when only one database is running. (For the purposes of this tutorial, the options revealed by clicking More are not needed).

2 After entering the user ID and password, press ENTER (or click OK) to connect to the sample database.

If you have made typing mistakes or the sample database is not found, an error message appears. You can use the tab key to move to the field in error and correct the problem using the cursor keys and the backspace key.

After connecting to the database, ISQL displays the database name, user ID, and server name for the connection on the title bar.

Accessing Help from ISQL

Help is available by pressing the F1 key, or by choosing ***Help➤Help*** from the menu bar. The Help system can also be activated with the HELP command. The Help files contain help on many topics—most of this manual is contained in the Help files.

For more information on using Help, choose ***Help➤Using Help*** from the menu bar.

The ISQL command window

The **Command window** appears at the bottom of the ISQL screen. It is a standard edit control for typing ISQL commands. If more lines are typed than will fit in this window, the window automatically scrolls. You can scroll the window using the cursor keys or the scroll bar on the right side of the window. This window can also be made larger and maximized to full screen size in the standard Windows or OS/2 fashion.

Commands are executed by pressing the execute key (F9) or you can click the Execute button.

Multiple commands can be entered at once by separating them with semicolons. Commands can be stored to an ASCII file or loaded from an ASCII file by choosing File➤Save or Open from the menu bar.

Leaving ISQL

When you have finished working with ISQL, the EXIT command returns you to the operating system (or choose File➤Exit from the menu bar). You can then stop the database engine by clicking on the SQL Anywhere icon and selecting the Close menu item.

If you leave ISQL now, you will have to restart ISQL to continue with the tutorial.

Displaying data in ISQL

One of the principal uses of ISQL is to look at information in databases.

The database used in this tutorial is for a fictional company. The sample database contains information about employees, departments, sales orders, and so on.

All this information is organized into a number of **tables**, consisting of **rows** and **columns**.

You display information from a database using the SELECT statement. The following example shows the command to type in the ISQL command window. Once you have typed the command, you must click Execute to carry out the command. The example displays the first several columns and rows of the results of the query, which are displayed in the ISQL data window. The format is used throughout this manual.

❖ **To list all the columns and rows of the employee table:**

- Type the following:

```
SELECT *
FROM employee
```

emp_id	manager_id	emp_lname	emp_fname	...
102	501	Fran	Whitney	...
105	501	Matthew	Cobb	...
129	902	Philip	Chin	...
148	1293	Julie	Jordan	...
160	501	Robert	Breault	...
...				

Notes

- SQL statements are case insensitive. SELECT is the same as select is the same as Select. In the examples, SQL keywords are shown in upper case, but you do not have to type them in upper case.
- SQL statements can be typed on more than one line. You can type the statements all on one line, or break them over several lines as you wish. Some SQL statements, such as the SELECT statement, consist of several parts, called clauses. In many examples, each clause is placed on a separate line, but you do not have to type them this way.

The ISQL Data window displays a set of rows and columns containing information about the employees. Each row contains information about one employee, and each column contains a particular piece of information for all employees.

Scrolling the data window

When you type the command

```
SELECT * FROM employee
```

in the ISQL command window, the visible portion of the ISQL data window cannot hold the entire employee table.

The visible portion of the data window does not display all the information about each employee, and does not display the entire list of employees.

Viewing other columns

To see more information about each employee (that is, other columns) you use the scroll bar at the bottom of the data window. This is a standard Windows or OS/2 scroll bar.

Viewing other rows

To see more information on other employees (that is, other rows), use the scroll bar to the right of the data window. The employee table in the sample database has information on about 75 employees.

Sometimes, the vertical scroll bar behaves slightly differently than standard scroll bars, as the number of rows in the result may be unknown. In this case, a guess as to the number of rows is used. If ISQL determines that its guess is wrong, the guess is adjusted and the slider "jumps".

Command recall in ISQL

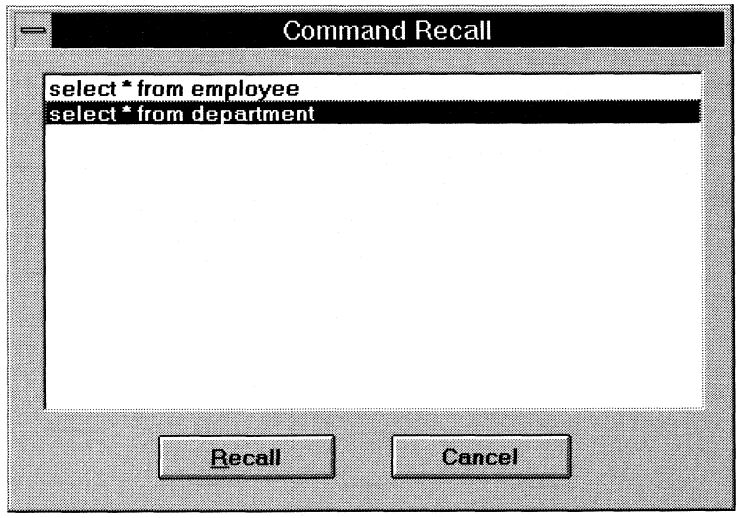
Let's execute another command.

1 Type the following:

```
SELECT * FROM department
```

2 Press F9.

The contents of the department database table are displayed in the Data window. As you execute commands with ISQL, they are saved in a command history. To recall commands, choose Command➤Recall from the menu bar. This activates the command recall window.



The command recall window displays the first line of the last 15 commands executed. Use the cursor up and down keys to scroll through the commands.

Position the cursor on the first command that you executed, which was:

```
SELECT *
FROM employee
```

and press the ENTER key. The cursor returns to the command window with the selected command in it. You can now re-execute that command or modify it to make a new command.

More recall keys

The following keys can also be used to recall previous commands:

Key sequence	Description
CTRL+R	Brings up the command recall window
CTRL+P	Cycles backwards through previously executed commands. Retrieved commands are placed into the command window
CTRL+N	Cycles forward through previously executed commands

Function keys

ISQL uses some function keys and special keys as follows:

Function key	Description
F1	Help
F5	Move data to the left by one column in the data window
SHIFT+F5	Move data to the left by one character
F6	Move data to the right by one column
SHIFT+F6	Move data to the right by one character
F7	Display a list of the tables in the database. The cursor up and down keys can be used to scroll through the table names changing the highlighted table name. With the list displayed, pressing ENTER will insert the current table name into the command window at the cursor position. The F7 key can be used while the table list is displayed, and a list of columns will be displayed for the highlighted table. Again, ENTER can be used to select the highlighted column name and put it into the command window at the cursor position.
F9	Execute the command that is in the command window. This operation can also be performed with the mouse by clicking Execute.
F10	Activate the menus
PGUP	Move data up a page
PGDN	Move data down a page
CTRL+PGUP	Move to top of data
CTRL+PGDN	Move to bottom of data

Canceling an ISQL command

The Stop button is used to cancel a command.

A Stop operation stops current processing and prompts for the next command. If a command file was being processed, you are prompted for an action to take (Stop command file, Continue, or Exit ISQL). These actions can be controlled with the ISQL ON_ERROR option (see "SET OPTION statement" in the chapter "Watcom-SQL Statements").

Reported errors

When an abort is detected, one of three different errors will be reported depending upon when the abort is detected.

1 If the abort is detected when ISQL is processing the request (as opposed to the database engine), then the following message is displayed:

```
ISQL command terminated by user
```

stops processing immediately and the database transaction is left alone.

2 If the abort is detected while the database engine is processing a data definition command (CREATE, DROP, ALTER, etc.), the following message appears:

```
Terminated by user -- transaction rolled back
```

Since data definition commands all perform a COMMIT automatically before the command starts, the effect of the ROLLBACK is to just cancel the current command.

This message also occurs when the database engine is running in bulk operations mode executing a command that modifies the database (INSERT, UPDATE, and DELETE). In this case, ROLLBACK cancels not only the current command, but everything that has been done since the last COMMIT. In some cases, it may take a considerable amount of time for the database engine to perform the automatic ROLLBACK.

3 If the cancel is detected by the database engine while processing a standard data manipulation command (SELECT, INSERT, DELETE, etc.) and the engine is not running in bulk operations mode, then the following message is displayed.

```
Statement interrupted by user.
```

The effects of the current command are undone, but the rest of the transaction is left intact.

What's next?

Once you are familiar with the basics of ISQL, you should proceed to the chapter "Selecting Data from Database Tables" for a tutorial on the SQL language.

CHAPTER 6

Using Character-Mode ISQL

About this chapter

This chapter discusses how to run ISQL (Interactive SQL) under those operating systems for which a character-mode ISQL is provided: DOS, QNX, or NetWare.

You should actually run the software on your computer as you work through this chapter. This chapter presents the various facilities offered by the ISQL environment.

Contents

Topic	Page
Tutorial files	82
Starting the SQL Anywhere software	83
Connecting to the sample database from ISQL	84
ISQL menu selection	85
Obtaining help from ISQL	86
The ISQL command window	87
Leaving ISQL	88
Displaying data in ISQL	89
Command window keys in ISQL	91
Scrolling the data window	93
Command recall in ISQL	94
Function keys	95
Aborting an ISQL command	96
What next?	97

Tutorial files

DOS: First, you should be in the subdirectory containing the SQL Anywhere software. This directory is chosen at installation time (usually C:\SQLANY50). The commands

```
c:
cd \sqlany50\dos
```

will get you there. The sample database, SADEMO.DB, will be located in the parent directory (C:\SQLANY50).

Alternatively, you may want to copy the sample database used in this tutorial into a different working directory. The only file you need to copy is: SADEMO.DB. In this case, you need to have SQL Anywhere in your PATH (this is done for you at installation time).

QNX: copy or link the sample database to a working directory
You should copy the sample database to your working directory or create a link to it. For example,

```
cp /usr/lib/sqlany50/sample/sademo.db .
```

or

```
ln /usr/lib/sqlany50/sample/sademo.db .
```

Starting the SQL Anywhere software

To start ISQL in DOS or QNX, type the command:

```
isql
```

To start ISQL on a NetWare machine, type the command:

```
load isql.nlm
```

This command loads ISQL.

Connecting to the sample database from ISQL

ISQL initially is not connected to any database. To connect, type the command

```
connect
```

and press the Execute key.

ISQL prompts you for a number of connection parameters. These include a user identification (**user ID**), **password** and **database file** Type the user ID DBA, then press the TAB key to move to the next field.

Type in the password SQL. The password does not appear when you type it. This prevents someone else from seeing your password.

Press the TAB key again to move to the database file field and enter SADEMO.DB. This is the sample database that you will use in the tutorial.

Now press the ENTER key (or click the OK button using a mouse) to connect to the database. If you have made typing mistakes or the sample database is not found, an error message will appear. You can use the tab key to move to the field in error and correct the problem using the cursor keys and the backspace key.

If you have typed everything correctly, the problem could be that the database SADEMO.DB is not in your current directory. You should return to the beginning of this chapter, making sure that the software has been installed correctly and that you have followed the instructions in "Tutorial files" on page 82.

If you successfully connect to the database, the statistics window displays the message Connected to database.

ISQL menu selection

Throughout this book, you will be instructed to **choose** items from pull-down menus. The pull-down menus are located at the top of the screen.

To illustrate their use, we will choose Help from the Help menu. :option.

❖ **To choose a menu item using the keyboard:**

1 Press the ALT key.

2 Press the highlighted letter in the name of the menu (H). The Help menu will "pull down".

3 Press the highlighted letter in the name of the item (H).

❖ **To choose a menu item using the mouse:**

1 Position the mouse on the name of a pull-down menu (Help).

2 Press and release (click) the left mouse button. The Help menu will "pull down".

3 Position the mouse on the correct item and click the left mouse button again.

To leave the Help system, press the ESCAPE key, or click on the Cancel button.

Obtaining help from ISQL

Help is available by pressing the F1 key, or by choosing Help from the Help menu. The help system can also be activated with the HELP command. The help files contain help on many topics — most of this manual is contained in the help files.

The ISQL command window

The **command window** appears at the bottom of the screen. It has four lines for typing ISQL commands. If more than 4 lines are typed, the window scrolls automatically. You can scroll the window using the cursor keys or the scroll bar on the right side of the window. This window can also be made larger and zoomed to full screen size by choosing the appropriate item from the Command menu.

Grow Window (CTRL+G) to grow window one line up to a maximum of half the screen.

Shrink Window (CTRL+S) to shrink window one line down to a minimum of 3 lines.

Zoom Window (CTRL+Z) to zoom window to full screen and back again.

Leaving ISQL

When you have finished working with ISQL, the EXIT command returns you to the operating system (or choose Exit from the File menu). Since ISQL started the database engine, the database engine is automatically stopped by ISQL on exit. If you exit from ISQL, you will have to start ISQL again to continue with the tutorial.

Displaying data in ISQL

One of the principal uses of ISQL is to look at information in databases.

The database used in this tutorial is for a fictional company. The sample database contains information about employees, departments, sales orders, and so on.

All this information is organized into a number of **tables**, consisting of **rows** and **columns**.

You display information from a database using the SELECT statement. The following example shows the command to type in the ISQL command window. Once you have typed the command, you must click Execute to carry out the command. The example displays the first several columns and rows of the results of the query, which are displayed in the ISQL data window. The format is used throughout this manual.

❖ **To list all the columns and rows of the employee table:**

- Type the following statement:

```
SELECT *cd2
FROM employee
```

emp_id	manager_id	emp_lname	emp_fname	...
102	501	Fran	Whitney	...
105	501	Matthew	Cobb	...
129	902	Philip	Chin	...
148	1293	Julie	Jordan	...
160	501	Robert	Breault	...
...				

Notes

- SQL statements are case insensitive. SELECT is the same as select is the same as Select. In the examples, SQL keywords are shown in upper case, but you do not have to type them in upper case.
- SQL statements can be typed on more than one line. You can type the statements all on one line, or break them over several lines as you wish. Some SQL statements, such as the SELECT statement, consist of several parts, called **clauses** In many examples, each clause is placed on a separate line, but you do not have to type them this way.

The ISQL Data window displays a set of rows and columns containing information about the employees. Each row contains information about one employee, and each column contains a particular piece of information for all employees.

Command window keys in ISQL

If you make a spelling mistake, you can correct it using the following keys:

CURSOR LEFT moves the cursor left over top of any characters already typed.

CURSOR RIGHT moves the cursor right.

HOME moves the cursor to the start of the line.

END moves the cursor past the last character of the line.

INS toggles **Insert mode**. When **Insert mode** is on, the cursor is different (usually fatter), and any characters typed will be inserted where the cursor is, pushing characters to the right of the cursor further to the right.

DEL deletes the character at the cursor, and moves the rest of the line back to fill the vacant position.

BACKSPACE deletes the character immediately to the left of the cursor. If this key is pressed at the beginning of a line, it will **join** the current line to the previous one.

CTRL+END erases from the cursor position to the end of the line.

ENTER moves the cursor to the beginning of the next line.

CURSOR DOWN moves the cursor down one line. If you are on the last line of the command window and there are more lines below the window, the text will scroll up one line so that you can see the next line.

CURSOR UP moves the cursor up one line. If you are on the first line of the command window and there are more lines above, the text will scroll down one line so that you can see the previous line.

ESC clears the entire command from the command window.

SHIFT+CURSORUP, SHIFT+CURSORDOWN, SHIFT+PGUP, SHIFT+PGDN used to mark an area (set of lines). This can also be done by pressing the left mouse button and holding it down while you drag the mouse over the lines to be marked.

CTRL+X cuts the marked area to the edit clipboard. The lines are removed from the current position and can be pasted back anywhere using the Ctrl+V key combination.

CTRL+V pastes the contents of the edit clipboard into position following the current line.

CTRL+C copies the marked area to the edit clipboard. This is different than the CTRL+X key combination in that it does not delete the marked area. There are also items in the Edit menu that can be used instead of these key combinations.

The following keys are also useful shortcut keys, but they will only work on newer keyboards.

ALT+INS insert a new line after the current line.

ALT+DEL delete the current line.

Commands are executed by pressing the execute key (F9). The right mouse button also executes the current command.

Multiple commands can be entered at once by separating them with semicolons. Commands can be stored to an ASCII file or loaded from an ASCII file by choosing Save or Open from the File menu.

Scrolling the data window

When you type the command

```
SELECT * FROM employee
```

in the ISQL command window, the visible portion of the ISQL data window cannot hold the entire employee table.

The visible portion of the data window does not display all the information about each employee, and does not display the entire list of employees.

To see more information about each employee (that is, other columns) you use the move left and move right operations. These operations can be done by choosing Left and Right from the Data menu. If your computer has a mouse, you can also move right by using the scroll bar at the bottom of the data window.

The employee table in the sample database has information on 75 employees. To see more information on other employees (that is, other rows), use the PAGE DOWN and PAGE UP keys. Try pressing the PAGE DOWN key, and observe that a new screen full of employees is shown. Continue pressing PAGE DOWN. and eventually the screen contains only the column names with no rows. Pressing PAGE UP. at this point will bring back the last screen full of employees in the employee table.

If your computer has a mouse, you can also move the information up and down using the scroll bar on the right side of the screen. Sometimes, this scroll bar behaves slightly differently than standard scroll bars when the number of rows in the result is unknown. In this case, a guess as to the number of rows is used. When ISQL determines that the guess is wrong, it will be adjusted and the slider "jumps".

The Data menu contains items for moving up and down a line. It also has items for going to the top and bottom of the query. These actions can also be performed with CTRL+PAGE DOWN and CTRL+PAGE UP. CTRL + PAGEG DOWN is actually two keys; hold down the CTRL key and press the PAGE DOWN key. When you do this, ISQL displays the last screen of employees in the employee table. Similarly, CTRL+PAGE UP will display the first screen of employees. You can move back to the original position by using CTRL+PAGE UP and then pressing the Left Arrow key until the emp_id column appears.

Command recall in ISQL

Let's execute another command. Type the following and then press &doit.:

```
SELECT *
FROM department
```

The contents of the department database table are displayed in the Data window. As you execute commands with ISQL, they are saved in a command history. To recall commands, choose Recall from the Command menu. This activates the command recall window. The window displays the first line of the last 15 commands that you have executed. Use the cursor up and down keys to scroll through the commands. As you do so, the full command appears in the command window at the bottom of the screen.

Position the cursor on the first command that you executed, which was:

```
SELECT *
FROM employee
```

Now press the ENTER key. The cursor will return to the command window with that command in it. You can now re-execute that command or you can modify it to make a new command.

The following keys can also be used to recall previous commands:

CTRL+R bring up the command recall window.

CTRL+P cycles backwards through previously executed commands. Retrieved commands are placed into the command window.

CTRL+N cycles forward through previously executed commands.

Function keys

ISQL uses some function keys and special keys as follows:

F1 Help.

F5 Move data to the left by one column in the data window.

Shift+F5 Move data to the left by one character.

F6 Move data to the right by one column.

Shift+F6 Move data to the right by one character.

F7 Display a list of the tables in the database. The cursor up and down keys can be used to scroll through the table names changing the highlighted table name. With the list displayed, pressing &enter. will insert the current table name into the command window at the cursor position. The :key.F7:ekey. key can be used while the table list is displayed, and a list of columns will be displayed for the highlighted table. Again, &enter. can be used to select the highlighted column name and put it into the command window at the cursor position.

F9 Execute the command that is in the command window. This operation can also be performed with the mouse by pressing the right mouse button or clicking Execute.

F10 Activate the menus.

PgUp Move data up a page.

PgDn Move data down a page.

Ctrl+PgUp Move to top of data.

Ctrl+PgDn Move to bottom of data.

Aborting an ISQL command

The CTRL+BREAK combination is used to abort a command.

A Stop operation stops current processing and prompts for the next command. If a command file was being processed, you are prompted for an action to take (Stop command file, Continue, or Exit ISQL). These actions can be controlled with the ISQL ON_ERROR option.

For more information on ISQL options, see "SET OPTION statement" in the chapter "Watcom-SQL Statements".

When an abort is detected, one of three different errors will be reported depending upon when the abort is detected.

1 If the abort is detected when ISQL is processing the request (as opposed to the database engine), then the following message is displayed:

```
ISQL command terminated by user
```

2 ISQL stops processing immediately and the database transaction is left alone.

3 If the abort is detected while the database engine is processing a data definition command (CREATE, DROP, ALTER, etc.), the following message is displayed: Terminated by user—transaction rolled back Since data definition commands all perform a COMMIT automatically before the command starts, the effect of the ROLLBACK is to just cancel the current command.

4 This message also occurs when the database engine is running in bulk operations mode executing a command that modifies the database (INSERT, UPDATE, and DELETE). In this case, ROLLBACK cancels not only the current command, but everything that has been done since the last COMMIT. In some cases, it may take a considerable amount of time for the database engine to perform the automatic ROLLBACK.

5 If the abort is detected by the database engine while processing a standard data manipulation command (SELECT, INSERT, DELETE, etc.) and the engine is not running in bulk operations mode, then the following message is displayed. Statement interrupted by user The effects of the current command are undone, but the rest of the transaction is left intact.

What next?

Once you are familiar with the basics of ISQL, you should proceed to "Selecting Data from Database Tables" for a tutorial on the SQL language.

CHAPTER 7

Selecting Data from Database Tables

About this chapter

This tutorial introduces the SELECT statement, the statement you use to retrieve information from databases. SELECT statements are commonly called **queries**, because they ask the database engine about the information in a database.

Contents

Topic	Page
Looking at the information in a table	101
Ordering query results	103
Selecting columns from a table	104
Selecting rows from a table	105
Comparing dates in queries	106
Compound search conditions in the WHERE clause	107
Pattern matching in search conditions	108
Matching rows by sound	109
Short cuts for typing search conditions	110

Before you begin the tutorial

The SELECT statement is a versatile command. SELECT statements can become highly complex in applications retrieving very specific information from large databases. This tutorial uses only simple SELECT statements: more advanced queries are described in later tutorials.

For more information about the full syntax of the select statement, see "SELECT statement" in the chapter "Watcom-SQL Statements".

Notes

Ideally, you should be running the SQL Anywhere software on your computer while you read and work through the tutorial lessons.

Each lesson instructs you to type commands into the computer and describes what you will see on your computer screen. If you cannot run the software as you read the tutorials, you will still be able to learn about SQL but you will not have the opportunity to experiment on your own.

This tutorial assumes that you have already started ISQL and connected to the sample database. If you have not already done so, see the chapter "Using ISQL" or "Using Character-Mode ISQL".

Looking at the information in a table

The database you use in this tutorial is for a fictional company. The database contains information about employees, departments, sales orders, and so on. All the information is organized into a number of **tables**.

Using the SELECT statement

In this lesson, you look at one of the tables in the database. The command used will look at everything in a table called **employee**. Execute the command:

```
SELECT * FROM Employee
```

Case sensitivity

The table name **employee** is shown starting with an upper case E, even though the real table name is all lower case. SQL Anywhere databases can be created as case-sensitive or case-sensitive in their string comparisons, but are always case insensitive in their use of identifiers. For information on creating databases, see the chapter "Initializing a database" in the chapter "Working with Database Objects", or the description of the DBINIT command in the chapter "SQL Anywhere Components".

You can type **select** or **Select** instead of **SELECT**. SQL Anywhere allows you to type keywords in uppercase, lowercase, or any combination of the two. In this manual, uppercase letters are generally used for SQL keywords.

The SELECT statement retrieves all the rows and columns of the **employee** table, and the ISQL Data window lists those that will fit:

emp_id	manager_id	emp_fname	emp_lname	dept_id
102	501	Fran	Whitney	100
105	501	Matthew	Cobb	100
129	902	Philip	Chin	200
148	1293	Julie	Jordan	300
160	501	Robert	Breault	100

You will also see some information in the ISQL statistics window. This information is explained later.

The **employee** table contains a number of **rows** organized into **columns**. Each column has a name, such as **emp_lname** or **emp_id**. There is a row for each employee of the company, and each row has a value in each column. For example, the employee with employee ID 102 is Fran Whitney, whose manager is employee ID 501.

Manipulation of the ISQL environment is specific to the operating system you are running in.

For information on how to scroll the data and manipulate the ISQL environment, see the chapter "Using ISQL" or "Using Character-Mode ISQL".

Ordering query results

Unless otherwise requested, SQL Anywhere displays the rows of a table in no particular order. Often it is useful to look at the rows in a table in a more meaningful sequence. For example, you might like to see employees in alphabetical order.

The following example shows how adding an ORDER BY clause to the SELECT statement causes the results to be retrieved in alphabetical order.

❖ **To list the employees in alphabetical order:**

- Type the following:

```
SELECT * FROM employee ORDER BY emp_lname
```

emp_id	manager_id	emp_fname	emp_lname	dept_id
1751	1576	Alex	Ahmed	400
1013	703	Joseph	Barker	500
591	1576	Irene	Barletta	400
191	703	Jeannette	Bertrand	500
1336	1293	Janet	Bigelow	300

Notes

The order of the clauses is important. The ORDER BY clause must follow the FROM clause and the SELECT clause.

Selecting columns from a table

Often, you are only interested in some of the columns in a table. For example, to make up birthday cards for employees you might want to see the **emp_lname**, **dept_id**, and **birth_date** columns.

❖ **List the last name, department, and birthdate of each employee:**

♦ Type the following:

```
SELECT emp_lname, dept_id, birth_date
FROM employee
```

emp_lname	dept_id	birth_date
Whitney	100	1958-06-05
Cobb	100	1960-12-04
Chin	200	1966-10-30
Jordan	300	1951-12-13
Breault	100	1947-05-13

Rearranging columns

The three columns appear in the order in which you typed them in the SELECT command. If you want to rearrange the columns, simply change the order of the column names in the command. For example, to put the **birth_date** column on the left, use the following command:

```
SELECT birth_date, emp_lname , dept_id
FROM employee
```

Ordering rows

You can order rows and look at only certain columns at the same time as follows:

```
SELECT birth_date, emp_lname , dept_id
FROM employee
ORDER BY emp_lname
```

As you might have guessed, the asterisk in

```
SELECT * FROM employee
```

is a short form for all columns in the table.

Selecting rows from a table

Sometimes you will not want to see information on all the employees in the **employee** table. Adding a WHERE clause to the SELECT statement allows only some rows to be selected from a table.

For example, suppose you would like to look at the employees with first name **John**.

❖ **List all employees named John:**

- Type the following:

```
SELECT *
FROM employee
WHERE emp_fname = 'John'
```

emp_id	manager_id	emp_fname	emp_lname	dept_id
318	1576	John	Crow	400
862	501	John	Sheffield	100
1483	1293	John	Letiecq	300

Apostrophes and case-sensitivity

- The apostrophes (single quotes) around the name 'John' are required. They indicate that **John** is a character string. Quotation marks (double quotes) have a different meaning. Quotation marks can be used to make otherwise invalid strings valid for column names and other identifiers.
- The sample database provided with SQL Anywhere is not case sensitive, so you would get the same results whether you searched for ' 'JOHN', 'john', or 'John'.

Again, you can combine what you have learned:

```
SELECT emp_fname, emp_lname, birth_date
FROM employee
WHERE emp_fname = 'John'
ORDER BY birth_date
```

Notes

- How you order clauses is important. The FROM clause comes first, followed by the WHERE clause, and then the ORDER BY clause. If you type the clauses in a different order, you will get a syntax error.
- You do not need to split the statement into several lines. You can enter the statement into the command window in any format. If you use more than the number of lines that fit on the screen, the text scrolls in the Command window.

Comparing dates in queries

Sometimes, you will not know exactly what value you are looking for, or you would like to see a set of values. You can use comparisons in the WHERE clause to select a set of rows that satisfy the search condition. The following example shows the use of a date inequality search condition.

❖ **To list all employees born before March 3, 1964"**

- Type the following:

```
SELECT emp_lname, birth_date
FROM employee
WHERE birth_date < 'March 3, 1964'
```

emp_lname	birth_date
Whitney	1958-06-05
Cobb	1960-12-04
Jordan	1951-12-13
Breault	1947-05-13
Espinoza	1939-12-14
Dill	1963-07-19

SQL Anywhere knows that the **birth_date** column contains a date, and converts *'March 3, 1964'* to a date automatically.

Compound search conditions in the WHERE clause

So far, you have seen equal (=) and less than (<) as comparison operators. SQL Anywhere also supports other comparison operators, such as greater than (>), greater than or equal (>=), less than or equal (<=), and not equal (<>).

These conditions can be combined using AND and OR to make more complicated search conditions.

❖ **To list all employees born before March 3, 1964, but exclude the employee named Whitney:**

- Type the following:

```
SELECT emp_lname, birth_date
FROM employee
WHERE birth_date < '1964-3-3'
AND emp_lname <> 'whitney'
```

emp_lname	birth_date
Cobb	1960-12-04
Jordan	1951-12-13
Breault	1947-05-13
Espinoza	1939-12-14
Dill	1963-07-19
Francis	1954-09-12

Pattern matching in search conditions

Another useful way to look for things is to search for a pattern. In SQL, the word LIKE is used to search for patterns. The use of LIKE can be explained by example.

❖ **To list all employees whose last name begins with BR:**

♦ Type the following:

```
SELECT emp_lname, emp_fname
FROM employee
WHERE emp_lname LIKE 'br%'
```

emp_lname	emp_fname
Breault	Robert
Braun	Jane

The % in the search condition indicates that any number of other characters may follow the letters **BR**.

❖ **To list all employees whose surname begins with** BR**, followed by zero or more letters and a** T**, followed by zero or more letters:**

♦ Type the following:

```
SELECT emp_lname, emp_fname
FROM employee
WHERE emp_lna -me LIKE 'BR%T%'
```

emp_lname	emp_fname
Breault	Robert

The first % sign matches the string *eaul*, while the second % sign matches the empty string (no characters).

Another special character that can be used with LIKE is the _ (underscore) character, which matches exactly one character.

The pattern **BR_U%** matches all names starting with **BR** and having **U** as the fourth letter. In **Braun** the _ matches the letter **A** and the **%** matches **N**.

Matching rows by sound

With the SOUNDEX function, you can match rows by sound, as well as by spelling. For example, suppose a phone message was left for a name that sounded like "Ms. Brown". Which employees in the company have names that sound like Brown?

❖ **To list employees with surnames that sound like Brown:**

- Type the following:

```
SELECT emp_lname, emp_fname
FROM employee
WHERE SOUNDEX( emp_lname ) = SOUNDEX( 'Brown' )
```

emp_lname	emp_fname
Braun	Jane

Jane Braun is the only employee matching the search condition.

Short cuts for typing search conditions

Using the shortform BETWEEN

SQL has two short forms for typing in search conditions. The first, BETWEEN, is used when you are looking for a range of values. For example,

```
SELECT emp_lname, birth_date
FROM employee
WHERE birth_date BETWEEN '1965-1-1'
AND '1965-3-31'
```

is equivalent to:

```
SELECT emp_lname, birth_date
FROM employee
WHERE birth_date >= '1965-1-1'
AND birth_date <= '1965-3-31'
```

Using the short form IN

The second short form, IN, may be used when looking for one of a number of values. The command

```
SELECT emp_lname, emp_id
FROM employee
WHERE emp_lname IN ('yeung','bucceri','charlton')
```

means the same as:

```
SELECT emp_lname, emp_id
FROM employee
WHERE emp_lname = 'yeung'
OR emp_lname = 'bucceri'
OR emp_lname = 'charlton'
```

CHAPTER 8

Joining Tables

About this chapter

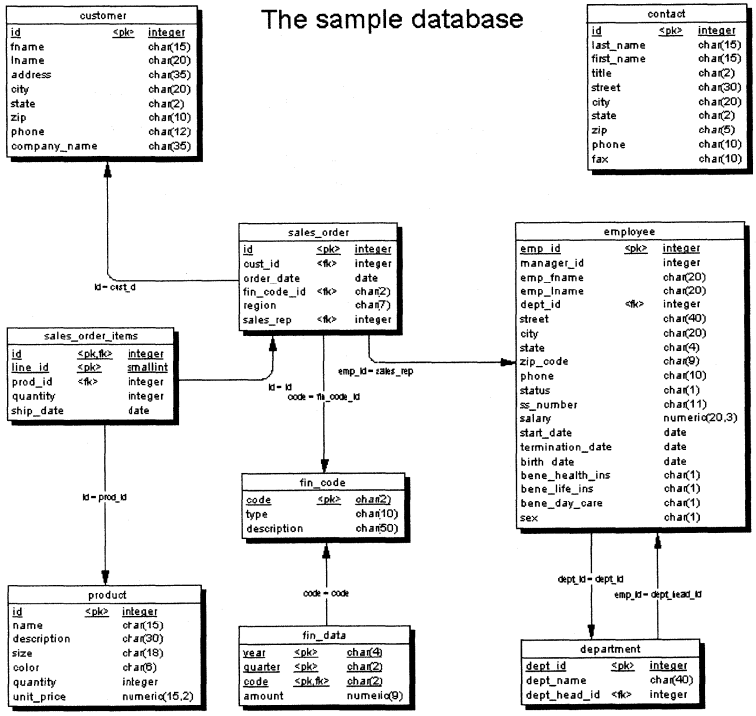
This chapter describes database queries that look at information in more than one table. To do this, SQL provides the **JOIN** operator. There are several different ways to join tables together in queries, and this chapter describes some of the more important ones.

Contents

Topic	Page
Displaying a list of tables	112
Joining tables with the cross product	114
Restricting a join	115
How tables are related	117
Join operators	118

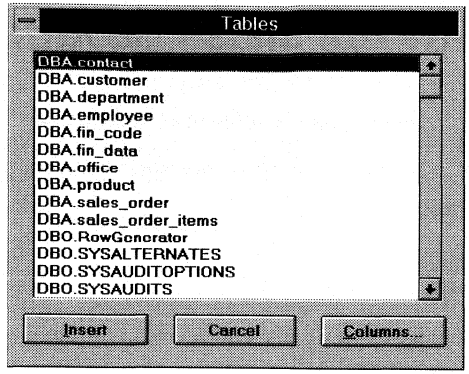
Displaying a list of tables

In the chapter "Selecting Data from Database Tables", you looked only at the employee table in the database. The sample database consists of nine tables, storing information about our fictional company.



Each box in the diagram represents a table in the database. The list of names in each box are the column names for the table.

In ISQL, you can display a list of tables by pressing the F7 key. The tables for the database are prefixed with *dba* (the owner of the tables).



The cursor keys can be used to scroll through the list of tables. Each table in the list is prefixed with a user name. This prefix is the ID of the user that created the table—the owner of the table.

Positioning the highlight on a particular table and pressing the Column List button displays the list of columns for that table. The ESCAPE key takes you back to the table list and pressing it again will take you back to the command window. Pressing ENTER instead of ESCAPE copies the highlighted table or column name into the command window at the current cursor position.

Press ESCAPE to leave the list.

Joining tables with the cross product

One of the tables in the sample database is sales_order, which lists the orders placed to the company. Each order has a sales_rep column, containing the employee ID of the sales representative responsible for the order. There are 648 rows in the sales_order table.

You can get information from two tables at the same time by listing both tables in the FROM clause of a SELECT query.

Example

The following example lists all the data in the employee table and the sales_order table:

```
SELECT *
FROM sales_order, employee
```

The results of this query, displayed in the ISQL data window, match every row in the employee table with every row in the sales_order table. Since there are 75 rows in the employee table and 648 rows in the sales_order table, there are 48,600 rows in the result of the join. Each row consists of all columns from the sales_order table followed by all columns from the employee table. This join is called a **full cross product**.

The cross product join is a simple starting point for understanding joins, but not very useful in itself.

Restricting a join

The most natural way to make a join useful is to insist that the sales_rep in the sales_order table be the same as the one in the employee table in every row of the result. Then each row contains information about an order and the sales rep responsible for it.

Example 1

To do this, add a WHERE clause to the previous query to show the list of employees and their course registrations:

```
SELECT *
FROM sales_order, employee
WHERE sales_order.sales_rep = employee.emp_id
```

The table name is given as a prefix to identify the columns. Although not strictly required in this case, using the table name prefix clarifies the statement, and is required when two tables have a column with the same name. A table name used in this context is called a **qualifier**.

The results of this query contain only 648 rows (one for each row in the sales_order table). Of the original 48,600 rows in the join, only 648 of them have the employee number equal in the two tables.

Example 2

The following query is a modified version that fetches only some of the columns and orders the results.

```
SELECT employee.emp_lname, sales_order.id,
sales_order.order_date
FROM sales_order, employee
WHERE sales_order.sales_rep = employee.emp_id
ORDER BY employee.emp_lname
```

If there are many tables in a SELECT command, you may need to type several qualifier names. This typing can be reduced by using a **correlation name**.

Correlation names

Correlation names are created by putting a short form for a table name immediately after the table name, separated by the word *AS*. The short form is then used as a **qualifier** instead of the corresponding table name, as follows:

```
SELECT E.emp_lname, S.id, S.order_date
FROM sales_order as S, employee as E
WHERE S.sales_rep = E.emp_id
ORDER BY E.emp_lname
```

Here, two correlation names S and E are created for the sales_order and employee tables.

If you try changing E.emp_id back to employee.emp_id, SQL Anywhere reports an error. If you make a correlation name for a table, you *must* use the correlation name when qualifying which table a column is from; you cannot use the original table name anymore.

How tables are related

In order to understand how to construct other kinds of joins, you must first understand how the information in one table is related to that in another.

The **primary key** for a table identifies each row in the table. Tables are related to each other using a **foreign key**.

This section shows how primary and foreign keys together let you construct queries from more than one table.

Rows are identified by a primary key

Every table in the employee database has a **primary key**. A primary key is one or more columns that uniquely identify a row in the table. For example, an employee number uniquely identifies an employee—emp_id is the primary key of the employee table.

The sales_order_items table is an example of a table with two columns that make up the primary key. The order ID by itself does not uniquely identify a row in the sales_order_items table because there can be several items in an order. Also, the line_id number does not uniquely identify a row in the sales_order_items table. Both the order ID name and line_id are required to uniquely identify a row in the sales_order_items table. The primary key of the table is both columns taken together.

Tables are related by a foreign key

There are several tables in the employee database that refer to other tables in the database. For example, the sales_order table has a sales_rep column to indicate which employee is responsible for an order. Only enough information to uniquely identify an employee is kept in the sales_order table. The sales_rep column in the sales_order table is a foreign key to the employee table.

Foreign key

A foreign key is one or more columns that contain primary key values from another table. Each foreign key relationship in the employee database is represented by an arrow between two tables. The arrow starts at the foreign key side of the relationship and points to the primary key side of the relationship.

Join operators

Many common joins are between two tables related by a foreign key. The most common join restricts foreign key values to be equal to primary key values. The example you have already seen restricts foreign key values in the sales_order table to be equal to the primary key values in the employee table.

```
SELECT emp_lname, id, order_date
FROM sales_order, employee
WHERE sales_order.sales_rep = employee.emp_id
```

The query can be more simply expressed using a KEY JOIN.

Joining tables using key joins

Key joins are an easy way to join tables related by a foreign key. For example:

```
SELECT emp_lname, id, order_date
FROM sales_order
   KEY JOIN employee
```

gives the same results as a query with a WHERE clause that equates the two employee number columns:

```
SELECT emp_lname, id, order_date
FROM sales_order, employee
WHERE sales_order.sales_rep = employee.emp_id
```

The join operator (KEY JOIN) is just a short cut for typing the WHERE clause; the two queries are identical.

If you look at the diagram of the employee database, foreign keys are represented by lines between tables. Anywhere that two tables are joined by a line in the diagram, you can use the KEY JOIN operator.

Joining two or more tables

Two or more tables can be joined using join operators. The following query uses four tables to list the total value of the orders placed by each customer. It connects the four tables customer, sales_order, sales_order_items and product using the lines between the tables.

```
SELECT company_name,
   CAST( SUM(sales_order_items.quantity *
   product.unit_price) AS INTEGER) AS value
FROM customer
   KEY JOIN sales_order
   KEY JOIN sales_order_items
   KEY JOIN product
GROUP BY company_name
```

company_name	value
Able Inc.	6120
AMF Corp.	3624
Amo & Sons	3216
Amy's Silk Screening	2028
Avco Ent	1752
...	

The CAST function used in this query converts the data type of an expression.

Joining tables using natural joins

The NATURAL JOIN operator joins two tables based on common column names. In other words, SQL Anywhere generates a WHERE clause that equates the common columns from each table.

Example

For example, for the following query:

```
SELECT emp_lname, dept_name
FROM employee
NATURAL JOIN department
```

the database engine looks at the two tables and determines that the only column name they have in common is dept_id. The following WHERE clause is internally generated and used to perform the join:

```
...
WHERE employee.dept_id = department.dept_id
```

Errors using NATURAL JOIN

This join operator can cause problems by equating columns you may not intend to be equated. For example, the following query generates unwanted results:

```
SELECT *
FROM sales_order
NATURAL JOIN customer
```

The result of this query has no rows.

The database engine internally generates the following WHERE clause:

```
WHERE sales_order.id = customer.id
```

The id column in the sales_order table is an ID number for the *order*. The ID column in the customer table is an ID number for the *customer*. None of them matched. Of course, even if a match were found, it would be a meaningless one.

You should be careful not to use join operators blindly. Always remember that the join operator just saves you from typing the WHERE clause for a foreign key or common column names. You should be conscious of the WHERE clause, or you may be creating queries that give results other than what you intend.

CHAPTER 9

Obtaining Aggregate Data

About this chapter

This chapter describes how to construct queries that tell you aggregate information. Examples of aggregate information are:

- The total of all values in a column
- The number of distinct entries in a column
- The average value of entries in a column

Contents

Topic	Page
A first look at aggregate functions	122
Using aggregate functions to obtain grouped data	123
Restricting groups	125

A first look at aggregate functions

Suppose you want to know how many employees there are. The following statement retrieves the number of rows in the employee table:

```
SELECT count( * )
FROM employee
```

count(*)
75

The result returned from this query is a table with only one column (with title count(*)) and one row, which contains the number of employees.

The following command is a slightly more complicated aggregate query:

```
SELECT   count( * ),
   min( birth_date ),
   max( birth_date )
FROM employee
```

count(*)	min(birth_date)	max(birth_date)
75	1936-01-02	1973-01-18

The result set from this query has three columns and only one row. The three columns contain the number of employees, the birthdate of the oldest employee, and the birthdate of the youngest employee.

COUNT, MIN and MAX are called **aggregate functions**. Each of these functions summarizes information for an entire table. In total, there are six aggregate functions: MIN, MAX, COUNT, AVG, SUM, and LIST. All but COUNT have the name of a column as a parameter. As you have seen, COUNT has an asterisk as its parameter.

Using aggregate functions to obtain grouped data

In addition to providing information about an entire table, aggregate functions can be used on groups of rows.

❖ **To list the number of orders each sales representative is responsible for:**

♦ Type the following:

```
SELECT sales_rep, count( * )
FROM sales_order
GROUP BY sales_rep
```

sales_rep	count(*)
129	57
195	50
299	114
467	56
667	54

The results of this query consist of one row for each sales rep ID number, containing the sales rep ID, and the number of rows in the sales_order table with that number.

Whenever GROUP BY is used, the resulting table has one row for each different value found in the GROUP BY column or columns.

A common error with GROUP BY

A common error with groups is to try to get information which cannot properly be put in a group. For example,

```
SELECT sales_rep, emp_lname, count( * )
FROM sales_order
   KEY JOIN employee
GROUP BY sales_rep
```

gives the error

```
column 'emp_lname' cannot be used unless it is in a
GROUP BY.
```

SQL does not realize that each of the rows for an employee with a given ID have the same value of emp_lname. An error is reported since SQL does not know which of the names to display.

However, the following is valid:

```
SELECT sales_rep, max( emp_lname ), count( * )
FROM sales_order
   KEY JOIN employee
GROUP BY sales_rep
```

The max function chooses the maximum (last alphabetically) surname from the detail rows for each group. The surname is the same on every detail row within a group so the max is just a trick to bypass a limitation of SQL.

Restricting groups

You have already seen how to restrict rows in a query using the WHERE clause. You can restrict GROUP BY clauses by using the HAVING keyword.

❖ **To list all sales reps with more than 55 orders:**

- Type the following:

```
SELECT sales_rep, count( * )
FROM  sales_order
KEY JOIN employee
GROUP BY sales_rep
HAVING count( * ) > 55
```

sales_rep	count(*)
129	57
299	114
467	56
1142	57

Order of clauses
GROUP BY must always appear before HAVING. In the same manner, WHERE must appear before GROUP BY.

HAVING clauses and WHERE clauses can be combined. When combining these clauses, the efficiency of the query can depend on whether criteria are placed in the HAVING clause or in the WHERE clause. Criteria in the HAVING clause restrict the rows of the result only after the groups have been constructed. Criteria in the WHERE clause are evaluated before the groups are constructed, and save time.

❖ **To list all sales reps with more than 55 orders and an ID of more than 1000:**

- Type the following:

```
SELECT sales_rep, count( * )
FROM sales_order
KEY JOIN employee
WHERE sales_rep > 1000
GROUP BY sales_rep
HAVING count( * ) > 55
```

The following statement produces the same results.

❖ **To list all sales reps with more than 55 orders and an ID of more than 1000:**

♦ Type the following:

```
SELECT sales_rep, count( * )
FROM sales_order
KEY JOIN employee
GROUP BY sales_rep
HAVING count( * ) > 55
AND sales_rep > 1000
```

The first statement is faster because it can eliminate making up groups for some of the employees. The second statement builds a group for each sales rep and then eliminates the groups with wrong employee numbers. For example, in the first statement, the database engine would not have to make up a group for the sales rep with employee ID 129. In the second command, the database engine would make up a group for employee 129 and eliminate that group with the HAVING clause.

Fortunately, SQL Anywhere detects this particular problem and changes the second query to be the same as the first. SQL Anywhere performs this optimization with simple conditions (nothing involving OR or IN). For this reason, when constructing queries with both a WHERE clause and a HAVING clause, you should be careful to put as many of the conditions as possible in the WHERE clause.

CHAPTER 10

Updating the Database

About this chapter

This chapter describes how to make changes in the contents of a database. It includes descriptions of how to add rows, remove rows, and change the contents of rows, as well as how to make changes permanent or back out of changes you have made.

Contents

Topic	Page
Adding rows to a table	128
Modifying rows in a table	129
Canceling changes	130
Making changes permanent	131
Deleting rows	132
Validity checking	133

Adding rows to a table

Suppose that a new eastern sales department is created, with the same manager as the current Sales department. You can add this information to the database using the following INSERT statement:

```
INSERT
INTO department ( dept_id, dept_name, dept_head_id )
VALUES ( 220, 'Eastern Sales', 902 )
```

If you make a mistake and forget to specify one of the columns, SQL Anywhere reports an error.

The NULL value is a special value used to indicate that something is either not known or not applicable. Some columns are allowed to contain the NULL value, and others are not.

A short form for INSERT

There is a short form that can be used if you are entering values for all the columns in a table in the order they appear when you SELECT * from the table (the same as the order in which they were created). The following is equivalent to the previous INSERT command:

```
INSERT
INTO department
VALUES ( 220, 'Eastern Sales', 902 )
```

Caution

You should use this form of INSERT with caution; it will not work as expected if you ever change the order of the columns in the table or if you add or remove a column from the table.

Modifying rows in a table

In most databases, you need to update data stored in the database. For example, suppose that the employee named James Klobucher (employee ID 467) is transferred from the sales department to the marketing department. In SQL, this is done using the UPDATE statement:

```
UPDATE employee
SET dept_id = 400
WHERE emp_id = 467
```

The WHERE clause identifies which employee to update.

Example: using the WHERE clause

SQL can update more than one column at a time. For example, the manager ID should change when employees are transferred between departments, as well as the department ID. The following statement carries out both updates at the same time for employee Marc Dill (employee ID 195):

```
UPDATE employee
SET dept_id = 400,
manager_id = 1576
WHERE emp_id = 195
```

The UPDATE and INSERT commands

The UPDATE and INSERT commands are two of the few places in SQL Anywhere where uppercase letters and lowercase letters are distinguished. New character values set by the UPDATE command are stored in the database exactly as they are entered.

SQL allows more than one row to be updated at one time. For example, if a group of sales employees are transferred into marketing and have their dept_id column updated, the following statement sets the manager_id for all employees in the marketing department to 1576.

```
UPDATE employee
SET manager_id = 1576
WHERE dept_id = 400
```

For employees already in the marketing department, no change is made.

It is also possible that an UPDATE statement updates no rows. For example, suppose you had made a mistake typing the employee ID in the first UPDATE statement above:

```
UPDATE employee
SET dept_id = 400
WHERE emp_id = 194
```

No rows would be updated since there is no employee with the employee ID 194.

Canceling changes

You may be a little concerned about all of the changes you have made to the employee table. However, SQL allows you to undo all of these changes with one command:

```
ROLLBACK
```

The ROLLBACK statement

The ROLLBACK statement undoes all changes you have made to the database since the last time you made changes permanent (see COMMIT in the next section).

The default action in ISQL is to do a COMMIT on exit. This can be controlled with the ISQL option COMMIT_ON_EXIT.

For more information on ISQL options, see "SET OPTION statement" in the chapter "Watcom-SQL Statements".

Making changes permanent

The SQL statement

```
COMMIT
```

makes all changes permanent.

> **Use COMMIT with care**
> When trying the examples in this tutorial, be careful not to COMMIT any changes until you are sure that you want to change the database permanently.

Making changes permanent in ISQL

The default action in ISQL is to do a COMMIT on exit. This can be controlled with the ISQL option COMMIT_ON_EXIT.

For more information on ISQL options, see "SET OPTION statement" in the chapter "Watcom-SQL Statements".

ISQL has another option, named AUTO_COMMIT. If this option is on, ISQL does a COMMIT operation after every command. The default for this option is OFF. Usually you will want it OFF, giving you the opportunity to ROLLBACK the changes (if, for example, an update or delete operation doesn't produce the intended results).

Deleting rows

Sometimes you will want to remove rows from a table. Suppose Rodrigo Guevara (employee ID 249) leaves the company. The following statement deletes Rodrigo Guevara from the employee table.

```
DELETE
FROM employee
WHERE emp_id = 249
```

Example: using the DELETE command

You can delete more than one row with one command. For example, the following statement would delete all employees who had a termination date that is not NULL from the employee table.

```
DELETE
FROM employee
WHERE termination_date IS NOT NULL
```

This example would not remove any employees from the database as the termination_date column is NULL for all employees.

With UPDATE and DELETE, the search condition can be as complicated as you need. For example, if the employee table is being reorganized, the following statement removes from the employee table all employees in the 617 area code with employee ID 902 as manager. This WHERE clause is a compound search condition including a function (LEFT).

```
DELETE
FROM employee
WHERE LEFT( phone, 3 ) = '617'
AND manager_id = 902
```

Since you have made changes to the database that you do not want to keep, you should undo the changes as follows:

```
ROLLBACK
```

Validity checking

SQL Anywhere automatically checks for some common errors in your data.

Inserting duplicate data

For example, suppose you attempt to create a department but supply a dept_id value that is already in use:

To do this, enter the command:

```
INSERT
INTO department ( dept_id, dept_name, dept_head_id )
VALUES ( 200, 'Eastern Sales', 902 )
```

The INSERT is rejected, as it would make the primary key for the table not unique.

Primary key

A primary key is a set of columns that uniquely identifies each row in a table. For example, the dept_id column is the primary key for the department table; given a valid department ID number, there is exactly one row in the department table with that number. The primary key for the sales_order_items table is composed of the id and line_id columns, meaning that there should never be two items in the same order with the same line number.

Inserting incorrect values

Another mistake is to type an incorrect value. The following statement inserts a new row in the sales_order table, but incorrectly supplies a sales_rep ID that does not exist in the employee table.

```
INSERT
INTO sales_order ( id, cust_id, order_date,
   sales_rep)
VALUES ( 2700, 186, '1995-10-19', 284 )
```

Foreign key

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

When you try to add an order for sales rep 284 you get an error message:

no primary key for foreign key 'ky_so_employee_id' in table 'sales_order'

There is no employee in the employee table with that ID number. This prevents you from inserting orders without a valid sales rep ID. This kind of validity checking is called **referential integrity** checking, as it maintains the integrity of references among the tables in the database.

Errors on DELETE or UPDATE

Foreign key errors can also arise when doing update or delete operations. For example, suppose you try to remove the R&D department from the department table.

```
DELETE
FROM department
WHERE dept_id = 100
```

Example: DELETE errors

An error is reported indicating that there are other records in the database that reference the R&D department, and the delete operation is not carried out.

primary key for row in table 'department' is referenced in another table

In order to remove the R&D department, you need to first get rid of all employees in that department:

```
DELETE
FROM employee
WHERE dept_id = 100
```

You can now perform the deletion of the R&D department.

You should cancel these changes to the database (for future use) by entering a ROLLBACK statement:

```
ROLLBACK WORK
```

All changes made since the last successful COMMIT WORK will be undone. If you have not done a COMMIT, then all changes since you started ISQL will be undone.

Example: UPDATE errors

The same error message is generated if you perform an update operation that makes the database inconsistent.

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

```
UPDATE department
    SET dept_id = 600
    WHERE dept_id = 100
```

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

Example: checking the integrity after the COMMIT WORK is complete

It is possible to configure the database so that the integrity is not checked until the COMMIT WORK is done. This is important if you want to change the value of a referenced primary key; for example, changing the R&D department's ID from 100 to 600 in the department and employee tables. In order to make these changes, the database has to be inconsistent in between the changes. In this case, you must configure the database to check only on commits.

For information on the WAIT_FOR_COMMIT database option, see "SET OPTION statement" in the chapter "Watcom-SQL Statements".

You can also define foreign keys in such a way that they are automatically fixed. In the above example, if the foreign key from employee to department were defined with ON UPDATE CASCADE, then updating the department ID would automatically update the employee table.

In the above cases, there is no way to have an inconsistent database committed as permanent. SQL Anywhere also supports alternative actions if changes would render the database inconsistent.

For more information, see the chapter "Ensuring Data Integrity".

CHAPTER 11

Introduction to Views

About this chapter

It is often impractical to repeatedly type complicated queries. SQL provides **views**, that allow you to give names to frequently executed SELECT commands. A view is a computed table which is useful for security purposes, and for tailoring the appearance of database information to make data access straightforward.

This chapter introduces you to views.

Contents

Topic	Page
Defining a view	138
Using views for security	140

Defining a view

Suppose that you frequently need to list a summary of employees and the departments they belong to. The following query produces the information you need.

❖ **To list employees and the departments to which they belong:**

♦ Type the following:

```
SELECT emp_fname, emp_lname, dept_name
FROM employee JOIN department
ON department.dept_id = employee.dept_id
```

emp_fname	emp_lname	department
Fran	Whitney	R&D
Matthew	Cobb	R&D
Robert	Breault	R&D
Natasha	Shishov	R&D
Kurt	Driscoll	R&D

You can create a view that produces the results of this command as follows:

```
CREATE VIEW emp_dept AS
SELECT emp_fname, emp_lname, dept_name
FROM employee JOIN department
ON department.dept_id = employee.dept_id
```

This command creates a view called emp_dept that looks in many respects just like any other table in the database.

You can list everything in this view just as you do from a table:

❖ **To list employees and the departments to which they belong:**

♦ Type the following:

```
SELECT *
FROM emp_dept
```

emp_fname	emp_lname	department
Fran	Whitney	R&D
Matthew	Cobb	R&D
Robert	Breault	R&D

emp_fname	emp_lname	department
Natasha	Shishov	R&D
Kurt	Driscoll	R&D

It is important to remember that the information in a view is not stored separately in the database. Each time you refer to the view, SQL executes the associated SELECT statement to find the appropriate data.

On one hand, this is good; it means that if someone modifies the employee table or the department table, the information in the emp_dept view will be automatically up to date. On the other hand, if the SELECT command is complicated it may take a long time for SQL to find the correct information every time you use the view.

Providing names for the view columns

You can provide names for the view columns explicitly. First you must get rid of the original view definition as follows:

```
DROP VIEW emp_dept
```

You can redefine the view with the new column name as follows:

```
CREATE VIEW emp_dept(FirstName, LastName,
Department) AS
SELECT emp_fname, emp_lname, dept_name
FROM employee JOIN department
ON department.dept_id = employee.dept_id
```

You have changed the names of the columns in the view by specifying new column names in parentheses after the view name.

More about views

Views can be thought of as computed tables. Any SELECT command can be used in a view definition except commands containing ORDER BY. Views can use GROUP BY clauses, subqueries, and joins. Disallowing ORDER BY is consistent with the fact that rows of a table in a relational database are not stored in any particular order. When you use the view, you can specify an ORDER BY.

You can also use views in more complicated queries. Here is an example using a join:

```
SELECT LastName, dept_head_id
FROM emp_dept, department
WHERE emp_dept.Department = department.dept_name
```

Using views for security

Example

Views can be used to restrict access to information in the database. For example, suppose you wanted to create a user ID for the sales department head, Moira Kelly, and restrict her user ID so that it can only examine information about employees in the sales department.

Creating the new user ID

First you need to create the new user ID for Moira Kelly using the GRANT statement. From ISQL, connected to the sample database as dba, enter the following:

```
GRANT CONNECT TO M_Kelly
IDENTIFIED BY SalesHead
```

Granting permissions

Next you need to grant user M_Kelly the right to look at employees of the sales department.

```
CREATE VIEW SalesEmployee AS
SELECT emp_id, emp_lname, emp_fname
FROM "dba".employee
WHERE dept_id = 200
```

The table should be identified as "dba".employee for the M_Kelly user ID to be able to use the view.

Now you must give M_Kelly permission to look at the new view by entering:

```
GRANT SELECT ON SalesEmployee TO M_Kelly
```

Looking at the view

Connect to the database as M_Kelly and now try looking at the view:

```
CONNECT USER M_Kelly IDENTIFIED BY SalesHead ;
SELECT * FROM "dba".SalesEmployee
```

emp_id	emp_lname	emp_fname
129	Chin	Philip
195	Dill	Marc
299	Overbey	Rollin
467	Klobucher	James
641	Powell	Thomas

However, you do not have permission to look directly at the employee and department tables. If you execute the following commands, you will get permission errors.

```
SELECT * FROM "dba".employee ;
SELECT * FROM "dba".department
```

CHAPTER 12

Introduction to Subqueries

About this chapter

This chapter shows how to use the results of one query as part of another SELECT statement. This is a useful tool in building more complex and informative queries.

Contents

Topic	Page
Preparing to use subqueries	142
A simple subquery	143
Comparisons using subqueries	145
Using subqueries instead of joins	148

Preparing to use subqueries

Sometimes it is useful to use the results of one statement as part of another statement.

Example 1

For example, suppose that you need a list of order items for products that are low in stock.

You can look up the products for which there are less than 20 items in stock in the product table.

❖ **To list all products for which there are less than 20 items in stock:**

♦ Type the following:

```
SELECT id, description, quantity
FROM product
WHERE quantity < 20
```

id	description	quantity
401	Wool cap	12

This query shows that only wool caps are low in stock.

Example 2

You can list all the order items for wool caps with the following query:

❖ **To list all orders for wool caps, most recent first:**

♦ Type the following:

```
SELECT *
FROM sales_order_items
WHERE prod_id = 401
ORDER BY ship_date DESC
```

id	line_id	prod_id	quantity	ship_date
2082	1	401	48	1994-07-09
2053	1	401	60	1994-06-30
2125	2	401	36	1994-06-28
2027	1	401	12	1994-06-17
2062	1	401	36	1994-06-17

This two-step process of identifying items low in stock and identifying orders for those items can be combined into a single query using subqueries.

A simple subquery

SQL provides another way to find orders for items low in stock. The following query incorporates a **subquery**.

Example 1

❖ **To list order items for products low in stock:**

♦ Type the following:

```
SELECT *
FROM sales_order_items
WHERE prod_id IN
 ( SELECT id
FROM product
WHERE quantity < 20 )
ORDER BY ship_date DESC
```

id	line_id	prod_id	quantity	ship_date
2082	1	401	48	1994-07-09
2053	1	401	60	1994-06-30
2125	2	401	36	1994-06-28
2027	1	401	12	1994-06-17
2062	1	401	36	1994-06-17

By using a subquery, the search can be carried out in just one query, instead of using one query to find the list of low-stock products and a second to find orders for those products.

The subquery in the statement is the phrase enclosed in parentheses:

```
(  SELECT id
FROM product
WHERE quantity < 20 )
```

The subquery makes a list of all values in the id column in the product table satisfying the WHERE clause search condition.

Example 2

Consider what would happen if an order for ten tank tops were shipped so that the quantity column for tank tops contained the value 18. The query using the subquery, would list all orders for both wool caps and tank tops. On the other hand, the first statement you used would have to be changed to the following:

```
SELECT *
FROM sales_order_items
WHERE prod_id IN ( 401, 300 )
ORDER BY ship_date DESC
```

The command using the subquery is an improvement because it still works even if data in the database is changed.

Example 3

As another example, you can list orders for everything *except* those products in short supply with the query:

```
SELECT *
FROM sales_order_items
WHERE prod_id NOT IN
    (  SELECT id
    FROM product
    WHERE quantity < 20 )
ORDER BY ship_date DESC
```

Comparisons using subqueries

Two tables in the sample database are concerned with financial results. The fin_code table is a small table holding the different codes for financial data and their meanings:

❖ **To list the contents of the fin_code table:**

- Type the following:

```
SELECT *
FROM fin_code
```

code	type	description
e1	expense	Fees
e2	expense	Services
e3	expense	Sales & Marketing
e4	expense	R&D
e5	expense	Administration
r1	revenue	Fees
r2	revenue	Services

The fin_data table holds financial data for each financial code for each quarter.

❖ **To list the contents of the fin_data table:**

- Type the following:

```
SELECT *
FROM fin_data
```

year	quarter	code	amount
1992	Q1	e1	101
1992	Q1	e2	403
1992	Q1	e3	1437
1992	Q1	e4	623
1992	Q1	e5	381

The following query uses a subquery to list just the revenue items from the fin_data table.

❖ **To list the revenue items from the fin_data table:**

♦ Type the following:

```
SELECT *
FROM fin_data
WHERE fin_data.code IN
(  SELECT fin_code.code
FROM fin_code
WHERE type = 'revenue' )
```

year	quarter	code	amount
1992	Q1	r1	1023
1992	Q2	r1	2033
1992	Q3	r1	2998
1992	Q4	r1	3014
1993	Q1	r1	3114

This example has used qualifiers to clearly identify the table to which the code column in each reference belongs. In this particular example, the qualifiers could have been omitted.

Notes about subqueries

Subqueries are restricted to one column name listed between SELECT and FROM: one select-list item. The following example does not make sense, since SQL would not know which column from fin_code to compare to the fin_data.code column.

```
SELECT *
FROM fin_data
WHERE fin_data.code IN
    (  SELECT fin_code.code, fin_code.type
    FROM fin_code
    WHERE type = 'revenue' )
```

Further, while subqueries used with an IN condition may return several rows, a subquery used with a comparison operator must return only one row. For example, the following command will result in an error since the subquery returns two rows *(r1*, and *r2*):

```
SELECT *
FROM fin_data
WHERE fin_data.code =
     (  SELECT fin_code.code
    FROM fin_code
    WHERE type = 'revenue' )
```

The IN comparison allows several rows. Two other keywords can be used as qualifiers for operators to allow them to work with multiple rows: ANY and ALL.

The following query is identical to the successful query above:

```
SELECT * s
FROM fin_data
WHERE fin_data.code = ANY (  SELECT fin_code.code
    FROM fin_code
    WHERE type = 'revenue' )
```

While the = *ANY* condition is identical to the IN condition, ANY can also be used with inequalities such as, or, to give more flexible use of subqueries.

The word ALL is similar to the word ANY. For example, the following query lists financial data that is not revenues:

```
SELECT *
FROM fin_data
WHERE fin_data.code <>
    ALL (  SELECT fin_code.code
    FROM fin_code
    WHERE type = 'revenue' )
```

This is equivalent to the following command using NOT IN:

```
SELECT *
FROM fin_data
WHERE fin_data.code NOT IN
     (  SELECT fin_code.code
    FROM fin_code
    WHERE type = 'revenue' )
```

Using subqueries instead of joins

Suppose you need a chronological list of orders and the company that placed them, but would like the company name instead of their customer ID. You can get this result using a join as follows:

Using a join

❖ **To list the order id, date, and company name for each order since the beginning of 1994:**

♦ Type the following:

```
SELECT     sales_order.id,
sales_order.order_date,
customer.company_name
FROM sales_order
KEY JOIN customer
WHERE order_date > '1994/01/01'
ORDER BY order_date
```

id	order_date	company_name
2473	1994-01-04	Peachtree Active Wear
2474	1994-01-04	Sampson & Sons
2036	1994-01-05	Hermanns
2106	1994-01-05	Salt & Pepper's
2475	1994-01-05	Cinnamon Rainbow's

Using a subquery

The following statement obtains the same results using a subquery instead of a join:

```
SELECT   sales_order.id,
sales_order.order_date,
    ( SELECT company_name FROM customer
    WHERE customer.id = sales_order.cust_id )
FROM   sales_order
WHERE order_date > '1994/01/01'
ORDER BY order_date
```

The subquery refers to the cust_id column in the sales_order table even though the sales_order table is not part of the subquery. Instead, the sales_order.cust_id column refers to the sales_order table in the main body of the statement. This is called an **outer reference**. Any subquery that contains an outer reference is called a **correlated subquery**.

A subquery can be used instead of a join whenever only one column is required from the other table. (Recall that subqueries can only return one column.) In this example, you only needed the company_name column so the join could be changed into a subquery.

If the subquery might have no result, this method is called an **outer join**. The join in previous sections of the tutorial is more fully called an **inner join**.

Using an outer join

❖ **To list all customers in Washington State together with their most recent order ID:**

♦ Type the following:

```
SELECT      company_name, state,
( SELECT MAX( id )
FROM sales_order
WHERE sales_order.cust_id = customer.id )
FROM customer
WHERE state = 'WA'
```

company_name	MAX(id)	state
Custom Designs	2547	WA
It's a Hit!	(NULL)	WA

The It's a Hit! company placed no orders, and the subquery returns NULL for this customer. Companies who have not placed an order would not be listed if an inner join was used.

You could also specify an outer join explicitly. In this case a GROUP BY clause is also required.

```
SELECT  company_name,
   MAX( sales_order.id ),state
FROM customer
   KEY LEFT OUTER JOIN sales_order
WHERE state = 'WA'
GROUP BY company_name, state
```

CHAPTER 13

Command Files

About this chapter

This chapter describes how to use the ISQL command window to enter multiple commands at a time and to process files consisting of a set of commands.

Contents

Topic	Page
Entering multiple statements in the ISQL Command window	152
Saving statements as command files	153
Command files with parameters	154

Entering multiple statements in the ISQL Command window

SQL commands can get quite large. You have already seen how to use the editor to enter commands on several lines. The ISQL environment also allows multiple commands to be entered at the same time. This is done by ending each command with a semi-colon (;).

You may want to grow the Command window:

- In DOS, OS/2 or QNX, choose Command➤Zoom from the menu bar.
- In Windows or Windows NT, click the maximize button.

Example: entering multiple statements

❖ To enter multiple statement in the ISQL Command Window

1 Try entering the following three commands into the Command window.

```
UPDATE employee
SET dept_id = 400,
manager_id = 1576
WHERE emp_id = 467;

UPDATE employee
SET dept_id = 400,
manager_id = 1576
WHERE emp_id = 195;

SELECT *
FROM employee
WHERE emp_id IN ( 195, 467 )
```

2 Press the execute key (F9). All three of these commands are executed. After execution, the commands are left in the Command window.

You can modify the commands if there are errors.

Saving statements as command files

You can also save the commands entered in the previous section to a command file. This keeps a permanent record of the SQL commands so they can be used later if you wish.

❖ **To save statements as a command file:**

1 Choose File ➤Save As from the menu bar. You are then prompted for a filename.

2 Type **transfer** and press ENTER.

3 The command file can be run using the ISQL READ command, but you should rollback the changes first. Press the ESCAPE key to clear the editor and then execute the ROLLBACK WORK command.

4 Now enter the following command:

```
READ transfer
```

This command executes the command file *transfer* which contains the three commands that we saved previously. As each command is executed, it flashes up in the Command window.

5 You can load command files back into the Command window by choosing File ➤Open from the menubar.

6 Enter **transfer** when prompted for the file name. Notice that the commands have been loaded back into the editor just the way they were when they were saved.

What are command files?

Command files are just ASCII files containing the ISQL commands as you see them in the editor. You can use any editor you like to create command files. You can include comment lines along with the SQL statements to be executed. Comments begin with a percent sign (%). The ISQL READ command is used to execute command files. Alternatively, they can be loaded into the ISQL Command window and executed directly from there.

The Command window in ISQL has a limit of 500 lines. For command files larger than this, you should use a generic editor capable of handling large files. The READ command has no limit on the number of lines that can be read.

Command files with parameters

An example of a command file that would take a parameter is a command file to show the department an employee belongs to, providing the employee's name as a parameter.

❖ **To create a command file with parameters:**

1 Create a command file as listed below.
The PARAMETERS command is used to give names to the parameters passed to a command file. In this case, we are giving the first parameter the name employee_name. The parameters are then used in the rest of the command file by enclosing them in braces ({}). Save the command file to emp_dept.sql.

```
% This command file has one parameter
parameters employee_name;

select emp_lname, dept_name
from  employee
    NATURAL JOIN department
where emp_lname = {employee_name};
```

2 Run this command file by typing:

```
READ emp_dept.sql
```

3 You will be prompted for the employee_name. Enter the following value, including the single quotes:

```
'Whitney'
```

You should now see that the employee with surname Whitney is in the R&D department.

Parameters specifies on the READ command

Parameters can also be specified on the READ command. Try the following command:

```
READ emp_dept.sql 'whitney'
```

In this case you have specified the parameter on the READ command, so ISQL will not prompt for it. ISQL will only prompt for parameters that are named in the PARAMETERS command but are not supplied on the READ command.

CHAPTER 14

System Tables

About this chapter

This chapter describes the **system tables**, several special tables found in every SQL Anywhere database. These system tables describe all the tables and columns in the database. The SQL Anywhere software automatically updates the system table as the database structure is changed.

This chapter introduces the system tables.

Contents

Topic	Page
The SYSCATALOG table	156
The SYSCOLUMNS table	157
Other system tables	158

The SYSCATALOG table

The first system table lists all the tables in the database.

❖ **To view the contents of the SYSCATALOG table for the sample database:**

1. Type the following command:

```
SELECT *
FROM sys.syscatalog
```

   The first screen of tables lists some of the system tables found in SQL Anywhere.

2. Scroll through the ISQL data window a few times, you will see the tables that make up the company database.

The creator of the system table

The creator of the system tables is the special user ID, SYS and the creator of the company tables is dba. In addition, there is a set of views owned by the special user ID DBO, which provide an emulation of the Sybase SQL Server system catalog; these tables are not discussed in this section.

(Recall that dba is the user ID you used when connecting to the database from ISQL.) So far, you have simply typed the table names employee and department; SQL looked in SYSCATALOG for tables with those names created by dba. In this example, by typing SYS.SYSCATALOG you specified that SYSCATALOG was created by the user ID SYS. Note the similarity to the way column names are qualified, such as employee.emp_id.

Other columns in the system table

The other columns in this table contain other important information. For example, the column named Ncols is the number of columns in each table, and the column named tabletype identifies the table as a permanent table (also called a **base table**) or a view.

The SYSCOLUMNS table

Another important system table is called SYSCOLUMNS describing all the columns in all the tables in the database. To see the contents of this table, type the command:

```
SELECT *
FROM sys.syscolumns
WHERE tname = 'employee'
```

This command lists all the columns in the employee table. If you look at the columns to the right, you can see from the Coltype column that some columns in the employee table contain character information; others contain integer and date information.

Other system tables

There are several other system tables in the database that will not be described in the tutorial. You can find out their names by examining SYS.SYSCATALOG and look at them if you want.

For a full description of each of the system tables, see the chapter "SQL Anywhere System Tables".

PART THREE

Using SQL Anywhere

This part describes how to use the SQL Anywhere database management system.

CHAPTER 15

Connecting to a Database

About this chapter

This chapter describes how client applications connect to SQL Anywhere databases and contains all the SQL Anywhere-specific information about connecting to databases from ODBC-enabled applications and application development systems.

If you are writing a program directly using one of the programming interfaces to SQL Anywhere, see "Programming Interfaces" for implementation details.

Contents

Topic	Page
Connection overview	162
Connecting from the SQL Anywhere utilities	168
Connecting from an ODBC-enabled application	169

Connection overview

Any client application that uses a database must establish a **connection** to that database before any work can be done. While the user may be prompted to enter a user ID, a password, and other parameters, the connection is established by the client application through one of SQL Anywhere's programming interfaces.

Once the connection is established, it forms the channel through which all the activity you engage in from the client application takes place. For example, the permissions you have to carry out actions on the database are determined by your user ID—and the database engine is aware of your user ID because it is part of the request to establish a connection.

Database connection parameters

When an application connects to a database, it uses a set of **connection parameters**, collected together in a **connection string**, to define the connection. For example, connection strings specify a user ID and password, as well as other optional information; all connections using ODBC must supply a Data Source name.

Connection parameters used with SQL Anywhere databases

A large class of client applications uses a very similar set of connection parameters when attempting to connect to a SQL Anywhere database. These are:

- All applications connecting to SQL Anywhere databases through the ODBC interface using the **SQLDriverConnect** function. This includes many ODBC-enabled application development systems, as well as applications developed using those systems.
- The SQL Anywhere database utilities ISQL, DBBACKUP, DBWATCH, DBUNLOAD, and DBVALID.
- All other embedded SQL applications connecting using the db_string_connect function, which is the recommended way to connect to a database in embedded SQL.

Methods used to specify connection parameters

The way you specify the connection parameters depends on the particular client application you are working from. Applications may use one of the following methods:

- ODBC-enabled applications use a Data Source configuration, which may be entered using the ODBC Administrator program

For more information on adding an ODBC data source using the ODBC Administrator, see "Adding an ODBC data source" on page 174.

- You may be prompted to fill in fields in a dialog box
- You may be required to enter command-line arguments
- The application may look in prespecified files to find the parameter values
- The application may use environment variable settings
- The application may have a fixed set of parameters built in permanently

Keywords used in connection strings

All these applications use a connection string consisting of a list of parameter settings, each of the form **keyword**=*value,* delimited by semicolons. The keywords must come from the following table.

Verbose keyword	Short form	Argument
Agent	Agent	string (**Client** or **Engine**)
AutoStop	AutoStop	YES/NO
ConnectionName *	CON	string
DatabaseFile	DBF	string
DatabaseName	DBN	string
DatabaseSwitches	DBS	string
DatasourceName #**	DSN	string
EngineName	ENG	string
Password **	PWD	string
Start	Start	string
Userid **	UID	string

* Not supported in ODBC connections

Supported in ODBC connections only

** Verbose form of keyword not supported in ODBC connection parameters

For information on other ODBC-specific connection parameters, see "Adding an ODBC data source", on page 174.

Connection keyword meanings

The meaning of each of the connection parameter keywords is as follows:

Keyword	Description
Agent	If you have both a standalone engine and a client available, and you wish to ensure that you correct to one of these, you can specify **Agent=client** or **Agent=engine** to ensure a connection to the appropriate agent..
AutoStop	You should supply an AutoStop keyword only if you are connecting to a database that is not currently running. If AutoStop is set to *yes* then the database is unloaded automatically as soon as there are no more open connections to it.
ConnectionName	An optional parameter, providing a name for the particular connection you are making. You may leave this unspecified unless you are going to establish more than one connection, and switch between them. ConnectionName is not used when connecting through ODBC
DatabaseFile	The root file of the database to which you want to connect You should supply a DatabaseFile only if you are connecting to a database that is not currently running. For example, to start and connect to the sample database (installed in directory c:\sqlany50), use a DatabaseFile entry of c:\sqlany50\sademo.db.
DatabaseName	When a database is started, it is assigned a database name. The default database name is the name of the database file with the extension and path removed. For more information about the database name see "Some database terms" in the chapter "Overview of SQL Anywhere". For information on this field, see "How client applications connect to a database" on page 166

Keyword	Description
DatabaseSwitches	You should supply DatabaseSwitches only if you are connecting to a database that is not currently running. When the engine starts the database specified by DatabaseFile, the engine uses the supplied DatabaseSwitches as command line options to determine startup options for the database. Only *database* switches can be supplied using this keyword. Engien or server switches must be supplied using the START connection parameter. For more information about database switches, see "The database engine" in the chapter "SQL Anywhere Components".
DataSourceName	The DataSourceName tells an ODBC application where to look in ODBC.INI to find information about the database to which you want to connect. DataSourceName is used only by ODBC-enabled applications, and is compulsory for this class of application
EngineName	The name of a running database engine or server to which you want to connect. You need to supply a server name only if more than one database engine is running. For more information about the engine name see "Some database terms" in the chapter "Overview of SQL Anywhere". In the Sybase Central and ISQL Connect dialog box, and in the ODBC Administrator, this is the *Server Name* field
Password	Your database password. You must always supply a password when connecting to a database
Start	You should supply a Start keyword only if you are connecting to a database engine that is not currently running. The Start parameter is a command line to start a database engine. For a detailed description of available command line switches, see "The database engine" in the chapter "SQL Anywhere Components".
Userid	The user ID with which you log on to the database. You must always supply a user id when connecting to a database

☞ For a detailed description of how client applications use the connection parameters when connecting to a database, see "How client applications connect to a database" next.

How client applications connect to a database

Once you have specified the connection parameters, using whatever method your client application requires, the application attempts to connect to a database. The procedure the client follows is exactly the same for each of two important sets of client application:

- Any ODBC-enabled client application. Many application development systems, such as Powersoft PowerBuilder, belong to this class of application.
- Any client application using embedded SQL and using the recommended function for connecting to a database (db_string_connect). All the SQL Anywhere database tools, including ISQL, are a part of this set.

Connecting to a database

Connecting to a database is of fundamental importance: you cannot do any work on a database without first connecting to it. For this reason, the process followed by client applications is described in detail:

1 The application tries to find the appropriate database engine or server.

   - If EngineName is specified, the application looks for a local database engine with that name, and then for a SQL Anywhere Client (DBCLIENT) with that server name. For QNX, the application looks on the network for an engine or server.
   - If EngineName is not specified, the application tries to connect to the default local engine. (If there is only one engine running, it is the default, otherwise the default choice is operating-system specific.)

2 If no matching local database engine is found, and the SQL Anywhere Client is not running, the application starts a database engine or SQL Anywhere Client using the Start parameter.

   - If no Start parameter is specified, but DatabaseFile is specified, the application attempts to start a database engine on the named file, using a default start parameter.

   ☞ For more information about database engine parameters, see "The database engine" in the chapter "SQL Anywhere Components".

3 If the application successfully finds or starts a database, the application tries to connect to the database. (If no engine has yet been found or started, the attempt to connect fails at this point)

- If neither a DatabaseName nor a DatabaseFile is given, the application attempts to connect to the default database on the engine, using the specified Userid, Password, and ConnectionName parameters.
- If the database named by DatabaseName is running, the application attempts to connect to the database using the specified Userid, Password, and ConnectionName parameters.
- If DatabaseName is not specified, but DatabaseFile is, the application attempts to connect to a database whose name is the root of DatabaseFile.
- If no running database is found, but a database specified by the root of DatabaseFile is running, the application attempts to connect to the database using the specified Userid, Password, and ConnectionName parameters.
- If the database corresponding to DatabaseFile is not running, the application sends a request to the engine or network server to start a database using the DatabaseFile, DatabaseName, and DatabaseSwitches parameters. (The AutoStop parameter determines if the database automatically stops when the last connection to the database is disconnected.) The application then attempts to connect to the database using the specified Userid, Password, and ConnectionName parameters.
- If no connection is made, the attempt fails at this point.

CONNECT statement from ISQL

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. This behavior is required for database reloading operations.

For an example, see "CONNECT statement" in the chapter "Watcom-SQL Statements".

Connecting from the SQL Anywhere utilities

All SQL Anywhere database utilities that communicate with the database engine (rather than acting directly on database files) do so using embedded SQL, and follow the procedure outlined in "How client applications connect to a database" on page 166 when connecting to a database.

How database tools obtain connection parameter values

Many of the database tools obtain the values of the connection parameters in the following way:

1. If there are values specified on the command line, those values are used for the connection parameters. For example, the following command starts a backup of the default database on the default database engine using the user ID *DBA* and the password *SQL*:

   ```
   DBBACKUP -c "UID=DBA;PWD=SQL" c:\backup
   ```

2. If any command line values are missing, the application looks at the setting of the SQLCONNECT environment variable. This variable is not set automatically by SQL Anywhere. If you use a single set of connection parameters frequently, you may want to place a SQLCONNECT environment variable in your AUTOEXEC.BAT file (under DOS and Windows 3.x), your CONFIG.SYS file (under OS/2), the Control Panel (under NT) or the login file (under QNX).

   For a description of the SQLCONNECT environment variable, see "Registry entries and environment variables" in the chapter "SQL Anywhere Components".

3. If parameters are not set in the command line (if applicable), or the SQLCONNECT environment variable, then by the connection procedure described above, the application prompts for a user ID and password to connect to the default database on the default database engine

For a description of command line switches for each database tool, see chapter "SQL Anywhere Components".

Connecting from an ODBC-enabled application

The **Open Database Connectivity** (**ODBC**) interface is defined by Microsoft Corporation, and is a standard interface for connecting client applications to database management systems in the Windows and Windows NT environments. Many client applications, including application development systems, use the ODBC interface to access a wide range of database systems. These are **ODBC-enabled** applications. SQL Anywhere supports the ODBC interface.

This section describes how to connect to SQL Anywhere from ODBC-enabled applications.

For information about how to write an ODBC program, see the chapter "ODBC Programming".

ODBC-enabled applications use a set of connection parameters in the manner described above. However, they obtain values for those parameters in a different manner to embedded SQL applications. ODBC defines data sources. Each data source description contains several of the required connection parameters. The other connection parameters are obtained from the user (for instance, you may be presented with a dialog box to enter a user ID and password), or internally by the client application.

SQL Anywhere support for ODBC

SQL Anywhere provides ODBC Version 2.5 support at conformance Level 2, which is the highest level of support for ODBC Version 2.5.

With Windows and Windows NT

For Windows and Windows NT, SQL Anywhere ODBC support takes place using the Microsoft ODBC driver manager, installed as part of the SQL Anywhere installation. The ODBC driver manager enables different ODBC drivers to run at the same time and allows an ODBC-enabled application to communicate with more than one ODBC driver and data source.

For more information about using ODBC under Windows and Windows NT, see "Using ODBC under Windows and Windows NT" on page 170.

With DOS and QNX

For DOS and QNX, SQL Anywhere supports ODBC as an API (application programming interface) only.

For more information about using ODBC under DOS and QNX, see "Using ODBC under DOS and QNX" on page 180.

With OS/2

SQL Anywhere supports ODBC as an application programming interface for OS/2. Additional software from third party vendors does allow the use of the SQL Anywhere ODBC driver from ODBC-enabled client applications through the third party ODBC driver manager.

For more information about using ODBC client applications under OS/2, see "Using ODBC under OS/2" on page 179

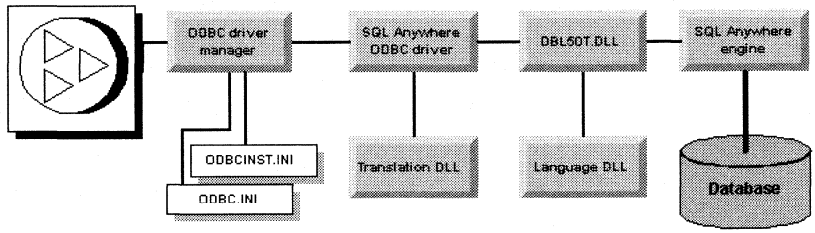
Using ODBC under Windows and Windows NT

All the information about installed ODBC database management systems and databases on your machine is held in two files (in Windows 3.x) or in the registry (in Windows 95 and NT). ODBCINST.INI holds the information about different ODBC DBMS drivers installed on your system, including SQL Anywhere. ODBC.INI holds information about available data sources.

You can look at and alter the information in these files using the ODBC Administrator. The SQL Anywhere installation for Windows operating systems installs the ODBC Administrator as one of the icons in your SQL Anywhere group.

Files needed for ODBC connections

The files needed for ODBC client applications to connect to a SQL Anywhere database are as follows for the case of a Windows 95 or NT setup:



Under Windows and Windows NT, ODBC-enabled client applications call the ODBC driver manager. For Windows 3.x, the ODBC driver manager is ODBC.DLL, installed in your Windows SYSTEM subdirectory; for Windows 95 and NT this is ODBC32.DLL, installed in your NT SYSTEM32 subdirectory or your Windows 95 SYSTEM subdirectory. A third-party driver manager is available from Intersolv, Inc., providing similar functionality for OS/2.

Each ODBC-supporting DBMS, including SQL Anywhere, supplies its own ODBC driver, which is called by the ODBC driver manager. Each ODBC driver is a DLL.

The SQL Anywhere ODBC driver

The SQL Anywhere ODBC driver is installed as part of your SQL Anywhere setup.

- For Windows 3.x, this driver is WOD50W.DLL. The SQL Anywhere installation places the WOD50W.DLL file in the WIN subdirectory of your SQL Anywhere installation directory.
- For Windows 95 and NT, the driver is WOD50T.DLL. The SQL Anywhere installation places the WOD50T.DLL file in theWIN 32 subdirectory.
- For OS/2, the driver is WOD502.DLL. The SQL Anywhere installation places the WOD502.DLL file in the OS2 subdirectory.

The location of the SQL Anywhere ODBC driver is recorded in ODBCINST.INI, which is kept in your Windows directory or in the registry.

Windows and DOS default character set

The default Windows character set differs in some cases from that used by DOS in the default (code page 437). The Windows character set is sometimes called the ANSI character set, and the DOS code page is sometimes called the OEM character set.

A translation DLL can be used to convert characters from the ANSI character set (used by a Windows application) to the default character set in use by the database (code page 437), and vice versa. This translation DLL is:

- For Windows 3.x, WTR50W.DLL
- For Windows 95 or NT, WTR50T.DLL
- For OS/2, WTR502.DLL

OEM to ANSI character set translation

OEM to ANSI character set translation does not affect the alphabetic and numeric characters. It does affect some graphics characters that occupy higher positions in the collation.

Additional files required by the SQL Anywhere ODBC driver

The SQL Anywhere ODBC driver requires additional files. Under Windows, DBL50W.DLL needs to be in the DOS path, under Windows 95 or NT, DBL50T.DLL is required; and under OS/2; DBL502.DLL is required. These files are installed in the Windows, NT, or OS2 subdirectories of your SQL Anywhere installation directory.

A language DLL is also required, as specified in the SQLANY.INI file or registry. For Windows 3.x, SQLANY.INI should be in your Windows directory; for Windows 95 and NT SQLANY.INI is a registry. The SQL Anywhere installation does not install SQLANY.INI; but it is created when you run ISQL or Sybase Central. The default language DLL is WL50EN.DLL (English); other languages must be specified separately.

The ODBCINST.INI file

A file or registry entry named ODBCINST.INI holds information about all the ODBC drivers on the computer. When SQL Anywhere is installed, it adds a description of the SQL Anywhere ODBC driver to ODBCINST.INI, which is held in the Windows 3.x **system** subdirectory, and in the Windows 95 or NT registry. If you start the ODBC Administrator, and click the Driver button, you should see Sybase SQL Anywhere 5.0 listed among the installed ODBC drivers.

You should not have to make any modifications concerning the SQL Anywhere driver.

SQL Anywhere driver description in Windows 3.x ODBCINST.INI

The description of the SQL Anywhere driver in the Windows 3.x ODBCINST.INI includes the following:

```
[ODBC Drivers]
Sybase SQL Anywhere 5.0=Installed

[ODBC Translators]
Sybase SQL Anywhere 5.0 Translator=Installed

[Sybase SQL Anywhere 5.0]
Driver=c:\sqlany50\win\wod50w.dll
Setup=c:\sqlany50\win\wod50w.dll

[Sybase SQL Anywhere 5.0 Translator]
Translator=c:\sqlany50\win\wtr50w.dll
Setup=c:\sqlany50\win\wtr50w.dll
```

The corresponding information is held in the Windows 95 or NT registry.

Working with ODBC data sources

For each database that you want to access from an ODBC client application on your computer, you need to enter a Data Source in the ODBC.INI file or registry. In Windows and Windows NT, you can enter the Data Source description using the ODBC Administrator. The information contained about each Data Source in ODBC.INI includes several of the connection parameters needed to connect to a database.

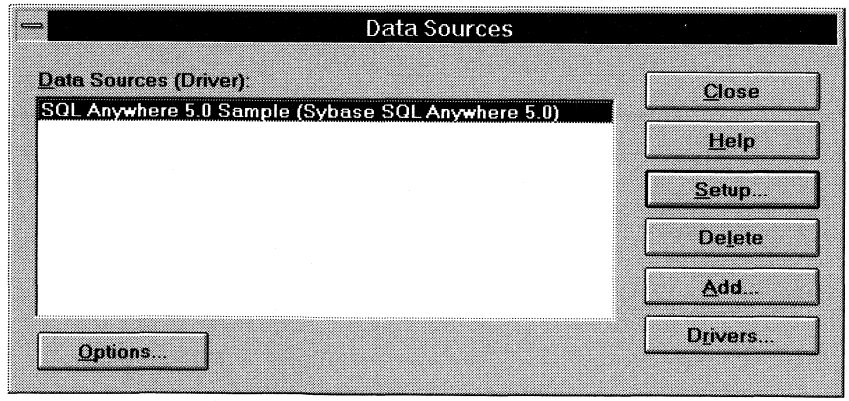
Some application development systems, such as PowerBuilder, add this information automatically if you create a data source from within the system. In this case you do not need to use the ODBC Administrator.

The SQL Anywhere sample database information is automatically installed into this file during SQL Anywhere setup. You can see the Data Source information for the sample database using the ODBC Administrator.

If you are having trouble connecting to a SQL Anywhere database from an ODBC-enabled client application, check that the information in the Data Source definition is correct.

Using the ODBC Administrator

Although you can add, remove and modify data source information by directly editing the ODBC.INI file in Windows or by using the registry editor in Windows NT, it is much easier to use the ODBC Administrator program.



The left side of the Administrator window lists the available data sources including the SQL Anywhere sample database.

If you have other ODBC software installed on your computer, you may have other data sources available. Pressing the Drivers button will display a list of the currently installed ODBC drivers, and allow you to install new ODBC drivers or remove drivers.

The following actions are available for data sources:

- Adding a data source
- Modifying a data source
- Removing a data source

These are discussed in the following sections.

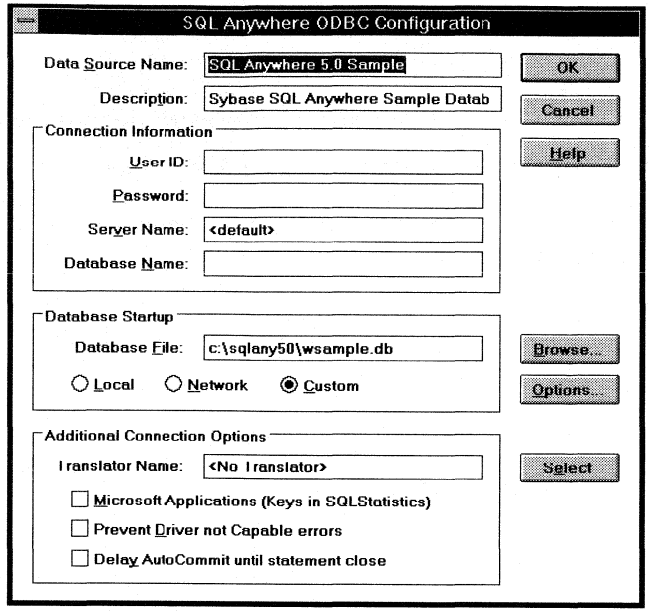
Adding an ODBC data source

The SQL Anywhere ODBC driver can access databases through local SQL Anywhere engines or SQL Anywhere network servers.

> **Database must exist**
> To add a data source for a database file, the database must already exist. See "Working with Database Objects" for information on creating a new database. You must create a database before using the ODBC Administrator program to add a data source for the database.

❖ **To add a new data source:**

1. If you want to add a new data source, click Add. A list of the available drivers is displayed.
2. Select the SQL Anywhere driver from the list and click OK. The SQL Anywhere ODBC Configuration window is displayed:



The SQL Anywhere ODBC Configuration window contains the following fields, which correspond to connection parameters in a connection string.

For a description of connection parameters and a description of the manner in which they are used to establish a connection with a database, see "Database connection parameters" on page 162.

ODBC Configuration dialog box - field descriptions

The following table lists the fields in the SQL Anywhere ODBC Configuration window.

Field	Description
Data Source Name	This should be a short name for the data source, such as Orders or Accounts Payable
Description	A longer description of the data source
User ID	(Optional) The user name to be used when connecting. If the user ID is omitted, most applications prompt you for a user ID and password when connecting to the data source
Password	(Optional) The password for the supplied User ID. Since the password supplied is stored in ODBC.INI, setting the password here may be a security risk. If the password is omitted, most applications prompt you to enter your password when connecting to the data source

Field	Description
Server Name	The name of a SQL Anywhere database engine or server. If not specified, the default engine is used. This field corresponds to the EngineName connection parameter
Database Name	If specified, this corresponds to the name of a database already running on a SQL Anywhere database engine or SQL Anywhere network server. This field corresponds to the DatabaseName connection parameter.
Database File	If specified, this contains the name of a database file—such as C:\sqlany50\sademo.db. You can use the Browse button to locate a database file name to place in this field. This field corresponds to the DatabaseFile connection parameter.
Local, Network, Custom	The command used to run the database software when the named database engine or server is not already executing. You can select Local or Network, as appropriate, if the default settings are satisfactory. Otherwise, select Custom and enter the command including any command line parameters by pressing the Options button
Microsoft Applications (Keys in SQLStatistics)	The ODBC specification states that primary and foreign keys should not be returned by SQLStatistics. Some programs (including Microsoft Visual Basic V3.0 and Microsoft Access V1.0 and V1.1) assume that primary and foreign keys are returned by SQLStatistics. Checking this option makes the SQL Anywhere ODBC driver mimic the required behavior so these applications work properly
Prevent Driver not Capable Errors	The SQL Anywhere ODBC driver returns a "Driver not Capable" error code because it does not support qualifiers. Some ODBC applications do not handle this error properly. Checking this box disables this error code, allowing these applications to work.
Delay AutoCommit until statement close	The ODBC standard specifies that the default mode of operation is to run in auto-commit mode (each statement is committed immediately after it is executed). Some applications do not provide a way for users to override this behaviour. Checking this option causes the SQL Anywhere ODBC driver to delay the commit operation until the statement is closed.

Windows ODBC.INI

The following example shows the description of the SQL Anywhere sample database and a sample SQL Anywhere client in the Windows ODBC.INI:

```
[ODBC Data Sources]
SQL Anywhere 5.0 Sample=Sybase SQL Anywhere 5.0
```

```
[SQL Anywhere 5.0 Sample]
driver=c:\sqlany50\win\wod50w.dll
description=Sybase SQL Anywhere Sample Database
DatabaseFile=c:\sqlany50\sademo.db
Start=c:\sqlany50\win\dbeng50w -d

[SQL Anywhere 5.0 Sample Client]
driver=c:\sqlany50\win\wod50w.dll
description=SQL Anywhere Client/Server
EngineName=place_server_name_here
Start=c:\sqlany50\win\dbclienw
```

Some sample ODBC data sources

This section presents some sample SQL Anywhere ODBC data source descriptions. The descriptions are presented as settings in the SQL Anywhere ODBC Configuration window.

The parameters Description, User ID, and Password are not described here. The Description parameter is not required, and the User ID and Password can be provided at connection time from the client application.

The examples describe a data source running the sample database (SADEMO.DB), running as a database named sademo on an engine named SampleServer.

Sample data source for connecting to a running engine

The following settings describe an ODBC data source for a standalone database engine running on the local machine. The database is assumed to be running when the application connects; there is no provision in this data source description for starting up a database engine or for loading a database on the engine.

Parameter	Setting
Data Source Name	Sample Name
Server Name	SampleServer
Database Name	sademo
Database File	No entry
Local / Network / Custom option	Local
Translator	No entry

Sample data source for a named database file

The following settings describe an ODBC data source for a named database file. The description loads the database file on the default local database engine, or connects to a database with the name sademo if it is already running.

Parameter	Setting
Data Source Name	Sample Name
Server Name	No entry
Database Name	No entry
Database File	C:\SQLANY50\SADEMO.DB
Local / Network / Custom option	Local
Translator	No entry

Sample data source for a network server

The following settings describe an ODBC data source for a database running on a network server. The database is assumed to be running when the application connects; there is no provision in this data source for starting up a database server or for loading a database on the server.

To use this data source you need the SQL Anywhere Client, which is installed with the SQL Anywhere Network Server.

> **PowerBuilder and InfoMaker users**
> This chapter contains some information about connecting to a SQL Anywhere network server. The SQL Anywhere network server is not included with PowerBuilder and InfoMaker.

Parameter	Setting
Data Source Name	Sample Name
Server Name	SampleServer
Database Name	sademo
Database File	No entry
Local / Network / Custom option	Network
Translator	No entry

Modifying an existing ODBC data source

❖ **To modify an existing data source:**

1 Select the data source in the ODBC Administrator.

2 Press the **Setup** button. You can modify any of the attributes set when the data source was added.

For a description of the attributes, see "Adding an ODBC data source" on page 174.

Removing an ODBC data source

❖ **To remove an ODBC data source:**

1 Select the data source in the ODBC Administrator.

2 Press the Delete button. You are prompted to confirm the deletion.

Database file is not deleted
Removing a data source does not delete the database file. It simply deletes the description of the data source from the ODBC.ini file or Windows NT registry. It can be added back as described above.

Using ODBC under OS/2

An ODBC Administrator program and an ODBC driver manager are available for OS/2 and can be acquired from INTERSOLV, Inc. It provides the same functionality as the ODBC Administrator program and ODBC.DLL driver manager under Windows or Windows NT. The Administrator program is not supplied with SQL Anywhere.

Even if you are not using the ODBC Administrator, the file ODBC.INI is still used to describe the available data sources. Under OS/2, ODBC.INI is a binary file. As such, it is not easily editable. The SQL Anywhere installation creates this file with a definition of the sample data source. The installation also includes an OS/2 command file, ODBCINI.CMD that uses the REXX API to modify the file. This command file can be modified and run to create additional data sources.

Using ODBC under DOS and QNX

The file ODBC.INI is still used to describe the available data sources. This file must be located somewhere in your path. The following example contains a data source definition for the SQL Anywhere sample database:

```
[ODBC Data Sources]
SQL Anywhere 5.0 Sample=Sybase SQL Anywhere 5.0
[SQL Anywhere 5.0 Sample]
Driver=C:\sqlany50\win\wod50w.dll
Description=Sybase SQL Anywhere Sample Database
DatabaseFile=C:\sqlany50\sademo.db
Start=C:\sqlany50\dos\dbeng50
```

The first section of the file ([ODBC Data Sources]) lists all of the currently defined data sources. In this case there is only one, called SQL Anywhere 5.0 Sample.

Fields in the second section of SQL Anywhere 5.0 Sample file

The second section of the file ([SQL Anywhere 5.0 Sample]) describes the data source named SQL Anywhere 5.0 Sample. This section contains the following fields:

Fields	Description
Description	A longer description of the data source
UID	(Optional) The user name to be used when connecting. If it is omitted, most ODBC applications will prompt you for a User ID and Password when connecting to the data source
PWD	(Optional) The password for the supplied User ID. Setting the password here may be a security risk because it is easily viewed. If the password is omitted, most applications will prompt you to enter your password when connecting to the data source
DatabaseFile	The name of a SQL Anywhere database file
Start	The command used to run the database software when the named database is not already executing. It contains the command including any command line parameters

CHAPTER 16

Designing Your Database

About this chapter

SQL Anywhere is a relational database management system. This chapter introduces the basic concepts of relational databases and gives you step-by-step suggestions for designing your database.

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

This chapter is not specific to SQL Anywhere. If you already have a database, you may wish to go to the next chapter.

Contents

Topic	Page
Relational database concepts	182
Planning the database	185
The design process	187
Designing the database table properties	199

This chapter covers database design in a cursory manner. For more information, you may wish to consult an introductory book such as *A Database Primer* by C. J. Date. If you are interested in pursuing database theory, C. J. Date's *An Introduction to Database Systems* is an excellent textbook on the subject.

Relational database concepts

This section introduces some of the terms and concepts that are important in talking about relational databases.

Database tables

In a relational database, all data is held in **tables**, which are made up of **rows** and **columns**.

Each table has one or more columns, and each column is assigned a specific **data type**, such as an integer number, a sequence of characters (for text), or a date. Each row in the table has a value for each column.

A typical fragment of a table containing employee information may look as follows:

emp_ID	emp_lname	emp_fname	emp_phone
10057	Huong	Zhang	1096
10693	Donaldson	Anne	7821

Characteristics of database tables

The tables of a relational database have some important characteristics:

- There is no significance to the order of the columns or rows
- Each row contains one and only one value for each column
- No two rows contain the same set of values
- Each value for a given column is of the same type

The following table lists some of the formal and informal relational database terms describing tables and their contents, together with their equivalent in other nonrelational databases. This manual uses the informal terms.

Formal	Informal	Equivalent
relational term	relational term	nonrelational term
Relation	Table	File
Attribute	Column	Field
Tuple	Row	Record

Keys in relational databases

Primary and foreign keys enable each row in the database tables to be identified, and enable relationships between the tables to be defined. These keys define the relational structure of a database.

Each table has a primary key

Each table in a relational database has a **primary key**. The primary key is a column, or set of columns, that allows each row in the table to be uniquely identified. No two rows may have the same value of a primary key.

Examples

In a table holding information about employees, the primary key may be an ID number assigned to each employee.

In the sample database, the table of sales order items has the following columns:

- An order number, identifying the order the item is part of.
- A line number, identifying each item on any order.
- A product ID, identifying the product being ordered.
- A quantity, showing how many items were ordered.
- A ship date, showing when the order was shipped. In order to identify a particular item, both the order number and the line number are required. The primary key is made up of both these columns.

Tables are related by foreign keys

The information in one table is related to that in other tables by **foreign key** relations.

Example

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

- dept_id (an ID number for the department that is the primary key for the table)
- dept_name (holding the name of the department)

To find out a particular employee's department, there is no need to put the name of the employee's department into the employee table. Instead, the employee table contains a column holding the department ID of the employee's department. This is called a **foreign key** to the department table. A foreign key references a particular row in the table containing the corresponding primary key.

In this example, the employee table (which contains the foreign key in the relationship) is called the foreign table or referencing table. The department table (which contains the referenced primary key) is called the primary table or the referenced table.

If a primary key is not assigned, the combination of all columns in the table becomes the primary key. This can lead to a very large transaction log.

Other database objects

A relational database holds more than a set of related tables. Among the other objects that make up a SQL Anywhere relational database are:

- **The index** Indexes allow quick lookup of information
- **The view** Views are computed tables
- **The stored procedure and the trigger** These are routines held in the database itself that act on the information in the database

All these objects are in some way built on top of the base tables that hold the information. A **base table** is a table that is stored permanently in the database. This chapter discusses only how to decide what tables you need and what each table needs to hold.

Planning the database

In designing a database you plan what tables you require and what data they will contain. You also determine how the tables are related.

You must determine what things you want to store information *about* (each one is an **entity**) and how these things are *related* (by a **relationship**). A useful technique in designing your database is to draw a picture of your tables. This graphical display of a database is called an Entity-Relationship (E-R) diagram. Usually, each box in an E-R diagram corresponds to a table in a relational database, and each line from the diagram corresponds to a foreign key.

Entity-Relationship design

Entity-Relationship design (E-R design) is an example of top-down design of databases. There are now sophisticated methods and tools available to pursue E-R design of databases in great detail. This chapter is an introductory chapter only, but it does contain enough information for the design of fairly straightforward databases.

Entity

Each table in the database describes an **entity**; it is the database equivalent of a noun. Employees, order items, departments and products are all examples of entities represented by a table in a database. The entities that you build into your database arise from the activities for which you will be using the database, whether that be tracking sales calls, maintaining employee information, or some other activity.

Relationship

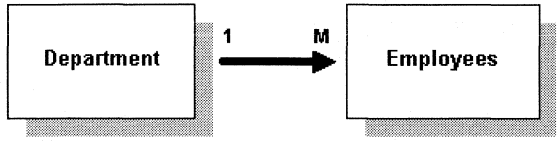
A **relationship** between entities is the database equivalent of a verb. An employee is associated with a department, or an office is located in a city. Relationships in a database may appear as foreign key relationships between tables, or may appear as separate tables themselves. We will see examples of each in this chapter.

The relationships in the database are an encoding of rules or practices governing the data in the table. If each department has one department head, then a single column can be built into the department table to hold the name of the department head. When these rules are built into the structure of the database, there is no provision for exceptions: there is nowhere to put a second department head, and duplicating the department entry would involve duplicating the department ID, which is the primary key. This is not allowed.

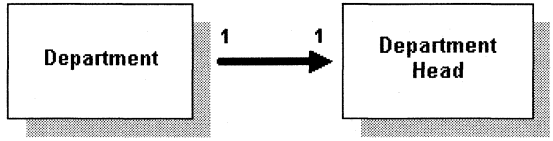
Relationships between tables

There are three kinds of relationships between tables:

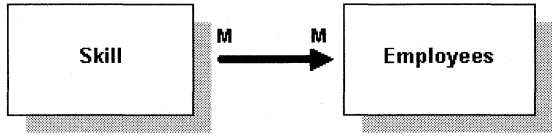
- One-to-many relationships



- One-to-one relationships



- Many-to-many relationships



The design process

There are five major steps in the design process.

- Step 1: identify entities and relationships
- Step 2: identify the required data
- Step 3: normalize the data
- Step 4: resolve the relationships
- Step 5: verify the design

For information about implementing the database design, see the chapter "Working with Database Objects".

Step 1: identify entities and relationships

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

1. Define high-level activities. Identify the general activities you will use this database for. For example, you may want to keep track of information about employees.
2. Identify entities. For the list of activities, identify the subject areas you need to maintain information about. These will become tables. For example, hire employees, assign to a department, and determine a skill level.
3. Identify relationships. Look at the activities and determine what the relationships will be between the tables. For example, there is a relationship between departments and employees. We give this relationship a name.
4. Break down the activities. You started out with high-level activities. Now examine these activities more carefully to see if some of them can be broken down into lower-level activities. For example, a high-level activity such *as maintain employee information* can be broken down into:
   - Add new employees
   - Change existing employee information
   - Delete terminated employees

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

Example

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

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

Define high-level activities

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

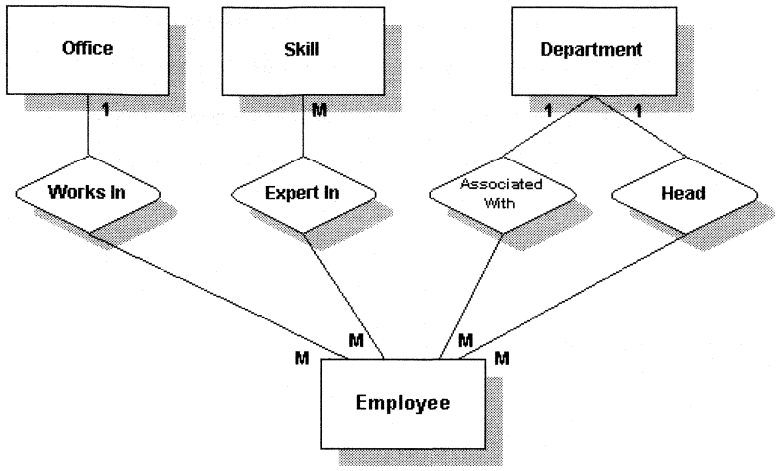
- Hire employees
- Terminate employees
- Maintain personal employee information
- Maintain information on skills required for the company
- Maintain information on which employees have which skills
- Maintain information on departments
- Maintain information on offices

Identify the entities and relationships

We can identify the subject areas (tables) and relationships that will hold the information and create a diagram based on the description and high-level activities.

We use boxes to show tables and diamonds to show relationships. We can also identify which relationships are one-to-many, one-to-one, and many-to-many.

This is a rough E-R diagram. It will be refined throughout the chapter.



Break down the high-level activities

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

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

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

Identify business rules

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

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

- There are now five offices; expansion plans allow for a maximum of 10.
- Employees can change department or office
- Each department has one department head

- Each office has a maximum of three telephone numbers
- Each telephone number has one or more extensions
- When an employee is hired, the level of expertise in each of several skills is identified
- Each employee can have from three to 20 skills
- An employee may or may not be assigned to an office

Step 2: identify the required data

❖ **To identify the required data:**

1 Identify supporting data.

2 List all the data you will need to keep track of. The data that describes the table (subject) answers the questions who, what, where, when, and why.

3 Set up data for each table.

4 List the available data for each table as it seems appropriate right now.

5 Set up data for each relationship.

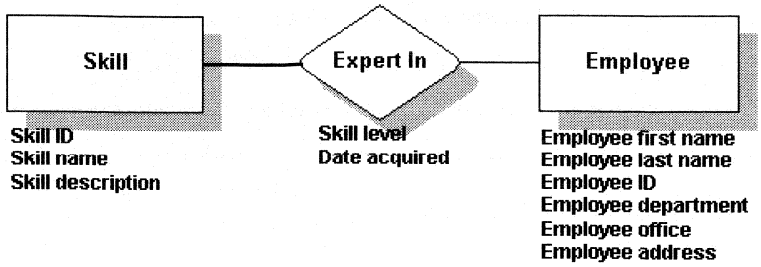
6 List the data that applies to each relationship (if any).

Identify supporting data

The supporting data you identify will become the names of the columns in the table. For example, the data below might apply to the Employee table, the Skill table, and the Expert In table:

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

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



Things to remember

- When you are identifying the supporting data, be sure to refer to the activities you identified earlier to see how you will need to access the data.
- For example, if you know that you will need a list of all employees sorted by last name, make sure that you specify supporting data as Last name and First name, rather than simply Name (which would contain both first and last names).
- The names you choose should be consistent. Consistency makes it easier to maintain your database and easier to read reports and output windows.
- For example, if you choose to use an abbreviated name such as Emp_status for one piece of data, you should not use the full name (Employee_ID) for another piece of data. Instead, the names should be Emp_status and Emp_ID.
- It is not crucial that the data be associated with the correct table. You can use your intuition. In the next section, you'll apply tests to check your judgment.

Step 3: normalize the data

Normalization is a series of tests you use to eliminate redundancy in the data and make sure the data is associated with the correct table or relationship. There are five tests. In this section, we will talk about the three tests that are usually used.

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

Normal forms

Normal forms are the tests you usually use to normalize data. When your data passes the first test, it is considered to be in first normal form, when it passes the second test, it is in second normal form, and when it passes the third test, it is in third normal form.

❖ **To normalize the data:**

1 List the data:

2 Identify at least one key for each table. Each table must have a primary key.

3 Identify keys for relationships. The keys for a relationship are the keys from the two tables it joins.

4 Check for calculated data in your supporting data list. Calculated data is not normally stored in the database.

5 Put data in first normal form:

6 Remove repeating data from tables and relationships.

7 Create one or more tables and relationships with the data you remove.

8 Put data in second normal form:

9 Identify tables and relationships with more than one key.

10 Remove data that depends on only one part of the key.

11 Create one or more tables and relationships with the data you remove.

12 Put data in third normal form:

13 Remove data that depends on other data in the table or relationship and not on the key.

14 Create one or more tables and relationships with the data you remove.

Data and keys

Before you begin to normalize (test your data), simply list the data and identify a unique **primary key** for each table. The key can be made up of one piece of data (column) or several (a concatenated key).

The primary key is the set of columns that uniquely identifies rows in a table. The primary key for the Employee table is the Employee ID column. The primary key for the Works In relationship consists of the Office Code and Employee ID columns. Give a key to each relationship in your database by taking the key from each of the tables it connects. In the example, the keys identified with an asterisk are the keys for the relationship:

Relationship	Key
Office	*Office code
	Office address
	Phone number
Works in	*Office code

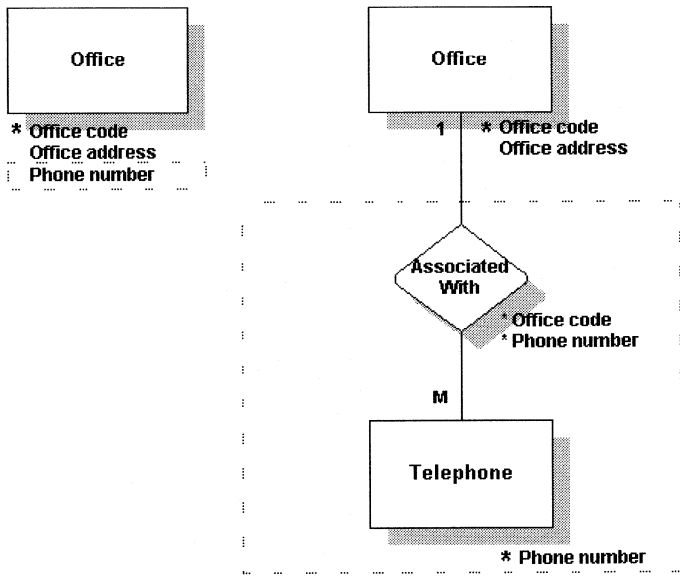
Relationship	Key
	*Employee ID
Department	*Department ID
	Department name
Heads	*Department ID
	*Employee ID
Assoc with	*Department ID
	*Employee ID
Skill	*Skill ID
	Skill name
	Skill description
Expert in	*Skill ID
	*Employee ID
	Skill level
	Date acquired
Employee	*Employee ID
	Employee last name
	Employee first name
	Social security number
	Employee street
	Employee city
	Employee state
	Employee phone
	Date of birth

Putting data in first normal form

- Remove repeating groups.
- To test for first normal form, remove repeating groups and put them into a table of their own.
- In the example below, Phone number can repeat. (An office can have more than one telephone number.) Remove the repeating group and make a new table called Telephone. Set up a relationship called Associated With between Telephone and Office.

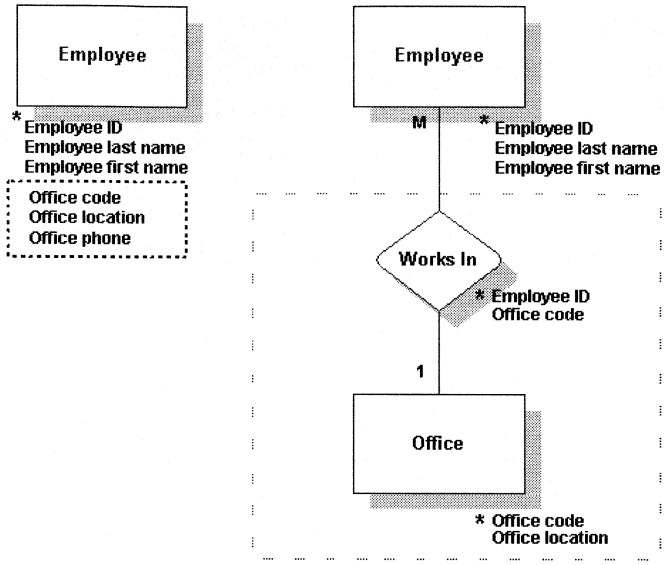
Putting data in second normal form

- Remove data that does not depend on the whole key.
- Look only at tables and relationships that have more than one key. To test for second normal form, remove any data that does not depend on the whole key (all the columns that make up the key).
- In this example, the original Employee table specifies a key composed of two columns. Some of the data does not depend on the whole key; for example, the department name depends on only one of those keys (Department ID). Therefore, the Department ID, which the other employee data does not depend on, is moved to a table of its own called Department, and a relationship called Assigned To is set up between Employee and Department.



Putting data in third normal form

- Remove data that doesn't depend directly on the key.
- To test for third normal form, remove any data that depends on other data rather than directly on the key.
- In this example, the original Employee table contains data that depends on its key (Employee ID). However, data such as office location and office phone depend on another piece of data, Office code. They do not depend directly on the key, Employee ID. Remove this group of data along with Office code, which it depends on, and make another table called Office. Then we will create a relationship called Works In that connects Employee with Office.

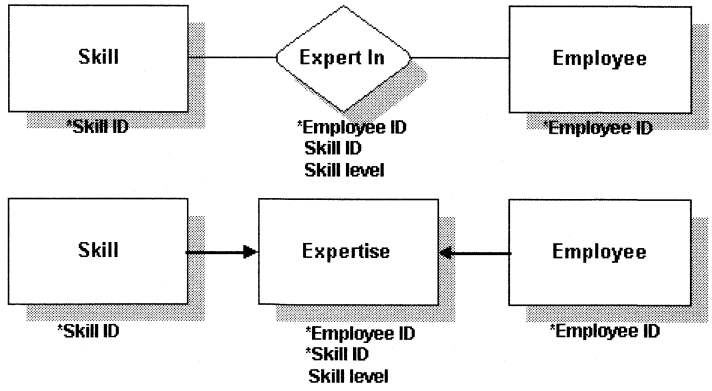


Step 4: resolve the relationships

When you finish the normalization process, your design is almost complete. All you need to do is resolve the relationships.

Resolving relationships that carry data

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



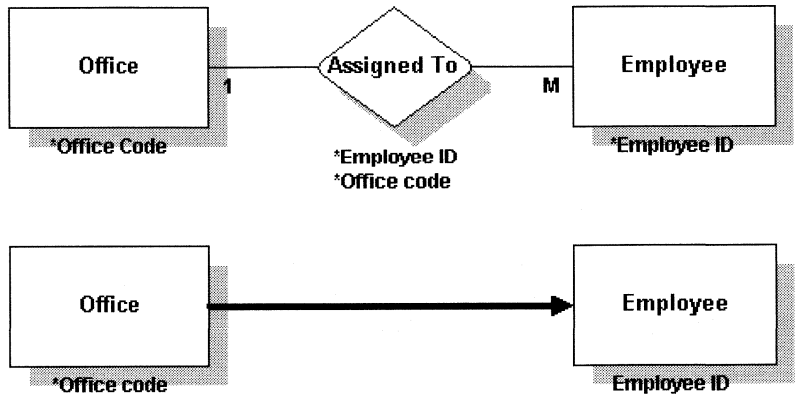
When this is the case, change the relationship to a table. The key to the new table remains the same as it was for the relationship.

Resolving relationships that do not carry data

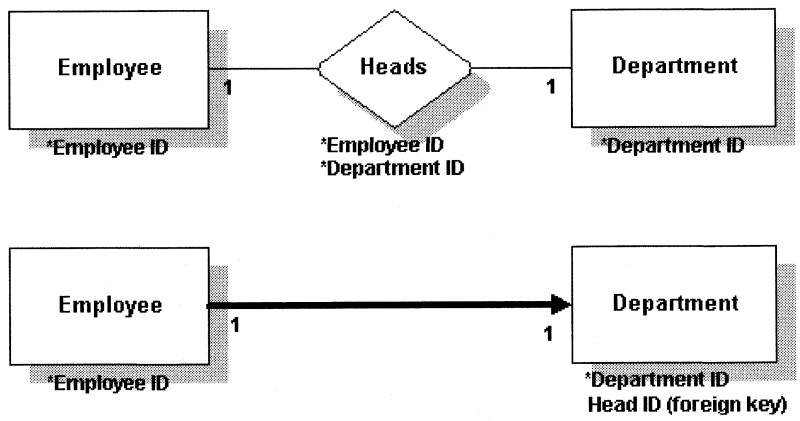
In order to implement relationships that do not carry data, you need to define foreign keys. A **foreign key** is a column or set of columns that contains primary key values from another table. The foreign key allows you to access data from more than one table at one time.

There are some basic rules that help you decide where to put the keys:

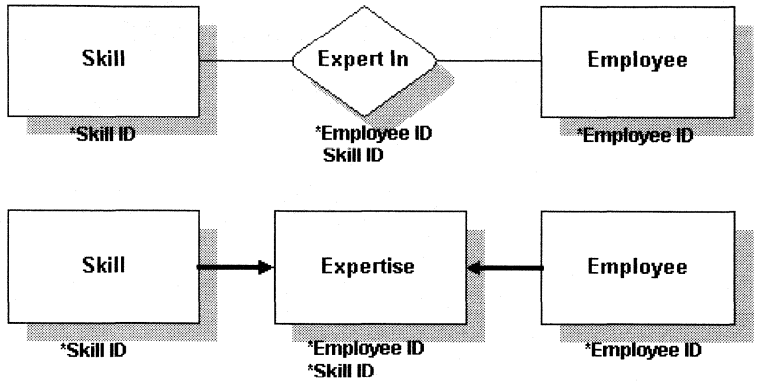
One to many In a one-to-many relationship, the primary key in the one is carried in the many. In this example, the foreign key goes into the Employee table.



One to one In a one-to-one relationship, the foreign key can go into either table. If it is mandatory on one side, but not on the other, it should go on the mandatory side. In this example, the foreign key (Head ID) is in the Department table because it is mandatory there.



Many to many In a many-to-many relationship, a new table is created with two foreign keys. The existing tables are now related to each other through this new table.



Step 5: verify the design

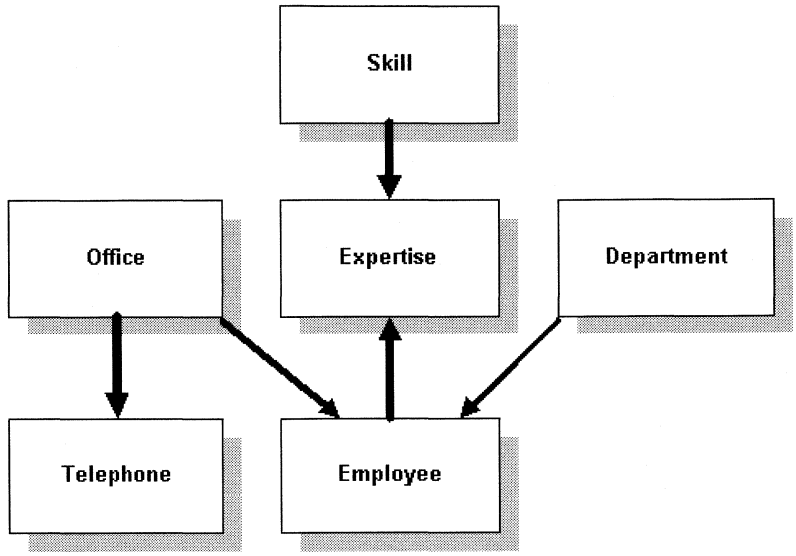
Before you implement your design, you need to make sure it supports your needs. Examine the activities you identified at the start of the design process and make sure you can access all the data the activities require:

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

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

Final design

The final design of the example looks like this:



For more information about creating a database in PowerBuilder or InfoMaker, see the PowerBuilder *User's Guide* or InfoMaker *User's Guide*.

Designing the database table properties

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

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

Choosing column names

Column names can be any set of letters, numbers or symbols. However, if the column name contains characters other than letters, numbers, or underscores, or it does not begin with a letter, or it is a keyword (see "Alphabetical list of keywords" in the chapter "SQL Anywhere Keywords"), then whenever the column name is used, it must be enclosed in double quotes.

Choosing data types for columns

The data types supported by SQL Anywhere include:

- Integer data types (int, integer, smallint)
- Decimal data types (decimal, numeric)
- Floating-point data types (float, double)
- Character data types (char, varchar, long varchar)
- Binary data types (binary, long binary)
- Date/time data types (date, time and timestamp)
- User-defined data types

For a description of data types, see the section "SQL Anywhere Data Types". The data type of the column affects the maximum size of the column. For example, if you specify SMALLINT, a column can contain a maximum value of 32,767. If you specify INTEGER, the maximum value is 2,147,483,647. In the case of CHAR, the maximum length of a value in the column must be specified.

The long binary data type can be used to hold information such as images (for instance, stored as bitmaps) or word-processing documents in a database. These types of information are commonly called binary large objects, or BLOBS.

For a complete description of each data type, see "SQL Anywhere Data Types".

NULL and NOT NULL

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

For a complete description of the NULL value, see "NULL value" in the chapter "Watcom-SQL Statements". For information on its use in comparisons, see "Search conditions" in the chapter "Watcom-SQL Language Reference".

Choosing constraints

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

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

For more information

For more information about valid conditions, see "Search conditions" in the chapter "Watcom-SQL Language Reference". For more information about assigning constraints to tables and columns, see the chapter "Ensuring Data Integrity".

Example

The sample database has a table called department, which has columns named dept_id, dept_name, and dept_head_id. Its definition is as follows:

Column	Data Type	Size	Null/Not Null	Constraint
dept_id	integer	—	not null	None
dept_name	char	40	not null	None
dept_head_id	integer	—	not null	None

Notice that "not null" is specified for each column. In this case, data for all columns is required for every row in the table.

Choosing primary and foreign keys

The primary key is the column or columns that uniquely identify the rows in the table. If your tables are properly normalized, a primary key should be defined as part of the database design.

A foreign key is a column or set of columns that contains primary key values from another table. Foreign key relationships build one-to-one and one-to-many relationships into your database. If your design is properly normalized, foreign keys should be defined as part of your database design.

CHAPTER 17

Working with Database Objects

About this chapter

This chapter describes the mechanics of creating, altering, and modifying database objects. The set of definitions of the database objects form the database schema: you can think of the schema as the database without any data.

This chapter also discusses creating and working with other database objects, (such as indexes and views).

Procedures and triggers are also database objects, and are discussed in the chapter "Using Procedures, Triggers, and Batches".

Contents

Topic	Page
Using Sybase Central to work with database objects	204
Using ISQL to work with database objects	205
Working with databases	206
Working with tables	211
Working with views	217
Working with indexes	223

Using Sybase Central to work with database objects

Sybase Central is the primary tool for working with database objects in SQL Anywhere. You can use SQL Anywhere to create, modify, and delete all kinds of database objects, including tables, procedures, triggers, views, indexes, users and groups, and so on.

This chapter is concerned with the SQL statements for working with database objects. If you are using Sybase Central, these SQL statements are generated for you. The primary source of information about Sybase Central is the Sybase Central online Help. In this chapter, only brief pointers are given for tasks you can carry out using Sybase Central.

For an introduction to using Sybase Central, see the chapter "Managing Databases with Sybase Central".

Using ISQL to work with database objects

ISQL is a utility for entering SQL statements, supplied with SQL Anywhere. If you are using ISQL to work with your database schema, instead of executing the SQL commands one at a time, you should build up the set of commands in an ISQL command file. This file can then be executed in ISQL to build the database.

If you are using a tool other than ISQL, all the information in this chapter concerning SQL statements still applies.

If you have not created your database using command files, you can create a command file that would recreate your database by unloading the database.

For a description of the DBUNLOAD utility, see "The Unload utility" in the chapter "SQL Anywhere Components".

ISQL command file

An ISQL command file is a text file with semicolons placed at the end of commands (see the chapter "Command Files") as shown below.

```
CREATE TABLE t1 ( .. );
CREATE TABLE t2 ( .. );
CREATE INDEX i2 ON t2 ( .. );
..
```

An ISQL command file is usually given a name with the extension SQL. To create your database using a command file, you can either paste it into the ISQL command window (for files with less than 500 lines), or execute the saved command file by entering a command to read the file into the ISQL command window. For example:

```
read makesdb
```

will read the ISQL commands in the file MAKEDB.SQL.

Working with databases

Some application design systems, such as Powersoft PowerBuilder, contain facilities for creating databases. These tools construct SQL statements that are submitted to the SQL Anywhere engine, typically through its ODBC interface. If you are using one of these tools, you do not need to construct SQL statements to create tables, assign permissions, and so on.

This chapter describes the SQL statements supported by SQL Anywhere. You can use these statements directly if you are building your database from an Interactive SQL tool, such as ISQL. Even if you are using an application design tool, you may want to use SQL statements to add features to the database if they are not supported by the system's database design tool.

Initializing a database

Initializing a database creates the root file for storing your database and the system tables, which hold the schema definition as you build your database.

Database files are compatible among all versions of SQL Anywhere. A database created from any operating system can be used from another operating system by copying the database file(s). A database created with a standalone database engine can be used with any SQL Anywhere network server as long as the server is not an earlier release.

You create a database using the database initialization utility. Database initialization is not controlled by SQL statements. Once the database is initialized, you can connect to it and build the objects in the database using SQL statements. If you are using Sybase Central, the SQL statements are constructed for you and you do not have to enter them yourself.

Accessing the initialization utility

A full description of the initialization utility, with the options available when you create a database is given in "The Initialization utility" in the chapter "SQL Anywhere Components". The initialization utility can be accessed in the following ways:

- In Sybase Central, click the Database Utilities folder in the left panel, then double-click Create Database to start the Create Database Wizard, which leads you through the process.

- For Windows 3.x and OS/2, use the ISQL Database Tools window. To open this window, select the Database Tools menu item from the Window menu. Select Create Database from the Tools list, and enter the path and name of the file in which you wish to store the database in the Database File field. It is recommended that you give the database file the standard filename extension of DB. Press the Create button. This displays another dialog box for specifying options such as page size and collation sequence.
- Use the DBINIT command (DBINITW for Windows). For a full description of the DBINIT command, see "The Initialization utility" in the chapter "SQL Anywhere Components".

  For example, the following command will create a database called COMPANY.DB:

  ```
  dbinit company.db
  ```
- Command line parameters allow different options for the database. For example, the following command creates a database with a 4K page size:

  ```
  dbinit -p 4096 company.db
  ```
- Use the ISQL DBTOOL statement. The database utility programs are available from within ISQL using the DBTOOL command. For example, the following command creates a database called COMPANY.DB:

  ```
  DBTOOL CREATE DATABASE 'company.db'
  ```
- The following command creates a database with 4K pages:

  ```
  DBTOOL CREATE DATABASE 'company.db' PAGE SIZE
  4096
  ```

Adding database files

When a database is initialized, it is composed of one file. This first database file is called the **root file**. All database objects and all data are placed in the root file. For many databases, it is convenient to keep the database as a single file. This section is intended only for users of large databases.

Each SQL Anywhere database file has a maximum size of 2 GB, so you may wish to divide large databases among more than one file. (On Windows NT, this limitation is removed, and files can be up to a terabyte). You create a new database file, or **dbspace**, using the CREATE DBSPACE statement. A new dbspace may be on the same disk drive as the root file or on another disk drive. You must have DBA authority to create new database files.

When created, a new dbspace has no contents. When you create a new table you can place it in the new dbspace with an IN clause in the CREATE TABLE statement. For information on creating tables, see "Creating tables" on page 211. If no IN clause is used, the table is placed in the root file. Each table must be contained in a single dbspace, and SQL Anywhere has a maximum of twelve dbspaces per database. By default, indexes are placed in the same dbspace as their table, but they can be placed in a separate dbspace by supplying an IN clause.

Example

The following command creates a new dbspace called **library** in the file LIBRARY.DB in the same directory as the root file:

```
CREATE DBSPACE library
AS 'library.db'
```

To create a table and place it in the library dbspace, you can use the following command:

```
CREATE TABLE Library_Books (
   title char(100),
   author char(50),
   isbn char(30)
   ) IN library
```

If you wish to split existing database objects among several dbspaces, you need to unload your database and modify the command file for rebuilding the database. To do so, add IN clauses to specify the dbspace for each table you do not wish to place in the root file.

Creating a dbspace in Sybase Central

❖ **To create a dbspace in Sybase Central:**

1. Connect to the database.
2. Click the DB Spaces folder for that database.
3. Double-click Add DB Space in the right panel.
4. Enter the dbspace name and filename
5. Click OK to create the dbspace.

Preallocating space for database files

SQL Anywhere automatically takes new disk space for database files as needed. Unless you are working with a large database with a high rate of inserts and deletes, you do not need to worry about explicitly allocating spaced for database files. SQL Anywhere does allow preallocation of disk space for database files or for transaction logs for those cases where rapidly changing database files could lead to excessive file fragmentation on the disk, and possible performance problems.

You can preallocate disk space for a database file or for the transaction log using the ALTER DBSPACE statement. For more information on this statement, see "ALTER DBSPACE statement" in the chapter "Watcom-SQL Statements".

For example, the following statement adds 200 pages to the database file with dbspace name **library**. (Thc database page size is fixed when the database is created.)

```
ALTER DBSPACE library
ADD 200
```

Running a disk defragmentation utility after preallocating disk space helps ensure that the database file is not fragmented over many disjoint areas of the disk drive. Performance can suffer if there is excessive fragmentation of database files.

Preallocating disk space in Sybase Central

❖ **To preallocate disk space for a dbspace in Sybase Central:**

1. Connect to the database.
2. Click the DB Spaces folder for that database.
3. Double-click the dbspace in the right panel.
4. Click Add Pages, and enter the number of database pages to preallocate.
5. Click OK.

Erasing a database

Erasing a database deletes all tables and data from disk, including the transaction log that records alterations to the database.

All SQL Anywhere databases are marked as read-only to prevent accidental modification or deletion of the database files.

☞ You can erase database files using the Erase utility. For a full description of the Erase utility, see "The Erase utility" in the chapter "SQL Anywhere Components".

Accessing the Erase utility

You can access the Erase utility using any of the following methods:

- **Using Sybase Central**. Click the Database Utilities folder, and double-click Erase Database to display the Erase Database Wizard, which leads you through the process.
- **Using the ISQL Database Tools window**. To open this window, select the Database Tools menu item from the Window menu. Select Erase Database or Write File from the Tools list, and enter the name of the database file in the Database File field. The database is erased when you press the Erase button.
- **Using the DBERASE command-line utility (DBERASEW for Windows).** The following command erases the database company.db and its transaction log:

  ```
  dberase company.db
  ```

  You will be asked to confirm that you really want to erase the files. To erase the files, type *y* and press ENTER.
- **Using the DBTOOL statement**. All the database utility programs are available from within ISQL using DBTOOL. For example, the following command erases the database company.db and its transaction log: DBTOOL DROP DATABASE company.db

☞ The Erase utility can also be used to erase write files and log files. For a description of write files, see "The Write File utility" in the chapter "SQL Anywhere Components".

Working with tables

When the database is initialized, the only tables in the database are the **system tables** which hold the database schema.

This section describes how to create, alter, and delete tables from a database The examples can be executed in ISQL, but the SQL statements are independent of the administration tool you are using.

You should create command files containing the CREATE TABLE and ALTER TABLE statements that define the tables in your database.

Creating tables

Creating tables in Sybase Central

To create a table in Sybase Central you first create an empty table, and then add columns, specify primary keys and constraints, and so on:

❖ **To create an empty table:**

1 Connect to the database.

2 Click the Tables folder for that database.

3 Double-click Add Table in the right panel.

4 Fill out the dialog box.

5 Click OK to create an empty table.

❖ **To add a column to a table:**

1 Double-click the new table.

2 Double-click the Columns folder.

3 Double-click Add Column.

4 Fill out the dialog box.

5 Click OK to the column.

Creating tables in ISQL

There are two ways to create new tables using ISQL:

- Type a CREATE TABLE statement in ISQL.
- Create a command file that contains the CREATE TABLE commands for your database. (This allows you easily to recreate your database and provides some documentation of the structure of your database)

This section describes how to create tables interactively using ISQL. The examples in this section use the sample database. To try the examples, run ISQL and connect to the sademo.db database with userid dba and password SQL.

For information on connecting to a database from ISQL, see "Connecting from the SQL Anywhere utilities" in the chapter "Connecting to a Database".

You can create tables from other tools in addition to ISQL. The SQL statements described here are independent of the tool you are using.

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

```
CREATE TABLE skill (
    skill_id INTEGER NOT NULL,
    skill_name CHAR( 20 ) NOT NULL,
    skill_type CHAR( 20 ) NOT NULL
    )
```

You can execute this command by typing it into the ISQL command window. and pressing the execute key (F9). Note the following:

- Each column has a **data type.** The skill_id is an integer (like 101), the skill_name is a character string containing up to 20 characters, and so on.
- All columns are mandatory as indicated by the phrase NOT NULL after their data types.

Before creating the table, SQL Anywhere makes all previous changes to the database permanent by internally executing the COMMIT statement. There is also a COMMIT after the table is created.

For a full description of the CREATE TABLE statement, see "CREATE TABLE statement" in the chapter "Watcom-SQL Statements". For information about building constraints into table definitions using CREATE TABLE, see the chapter "Ensuring Data Integrity".

Altering tables

This section describes how to change the structure of a table using the ALTER TABLE statement.

Example 1

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

```
ALTER TABLE skill
ADD skill_description CHAR( 254 )
```

This statement adds a column called skill_description that holds up to a few sentences describing the skill.

Example 2

Column attributes can also be modified with the ALTER TABLE statement. The following statement shortens the skill_description column of the sample database from a maximum of 254 characters to a maximum of 80:

```
ALTER TABLE skill
MODIFY skill_description CHAR( 80 )
```

Any current entries that are longer than 80 characters are trimmed to conform to the 80-character limit, and a warning is displayed.

Example 3

The following statement changes the name of the skill_type column to classification:

```
ALTER TABLE skill
RENAME skill_type TO classification
```

The following statement deletes the classification column.

```
ALTER TABLE skill
DELETE classification
```

Example 4

As a final example, the following statement changes the name of the entire table:

```
ALTER TABLE skill
RENAME qualification
```

These examples show how to change the structure of the database. The ALTER TABLE statement can change just about anything pertaining to a table—foreign keys can be added or deleted, columns can be changed from one type to another, and so on.

For a complete description of the ALTER TABLE command, see "ALTER TABLE statement" in the chapter "Watcom-SQL Statements". For information about building constraints into table definitions using ALTER TABLE, see the chapter "Ensuring Data Integrity".

Altering tables in Sybase Central

The property sheets for tables and columns display all the table or column attributes. You can alter a table definition in Sybase Central by displaying the property sheet for the table or column you wish to change, altering the property, and clicking OK to commit the change.

Deleting tables

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

```
DROP TABLE skill
```

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

For a full description of the DROP statement, see "DROP statement" in the chapter "Watcom-SQL Statements".

❖ **To drop a table in Sybase Central:**

1. Connect to the database.
2. Click the Tables folder for that database.
3. Right-click the table you wish to delete, and select Delete from the popup menu.

Creating primary and foreign keys

The CREATE TABLE and ALTER TABLE statements allow many attributes of tables to be set, including column constraints and checks. This section shows how to set table attributes using the primary and foreign keys as an example.

Creating a primary key

The following statement creates the same skill table as before, except that a primary key is added:

```
CREATE TABLE skill (
skill_id INTEGER NOT NULL,
skill_name CHAR( 20 ) NOT NULL,
skill_type CHAR( 20 ) NOT NULL,
primary key( skill_id )
)
```

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

Creating a primary key in Sybase Central

Columns in the primary key are not allowed to contain the NULL value. You must specify NOT NULL on the column in the primary key.

❖ **To create a primary key in Sybase Central:**

1 Connect to the database.

2 Click the Tables folder for that database.

3 Right-click the table you wish to modify, and select Properties from the popup menu, to display its property sheet.

4 Click the Columns tab, and add columns to the primary key, or remove them from the primary key.

☞ For more information, see the Sybase Central online Help.

Creating foreign keys

You can create a table named emp_skill, which holds a description of each employee's skill level for each skill in which they are qualified, as follows:

```
CREATE TABLE emp_skill(
    emp_id INTEGER NOT NULL,
    skill_id INTEGER NOT NULL,
    "skill level" INTEGER NOT NULL,
    PRIMARY KEY( emp_id, skill_id ),
    FOREIGN KEY REFERENCES employee,
    FOREIGN KEY REFERENCES skill
)
```

The emp_skill table definition has a primary key that consists of two columns: the emp_id column and the skill_id column. An employee may have more than one skill, and so appear in several rows, and several employees may possess a given skill, so that the skill_id may appear several times. However, there may be no more than one entry for a given employee's level at a particular skill:

The emp_skill table also has two foreign keys. The **foreign key** entries indicate that the emp_id column must contain a valid employee number from the employee, and that the skill_id must contain a valid entry from the skill table.

The skill level contains a space and is surrounded by quotation marks ("double quotes"). SQL Anywhere allows column names and table names to contain any characters, but the names must be enclosed in quotation marks if any characters other than letters, digits or underscore are used, or if the name does not begin with a letter, or if the name is a keyword.

Single and double quotes in SQL

Remember, in SQL:

- Single quotes (apostrophes) are used to indicate database values (for example, *'SMITH', '100 Apple St.', '1988-1-1'*).
- Double quotes (quotation marks) are used to indicate table or column names (for example, "skill level", "emp_id", "skill_type").

- To include a single quote inside a string, use two single quotes:

```
'''Plankton'' said the cat'
```

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

For more information about using primary and foreign keys, see the chapter "Ensuring Data Integrity".

Creating a foreign key in Sybase Central

❖ To create a foreign key in Sybase Central:

1. Connect to the database.
2. Click the Tables folder for that database.
3. Click the primary key table, and drag it to the foreign key table.
4. When the primary key table is dropped on the foreign key table, the Foreign Key Wizard is displayed, which leads you through the process of creating the foreign key.

For more information, see the Sybase Central online Help.

Table information in the system tables

All the information about tables in a database is held in the system tables. The information is distributed among several tables. For more information, see "SQL Anywhere System Tables".

You can use Sybase Central or ISQL to browse the information in these tables. Type the following command in the ISQL command window to see all the columns in the SYS.SYSTABLE table:

```
SELECT *
FROM SYS.SYSTABLE
```

❖ To display the system tables in Sybase Central:

1. Connect to the database.
2. Right-click the database, and select Show System Objects from the popup menu.
3. When you view the database tables or views with Show System Objects checked, the system tables or views are also shown.

Working with views

Views are computed tables. You can use views to provide to database users exactly the information you want to present, in a format you can control.

Similarities between views and base tables

Views are similar to the permanent tables of the database (a permanent table is also called a **base table**) in many ways:

- You can assign access permissions to views just as to base tables.
- You can perform SELECT queries on views.
- You can perform UPDATE, INSERT, and DELETE operations on some views.
- You can create views based on other views.

Differences between view and permanent tables

There are some differences between views and permanent tables:

- You cannot create indexes on views.
- You cannot perform UPDATE, INSERT, and DELETE operations on all views.
- You cannot assign integrity constraints and keys to views.
- Views are recomputed each time they are invoked. Views refer to the information in base tables, but do not hold copies of that information.

Benefits of tailoring access

Views are used to tailor access to data in the database Tailoring access serves several purposes:

- Improved security: by not allowing access to information that is not relevant.
- Improved usability: by presenting users and application developers with data in a more easily understood form than in the base tables.
- Improved consistency: by centralizing in the database the definition of common queries.

Creating views

A SELECT statement operates on one or more tables and produces a result set that is also a table: just like a base table, a result set from a SELECT query has columns and rows.

A view gives a name to a particular query, and holds the definition in the database system tables.

Example

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

```
SELECT dept_ID, count(*)
FROM employee
GROUP BY dept_ID
```

You can create a view containing the results of this command as follows:

```
CREATE VIEW DepartmentSize AS
SELECT dept_ID, count(*)
FROM employee
GROUP BY dept_ID
```

The information in a view is not stored separately in the database. Each time you refer to the view, SQL Anywhere executes the associated SELECT statement to retrieve the appropriate data. On one hand, this is good because it means that if someone modifies the Employee table, the information in the DepartmentSize view will be automatically up to date. On the other hand, if the SELECT command is complicated it may take a long time for SQL to find the correct information every time you use the view.

❖ **To create a view in Sybase Central:**

1. Connect to the database.
2. Click the Views folder for that database.
3. Double-click Add View.
4. Enter the CREATE VIEW statement in the Sybase Central editor, and click Execute Script to save the view.

For more information, see the Sybase Central online Help.

Using views

Restrictions on SELECT statements

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

To develop a view, you should tune the SELECT query by itself until it provides exactly the results you need in the format you want. Once you have the SELECT query just right, you can add a

```
CREATE VIEW viewname AS
```

phrase in front of the query to create the view.

Updating views

UPDATE, INSERT, and DELETE statements are allowed on some views, but not on others, depending on its associated SELECT statement.

Views containing aggregate functions, such as COUNT(*), cannot be updated. Views containing a GROUP BY clause in the SELECT statement cannot be updated. Also, views containing a UNION operation cannot be updated. In all these cases, there is no way for the database engine to translate the UPDATE into an action on the underlying tables.

Using the WITH CHECK OPTION clause

Even when INSERT and UPDATE statements are allowed against a view, it is possible that the inserted or updated row or rows in the underlying tables may not meet the requirements for the view itself: the view would have no new rows even though the INSERT or UPDATE does modify the underlying tables.

Examples using the WITH CHECK option clause

The following set of examples illustrates the meaning and usefulness of the WITH CHECK OPTION clause. This optional clause is the final clause in the CREATE VIEW statement.

❖ **To create a view displaying the employees in the sales department.**

- Type the following commands:

```
CREATE VIEW sales_employee
AS SELECT   emp_id,
       emp_fname,
       emp_lname,
       dept_id
FROM employee
WHERE dept_id = 200 ;
```

The contents of this view are as follows:

```
SELECT *
   FROM sales_employee
```

emp_id	emp_fname	emp_lname	dept_id
129	Philip	Chin	200
195	Marc	Dill	200
299	Rollin	Overbey	200
467	James	Klobucher	200
641	Thomas	Powell	200

The following UPDATE statement modifies this list, but in such a way that the modified row no longer meets the criterion for the view, and so vanishes from the view.

- **Transfer Philip Chin to the marketing department.** This view update causes the entry to vanish from the view, as it no longer meets the view selection criterion.

```
UPDATE sales_employee
SET dept_id = 400
WHERE emp_id = 129
```

- **List all employees in the sales department.**

```
SELECT *
FROM sales_employee
```

emp_id	emp_fname	emp_lname	dept_id
195	Marc	Dill	200
299	Rollin	Overbey	200
467	James	Klobucher	200
641	Thomas	Powell	200
667	Mary	Garcia	200

When a view is created WITH CHECK OPTION, any UPDATE or INSERT statement on the view is checked to ensure that the new row does match the view condition. If it does not, the operation causes an error and is rejected.

The following modified sales_employee view rejects the update statement, generating an error message "invalid value for column 'dept_id' in table 'employee'".

- **Create a view displaying the employees in the sales department (second attempt).**

```
CREATE VIEW sales_employee
AS SELECT emp_id, emp_fname, emp_lname, dept_id
FROM employee
WHERE dept_id = 200
WITH CHECK OPTION;
```

The check option is inherited

If a view (say V2) is defined on the sales_employee view, any updates or inserts on V2 that cause the WITH CHECK OPTION criterion on sales_employee to fail are rejected, even if V2 is defined without a check option.

Modifying views

To modify a view, you need to first remove it from the database using the DROP statement, and then create a replacement using the CREATE VIEW statement.

For example, to replace the base table column names with more informative names in the DepartmentSize view, you would first drop the view using the DROP VIEW command (see "Deleting views" on page 221) and then create a new view with the same name (see "Creating views" on page 217):

```
CREATE VIEW DepartmentSize (Dept_ID, NumEmployees)
AS
SELECT dept_ID, count(*)
FROM Employee
GROUP BY dept_ID
```

Permissions are lost when you modify views
All permissions governing access to the view are lost when you execute the DROP VIEW command. You need to reassign permissions when you modify views.

Permissions on views

With release 5.0, a change has been made to permissions on views. Before release 5.0, permissions on the underlying tables were required in order for permissions to be granted on views. Permissions can now be granted on views without permissions on the underlying tables.

An INSERT, DELETE, or UPDATE operation is allowed either if permission on the view has been granted or if permission on the underlying tables has been granted. Previously permissions on both the table and the view were required.

UPDATE permissions can be granted only on an entire view. Unlike tables, UPDATE permissions cannot be granted on individual columns within a view.

Deleting views

To delete a view from the database, you use the DROP statement. The following command removes the DepartmentSize view:

```
DROP VIEW DepartmentSize
```

Dropping a view in Sybase Central

To drop a view in Sybase Central, right-click the view you wish to delete and select Delete from the popup menu.

For more information, see the Sybase Central online Help.

Views in the system tables

All the information about views in a database is held in the system table SYS.SYSTABLE. The information is presented in a more readable format in the system view SYS.SYSVIEWS. For more information about these, see "SYSTABLE system table" in the chapter "SQL Anywhere System Tables" and "SYS.SYSVIEWS" in the chapter "SQL Anywhere System Views".

You can use ISQL to browse the information in these tables. Type the following command in the ISQL command window to see all the columns in the SYS.SYSVIEWS view:

```
SELECT * FROM SYS.SYSVIEWS
```

To extract a text file containing the definition of a specific view, use a command such as the following:

```
SELECT viewtext FROM SYS.SYSVIEWS
   WHERE viewname = 'Marks';
OUTPUT TO viewtext.sql FORMAT ASCII
```

Working with indexes

Performance is an important consideration when designing and creating your database. Indexes can dramatically improve the performance of database searches (operations using SELECT, UPDATE and DELETE commands) on specified columns.

When to use indexes

An index is similar to a telephone book which first sorts people by their last name, and then sorts all the people with the same last name by their first name. Telephone books are indexed on the last name and first name. This speeds up searches for phone numbers given a particular last name. Just as a standard telephone book is, however, no use at all for finding the phone number at a particular address, so an index is useful only for searches on a specific column or columns.

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

Use indexes for frequently-searched columns

Indexes share one other feature with a phone book: they can take up a great deal of space for large data sets. For this reason, you should build indexes only for columns that are searched frequently or when disk space is not an issue.

If a column is already a **primary key** or **foreign key**, searches will be fast on this column because SQL Anywhere has facilities to optimize searches on these key columns. Thus, creating an index on a key column is not necessary and generally not recommended. If a column is only part of a key, an index may help.

> **When indexes on primary keys may be useful**
> One case where an index on a key column may assist performance is when a large number of foreign keys reference a primary key. The performance increase is because of SQL Anywhere storage of key indexes.

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

Indexes are created on a specified table. You cannot create an index on a view.

If an index is no longer required, you can remove it from the database using the DROP command.

Example

In order to speed up a search on employee surnames in the sample database, you could create an index called EmpNames with the following statement:

```
CREATE INDEX EmpNames
ON employee (emp_lname, emp_fname)
```

The following statement removes the index from the database:

```
DROP INDEX EmpNames
```

For more information

- For more information about improving database performance, including the use of indexes, see the chapter "Monitoring and Improving Performance".
- For a detailed description of the CREATE INDEX command, including syntax and permission requirements, see "CREATE INDEX statement" in the chapter "Watcom-SQL Statements".
- For a detailed description of the DROP command, including syntax and permission requirements, see "DROP statement" in the chapter "Watcom-SQL Statements".

Creating and dropping indexes in Sybase Central

❖ **To create an index on a table in Sybase Central:**

1. Connect to the database.
2. Double-click the table you wish to modify.
3. Double-click the Indexes folder, and then double-click Add Index.
4. Fill in the dialog box and click OK to complete.

You can drop an index in Sybase Central by right-clicking it, and selecting Delete from the popup menu.

For more information, see the Sybase Central online Help.

Indexes in the system tables

All the information about indexes in a database is held in the system tables SYS.SYSINDEX and SYS.SYSIXCOL. The information is presented in a more readable format in the system view SYS.SYSINDEXES. You can use Sybase Central or ISQL to browse the information in these tables.

CHAPTER 18

Ensuring Data Integrity

About this chapter

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

Several types of integrity constraints can be enforced in SQL Anywhere databases. You can ensure individual entries are correct by imposing constraints and CHECK conditions on tables and columns. Setting column properties by choosing an appropriate data type or setting special default values assists this task.

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

Contents

Topic	Page
Data integrity overview	226
Using column defaults	230
Using table and column constraints	235
Enforcing entity and referential integrity	239
Integrity rules in the system tables	244

Data integrity overview

For data to have integrity means that the data is valid—correct and accurate—and that the relational structure of the database is intact. The relational structure of the database is enforced through referential integrity constraints. These are rules that maintain the consistency of data between tables.

SQL Anywhere supports stored procedures, which allow you detailed control over how data gets entered into the database. SQL Anywhere also allows you to create triggers: custom procedures stored in the database that are invoked automatically when a certain action, such as an update of a particular column, is carried out. Procedures and triggers are discussed in the chapter "Using Procedures, Triggers, and Batches".

How data can become invalid

Here are a few examples of how the data in a database may become invalid if proper checks are not made. Each of these examples can be prevented by SQL Anywhere facilities described in this chapter.

Examples

- Incorrect information
- A sales transaction takes place, but the operator entering the date of the transaction does so incorrectly
- A zero is missed off a salary entry, making an employee's salary ten times too small
- Duplicated data
- A new department has been created, with dept_id 200, and needs to be added to the department table of the organization's database—but two people enter this information into the table
- Foreign key relations invalidated
- In a reorganization, the department identified by dept_id 300 is closed down.

  Each employee record for employees in this department is given a new dept_id entry, and then the department 300 row is deleted from the department table. But one employee was missed, and still has dept_id 300 in their record.

Integrity constraints belong in the database

In order to ensure that the data in a database are valid, you need to formulate checks that define valid and invalid data and design rules to which data must adhere. The rules to which data must conform are often called **business rules**. The collective name for checks and rules is **constraints**.

Build integrity constraints into database

Constraints built in to the database itself are inherently more reliable than those built in to client applications, or spelled out as instructions to database users. Constraints built into the database are part of the definition of the database itself and enforced consistently across all applications.

Setting a constraint once, in the database, imposes it for all subsequent interactions with the database, no matter from what source. In contrast, constraints built into client applications are vulnerable every time the software is altered, and may need to be imposed in several applications, or several places in a single client application.

How database contents get changed

Information in SQL Anywhere database tables is changed by submitting SQL statements from client applications. Only a few SQL statements actually modify the information in a database.

- Information in a row of a table may be **updated**, using the UPDATE statement.
- An existing row of a table may be **deleted**, using the DELETE statement.
- A new row may be **inserted** into a table, using the INSERT statement.

Data integrity tools in SQL Anywhere

To assist in maintaining data integrity, SQL Anywhere provides defaults, data constraints, and constraints that maintain the referential structure of the database.

Defaults

SQL Anywhere allows you to assign default values to columns, to make certain kinds of data entry more reliable. For example:

- A column can have a current date default for recording the date of transactions with any user or client application action.

- A particular kind of default allows column values to be incremented automatically whenever a new row is entered. Items such as purchase orders, for example, can be guaranteed unique sequential numbers in this way without any user action.

These and other column defaults are discussed in "Using column defaults" on page 230.

Constraints

SQL Anywhere also supports several types of constraints on the data in individual columns or tables. For example:

A NOT NULL constraint prevents a column from containing a null entry.

- Columns can have CHECK conditions assigned to them, to ensure that a particular condition is met by every item in the column. You could ensure, for example, that salary column entries are within a specified range, protecting against user error when typing in new values.
- CHECK conditions can be made on the relative values in different columns, to ensure, for example, that in a library database a date_returned entry is later than a date_borrowed entry.
- More sophisticated CHECK conditions can be enforced using a trigger. Triggers are discussed in the chapter "Using Procedures, Triggers, and Batches".

These and other table and column constraints are discussed in "Using table and column constraints" on page 235. Column constraints can be inherited from user-defined data types.

Entity and referential integrity

The information in relational database tables is tied together by the relations between tables. These relations are defined by the primary keys and foreign keys built in to the database design. SQL Anywhere supports two integrity rules that maintain the structure of the database:

- **Entity integrity** Keeps track of the primary keys. It guarantees that every row of a given table can be uniquely identified by a primary key that guarantees no nulls.
- **Referential integrity** Keeps track of the foreign keys that define the relationships between tables. It guarantees that all foreign key values either match a value in the corresponding primary key or contain the NULL value if they are defined to allow NULL.

In addition, SQL Anywhere provides triggers. A **trigger** is a procedure stored in the database that is executed automatically whenever the information in a specified table is altered. Triggers are a powerful mechanism for database administrators and developers to ensure that data is kept reliable. Triggers are discussed in the chapter "Using Procedures, Triggers, and Batches".

For more information about enforcing referential integrity, see "Enforcing entity and referential integrity" on page 239. For more information about designing appropriate primary and foreign key relations, see the chapter "Designing Your Database".

SQL statements for implementing integrity constraints

The following SQL statements are used to implement integrity constraints:

CREATE TABLE statement This statement is used to implement integrity constraints as the database is being created.

ALTER TABLE statement This statement is used to add integrity constraints to an existing database, or to modify constraints for an existing database.

CREATE TRIGGER statement This statement is used to create triggers to enforce more complex business rules.

For full descriptions of the syntax of these statements, see the chapter "Watcom-SQL Language Reference".

Using column defaults

Column defaults automatically assign a specified value 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 default value.

Column defaults are useful for filling columns that contain information such as the date or time a row is inserted, or the user ID of the person entering the information, that is available to the computer automatically.

Using column defaults encourages data integrity, but does not enforce it. Defaults can always be overridden by client applications.

Supported default values

The following default values are supported:

- A string specified in the CREATE TABLE statement or ALTER TABLE statement
- A number specified in the CREATE TABLE statement or ALTER TABLE statement
- An automatically incremented value, one more than the previous highest value in the column
- The current date, or time, or timestamp
- The current user ID of the database user
- A NULL value
- A constant expression, as long as it does not reference database objects.

Creating column defaults

Column defaults can be created at the time a table is created, using the CREATE TABLE statement, or added at a later time using the ALTER TABLE statement.

Example

The following statement adds a condition to an existing column named id in the sales_order table, so that it is automatically incremented (unless a value is specified by a client application):

```
ALTER TABLE sales_order
MODIFY id DEFAULT AUTOINCREMENT
```

☞ Each of the other default values is specified in a similar manner. For a detailed description of the syntax, see "CREATE TABLE statement" in the chapter "Watcom-SQL Statements".

Modifying and deleting column defaults

Column defaults can be changed or removed by using the same form of the ALTER TABLE statement as used to create defaults. The following statement changes the default value of a column named order_date from its current setting to CURRENT DATE:

```
ALTER TABLE sales_order
MODIFY order_date DEFAULT CURRENT DATE
```

Column defaults are removed by modifying them to be NULL. The following statement removes the default from the order_date column:

```
ALTER TABLE sales_order
MODIFY order_date DEFAULT NULL
```

Working with column defaults in Sybase Central

All adding, altering, and deleting of column defaults in Sybase Central is carried out in the Type tab of the column properties sheet.

❖ **To display the property sheet for a column:**

1 Connect to the database.

2 Click the Tables folder for that database, and click the table holding the column you want to change.

3 Double-click the Columns folder to open it, and double-click the column to display its property sheet.

☞ For more information, see the Sybase Central online Help.

Current date and time defaults

For columns with the DATE, TIME, or TIMESTAMP data type, the current date, current time, or current timestamp may be used as a default. The default specified must be compatible with the column's data type.

Useful examples of current date default

The following are just a few examples of when a current date default would be useful:

- To record dates of phone calls and contacts in contact databases
- To record the dates of orders in sales entry databases
- To record the date a book is borrowed in a library database

Current timestamp

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

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

For more information about timestamps, times and dates, see "SQL Anywhere Data Types".

The user ID default

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

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

The autoincrement default

The autoincrement default assigns each new row a value one greater than that of the previous highest value in the column. Autoincrement columns can be used to record purchase order numbers, to identify customer service calls, or other entries where an identifying number is required, but where the value of the number itself has no meaning.

Autoincrement columns are typically primary key. columns or columns constrained to hold unique values (see "Enforcing entity integrity" on page 239). It is highly recommended that the autoincrement default not be used in cases other than these, as doing so can adversely affect the database performance. One case when an autoincrement default does not adversely affect performance is when the column is the first column of an index. This is because the engine uses an index or key definition to find the highest value.

The next value to be used for each column is stored as a long integer (4 bytes). Using values greater than (2**31 - 1), that is, large double or numeric values, may cause wraparound to negative values, and AUTOINCREMENT should not be used in such cases.

The NULL default

For columns that allow NULL values, specifying a NULL default is exactly the same as not specifying a default at all: a NULL value is assigned to the column if no value is explicitly assigned by the client when inserting the row.

NULL defaults are typically used when information for some columns is optional or not always available and is not required for the data in the database be correct.

For more information on the NULL value, see "NULL value" in the chapter "Watcom-SQL Statements".

String and number defaults

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

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

Constant expression defaults

A constant expression can be used as a default value, as long as it does not reference database objects. This allows column defaults to contain entries such as "fifteen days from today", which would be entered as

```
... DEFAULT ( dateadd( day, 15, getdate() ) )
```

Using table and column constraints

The CREATE TABLE statement and ALTER TABLE statement can specify many different attributes for a table. Along with the basic table structure (number, name and data type of columns, name and location of the table), you can specify other features that allow control over data integrity.

> ***Caution***
> *Altering tables can interfere with other users of the database. Although the ALTER TABLE statement can be executed while other connections are active, it is prevented if any other connection is using the table to be altered. For large tables, ALTER TABLE is a time consuming operation, and no other requests referencing the table being altered are allowed while the statement is being processed.*

This section describes how to use constraints to help ensure that the data entered in the table is correct.

Using CHECK conditions on columns

A CHECK condition can be applied to values in a single column, to ensure that they satisfy rules. These rules may be rules that data must satisfy in order to be reasonable, or they may be more rigid rules that reflect organization policies and procedures.

You use a CHECK condition to ensure that the values in a column satisfy some definite criterion.

CHECK conditions on individual column values are useful when only a restricted range of values are valid for that column. Here are some examples:

Example 1

- You can enforce a particular formatting requirement. If a table has a column for phone numbers you may wish to ensure that they are all entered in the same manner. For North American phone numbers, you could use a constraint such as the following:

```
ALTER TABLE customer
MODIFY phone
CHECK ( phone LIKE '(___) ___-____' )
```

Example 2

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

```
ALTER TABLE office
MODIFY city
CHECK ( city IN ( 'city_1', 'city_2', 'city_3' ) )
```

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

Example 3

- You can ensure that a date or number falls in a particular range. For example, you may want to require that the start_date column of an employee table must be between the date the organization was formed and the current date. This could be achieved as follows:

```
ALTER TABLE employee
MODIFY start_date
CHECK ( start_date BETWEEN '1983/06/27'
    AND CURRENT DATE )
```

- SQL Anywhere supports several date formats: the YYYY/MM/DD format used in this example has the virtue of always being recognized regardless of the current option settings.

Column CHECK tests only fail if the condition returns a value of FALSE. If a value of UNKNOWN is returned, the change is allowed.

Column CHECK conditions in previous releases
There is a change in the way that column CHECK conditions are held in this release. In previous releases, column CHECK conditions were merged together with all other CHECK conditions on a table into a single CHECK condition. Consequently, they could not be individually replaced or deleted. In this release, column CHECK conditions are held individually in the system tables, and can be replaced or deleted individually. Column CHECK conditions added before this release are still held in a single table constraint, even if the database is upgraded.

Column CHECK conditions from user-defined data types

You can attach CHECK conditions to user-defined data types, and columns defined on those data types inherit the CHECK conditions. A CHECK condition explicitly specified for the column overrides that from the user-defined data type.

When defining a CHECK condition on a user-defined data type, any variable prefixed with the @ sign is replaced by the name of the column when the CHECK condition is evaluated. For example, the following user-defined data type accepts only positive integers:

```
CREATE DATATYPE posint INT
```

```
CHECK ( @col > 0 )
```

Any variable name prefixed with @ could be used instead of @col. Any column defined using the posint data type accepts only positive integers unless it has a CHECK condition explicitly specified.

An ALTER TABLE statement with the DELETE CHECK clause deletes all CHECK conditions from the table definition, including those inherited from user-defined data types.

For information on user-defined data types, see "User-defined data types" in the chapter "SQL Anywhere Data Types".

Working with column constraints in Sybase Central

All adding, altering, and deleting of column constraints in Sybase Central is carried out in the Constraints tab of the column properties sheet.

❖ **To display the property sheet for a column:**

1. Connect to the database.
2. Click the Tables folder for that database, and click the table holding the column you wish to change.
3. Double-click the Columns folder to open it, and double-click the column to display its property sheet.

For more information, see the Sybase Central online Help.

Using CHECK conditions on tables

A CHECK condition can be applied as a constraint on the table, instead of on a single column. Such CHECK conditions typically ensure that two values in a row being entered or modified have a proper relation to each other. Column CHECK conditions are held individually in the system tables, and can be replaced or deleted individually. This is more flexible behavior, and CHECK conditions on individual columns are recommended where possible.

For example, in a library database, the date_returned column for a particular entry must be later than (or the same as) the date_borrowed entry:

```
ALTER TABLE loan
ADD CHECK(date_returned >= date_borrowed)
```

Modifying and deleting CHECK conditions

There are several ways of altering the existing set of CHECK conditions on a table.

- You can add a new CHECK condition to the table or to an individual column, as described above.
- You can delete a CHECK condition on a column by setting it to NULL. The following statement removes the CHECK condition on the phone column in the customer table: ALTER TABLE customer MODIFY phone CHECK NULL
- You can replace a CHECK condition on a column in the same was as adding a CHECK condition. The following statement adds or replaces a CHECK condition on the phone column of the customer table:

```
ALTER TABLE customer
MODIFY phone
CHECK ( phone LIKE '___-___-____' )
```

- There are two ways of modifying a CHECK condition defined on the table, as opposed to a CHECK condition defined on a column.
- You can add a new CHECK condition using ALTER TABLE with an ADD table-constraint clause.
- You can delete all existing CHECK conditions, including column CHECK conditions, using ALTER TABLE DELETE CHECK, and then add in new CHECK conditions.

All CHECK conditions on a table, including CHECK conditions on all its columns and CHECK conditions inherited from user-defined data types, are removed using the ALTER TABLE statement with the DELETE CHECK clause, as follows:

```
ALTER TABLE table_name
DELETE CHECK
```

Deleting a column from a table does not delete CHECK conditions associated with the column that are held in the table constraint. If the constraints are not removed, any attempt to query data in the table will produce a column not found error message.

Table CHECK conditions fail only if a value of FALSE is returned. If a value of UNKNOWN is returned, the change is allowed.

Enforcing entity and referential integrity

The relational structure of the database enables information within the database to be identified by the database engine, and ensures that relationships described in the database structure between tables are properly upheld by all the rows in each table.

Enforcing entity integrity

When a new row in a table is created, or when a row is updated, SQL Anywhere ensures that the primary key for the table is still valid: that each row in the table is uniquely identified by the primary key.

Example 1

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

Example 2

The sales_order_items table in the sample database uses two columns to define a primary key.

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

If a client application breaches entity integrity

Entity integrity requires that each value of a primary key be unique within the table, and that there are no NULL values. If a client application attempts to insert or update a primary key value, and provides values that are not unique, entity integrity would be breached.

If SQL Anywhere detects an attempt to breach entity integrity, it does not add the new information to the database, and instead reports an error to the client application.

It is up to the application programmer to decide how to present this information to the user and enable the user to take appropriate action. The appropriate action in this case is usually just to provide a unique value for the primary key.

Primary keys enforce entity integrity

Once the primary key for each table is specified, no further action is needed by client application developers or by the database administrator to maintain entity integrity.

The primary key for a table is defined by the table owner when the table is created. If the structure of a table is modified at a later date, the primary key may also be redefined.

Some application development systems and database design tools allow you to create and alter database tables. If you are using such a system, you may not have to enter the CREATE TABLE or ALTER TABLE command explicitly: the application generates the statement itself from the information you provide.

For information on creating primary keys, see "Creating primary and foreign keys" in the chapter "Working with Database Objects". For the detailed syntax of the CREATE TABLE statement, see "CREATE TABLE statement" in the chapter "Watcom-SQL Statements". For information about changing table structure, see the "ALTER TABLE statement" in the chapter "Watcom-SQL Statements".

Enforcing referential integrity

A foreign key relates the information in one table (the foreign table) to information in another (referenced or primary) table. A particular column (or combination of columns) in a foreign table is designated as a foreign key to the primary table.

The entries in the foreign key must correspond to the primary key values of a row in the referenced table for the foreign key relationship to be valid. (Occasionally, some other unique column combination may be referenced, instead of a primary key.)

Example 1

The sample database contains an employee table and a department table. The primary key for the employee table is the employee ID, and the primary key for the department table is the department ID.

One of the items of information about each employee is the department ID of the department to which they belong. In the employee table, the department ID is called a foreign key for the department table; each department ID in the employee table corresponds exactly to a department ID in the department table.

The foreign key relationship is a many-to-one relationship. Several entries in the employee table have the same department ID entry, but the department ID is the primary key for the department table, and so is unique. If a foreign key were able to reference a column in the department table containing duplicate entries, there would be no way of knowing which of the rows in the department table is the appropriate reference.

Example 2

Suppose the database also contained an office table, listing office locations. The employee table might have a foreign key for the office table that indicates where the employee's office is located. The database designer may wish to allow for an office location not being assigned at the time the employee is hired. In this case, the foreign key is **optional** and should allow the NULL value to indicate that it is optional when the office location is unknown or when the employee does not work out of an office. A foreign key that is not optional is called **mandatory**.

Foreign keys enforce referential integrity

Like primary keys, foreign keys are created using the CREATE TABLE statement or ALTER TABLE statement.

Once a foreign key has been created, SQL Anywhere ensures that the columns contain only values that are present as primary key values in the table associated with the foreign key.

For information on creating foreign keys, see "Creating primary and foreign keys" in the chapter "Working with Database Objects".

Losing referential integrity

Referential integrity can be lost in the following ways:

- If a primary key value is updated or deleted, all those foreign keys referencing it would be left in an invalid state.
- If a new row is added to the foreign table, and a value is entered for the foreign key that has no corresponding primary key values, the database would be left in an invalid state.

SQL Anywhere provides protection against both types of integrity loss.

If a client application breaches referential integrity

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

Example

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

While breaches of entity integrity are generally straightforward for SQL Anywhere to handle, simply by refusing to enter the data and returning an error message, these potential breaches of referential integrity are more complicated.

SQL Anywhere ensures that referential integrity is maintained, and provides four options for how this is to be done. These options are called referential integrity actions.

Referential integrity actions

The simplest way of maintaining referential integrity when a referenced primary key is updated or deleted is to disallow the update or delete.

It is often possible to take an action on each foreign key to maintain referential integrity. The CREATE TABLE and ALTER TABLE statements allow database administrators and table owners to specify what action should be taken on foreign keys that reference a modified primary key.

Each of the available referential integrity actions may be specified separately for updates and deletes of the primary key:

- **RESTRICT** Generate an error if an attempt is made to modify a referenced primary key value, and do not carry out the modification. This is the default referential integrity action.
- **SET NULL** Set all foreign keys that reference the modified primary key to NULL.
- **SET DEFAULT** Set all foreign keys that reference the modified primary key to the default clause for that column (as specified in the table definition).

- **CASCADE** When used with ON UPDATE, update all foreign keys that reference the updated primary key to the new value of the primary key. When used with ON DELETE, delete all rows containing foreign keys that reference the deleted primary key.

Referential integrity actions are implemented using system triggers. The trigger is defined on the primary table, and is executed using the permissions of the owner of the primary table.

Referential integrity checking

For foreign keys defined to RESTRICT operations that would violate referential integrity, checks can be carried out at the time a statement is executed (the default) or only when a transaction is committed, by specifying the CHECK ON COMMIT clause.

Using a database option to control check time

SQL Anywhere operates in two different modes when a foreign key is defined to RESTRICT operations that would violate referential integrity, depending on the setting of the wait_for_commit database option. This option is overridden by the CHECK ON COMMIT clause.

Wait_for_commit = off

With the default setting (wait_for_commit = off — see "SET OPTION statement" in the chapter "Watcom-SQL Statements") a database operation that would leave the database inconsistent is not allowed to execute. For example, a DELETE operation of a department that has employees in it is not allowed. The statement:

```
DELETE FROM department
WHERE dept_id = 200
```

gives the error primary key for row in table 'department' is referenced in another table.

Wait_for_commit = on

If wait_for_commit is set to *on*, referential integrity is not checked until a commit is executed. If the database is in an inconsistent state, the commit is not allowed and an error is reported. In this mode, a department with employees could be deleted. However, the change could not be committed to the database until one of the following actions is taken:

- The employees belonging to that department are also deleted or reassigned.
- This search condition can also be used on a SELECT statement to select the rows that violate referential integrity.
- The dept_id row is inserted back into the department table.
- The transaction is rolled back to undo the DELETE operation.

Integrity rules in the system tables

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

System table	Description
SYS.SYSTABLE	CHECK constraints are held in the view_def column of SYS.SYSTABLE. For views, the view_def holds the CREATE VIEW command that created the view. You can check whether a particular table is a base table or a view by looking at the table_type column, which is BASE or VIEW respectively
SYS.SYSTRIGGER	Referential integrity actions are held in SYS.SYSTRIGGER. The referential_action column holds a single character indicating whether the action is cascade (C), delete (D), set null (N), or restrict (R). The event column holds a single character specifying the event that causes the action to occur: a delete (D), insert (I), update (U), or update of column-list (C). The trigger_time column shows whether the action occurs after (A) or before (B) the triggering event
SYS.SYSFOREIGN KEYS	This view presents the foreign key information from the two tables SYS.SYSFOREIGNKEY and SYS.SYSFKCOL in a more readable format
SYS.SYSCOLUMNS	This view presents the information from the SYS.SYSCOLUMN table in a more readable format. It includes default settings and primary key information for columns

For a description of the contents of each system table, see the chapter "SQL Anywhere System Tables". You can use Sybase Central or ISQL to browse these tables and views.

CHAPTER 19

Using Transactions and Locks

About this chapter

This chapter describes transactions and how to use them in applications. It also describes the locking mechanisms available to assist concurrent usage of the database.

Contents

Topic	Page
An overview of transactions	246
How locking works	250
Isolation levels and consistency	251
How SQL Anywhere handles locking conflicts	254
Choosing an isolation level	256
Savepoints within transactions	258
Particular concurrency issues	259
Transactions and portable computers	262

An overview of transactions

About transaction processing

SQL Anywhere supports transaction processing to ensure that logically related commands get executed as a unit, and to enable multiple applications to have access to a database.

When several users are working with the same information in a database at the same time (concurrently), their actions may interfere with each other to produce inconsistent and incorrect information. Transaction processing and row-level locking allow concurrent use while maintaining database consistency.

Who needs to know about transactions

Transaction processing is not a concern simply for developers of applications for multi-user database engines; developers working with single-user databases also have to be concerned with concurrency. Single-user SQL Anywhere databases may still have multiple applications connected to them, or may have multiple connections from a single application. These applications and connections can interfere with each other in exactly the same way as multiple users in a network setting.

Transaction processing is fundamental to proper data recovery in case of system failures.

For information about database backups and data recovery, see the chapter "Backup and Data Recovery".

Transactions are logical units of work

SQL statements are grouped into transactions. A **transaction** is a logical unit of work, meaning that the set of commands making up a transaction must be processed in its entirety, or not at all. From the user's point of view, a transaction is indivisible. Transactions are **atomic**.

Starting transactions

Transactions start with one of the following events:

- The first statement following a connection to a database
- The first statement following the end of a transaction

Completing transactions

Transactions complete with one of the following events:

- A COMMIT statement makes the changes to the database permanent
- A ROLLBACK statement undoes all the changes made by the transaction
- A statement with a side effect of an automatic commit is issued (Database definition commands, such as ALTER, CREATE, COMMENT, or DROP all have the side effect of an automatic commit)
- A disconnection from a database (implicit rollback)

SQL Anywhere's **transaction processing** ensures that each transaction is processed in its entirety or not at all. Transaction processing is fundamental to ensuring that a database contains correct information. It addresses two distinct, yet related, problems: data recovery and database consistency in the face of concurrent usage.

Example

A transfer of funds from one account to another is an archetypal transaction. This transaction consists of two operations:

1 Debit the account the money is coming from.

2 Credit the account the money is going to.

If the transaction is completed properly both operations are recorded in the database. However, if something happens after the *from* account is debited to prevent the transaction being completed, then it is unacceptable to leave the debit in the database without recording the credit. Either both the debit and the credit must be processed, or neither. In case of failure, the debit needs to be undone, or *rolled back.*

Transactions and data recovery

Transaction processing ensures that if, for any reason, a transaction is not successfully completed, then the entire transaction is undone, or rolled back. The database is left entirely unaffected by failed transactions.

SQL Anywhere's transaction processing ensures that the contents of a transaction are processed securely, even in the event of system failures in the middle of a transaction. The mechanisms for data recovery are described in the chapter "Backup and Data Recovery". The remainder of this chapter is devoted to concurrency and consistency of transactions.

Transactions and concurrency

When several users are using a database at the same time they are said to be **concurrent** users. Even single-user databases need to be aware of concurrency problems, as several applications could be run at the same time on one machine, or a single application could have several active connections to the database at one time.

If several connections access the same information in a database at the same time, they could interfere with each other and produce inconsistencies in the database. Transaction processing helps to ensure databases remain consistent while allowing concurrent transactions.

Three types of inconsistency

There are three types of inconsistency that can occur during the execution of concurrent transactions:

- **Dirty read** Transaction A modifies a row. Transaction B then reads that row before transaction A performs a COMMIT. If transaction A then performs a ROLLBACK, transaction B will have read a row that was never committed.
- **Nonrepeatable read** Transaction A reads a row. Transaction B then modifies or deletes the row and performs a COMMIT. If transaction A then attempts to read the same row again, the row will have been changed or deleted.
- **Phantom row** Transaction A reads a set of rows that satisfy some condition. Transaction B then executes an INSERT, or an UPDATE (that generates one or more rows that satisfy the condition used by transaction A) and then performs a COMMIT. Transaction A then repeats the initial read and obtains a different set of rows.

Using locks to ensure consistency

SQL Anywhere uses a locking scheme to ensure that concurrent transactions do not interfere with each other to produce inconsistencies.

Inconsistency in the information an application sees is tolerable in some cases. Therefore, we do not need to prohibit all forms of inconsistent behavior in all cases. For this reason, the client application developer is given some control over the level of consistency required in the information the application sees.

SQL Anywhere locking scheme

The SQL Anywhere locking scheme restricts access to the information that a particular transaction is working with to ensure that other transactions do not see information that may not be committed to the database, and do not alter information on which the transaction is relying. The locking scheme is discussed in detail later in this chapter.

The way SQL statements are grouped into transactions can have significant effects on data integrity and on system performance. If a transaction is too short, and does not contain an entire logical unit of work, then inconsistencies can be introduced into the database. If a transaction is too long, and contains several unrelated actions, then there is greater chance of a ROLLBACK unnecessarily undoing work that could have been committed quite safely into the database. Also, if transactions are too long, and lock large amounts of data, they can prevent other transactions from being processed and so reduce concurrency.

There are many factors determining the appropriate length of a transaction, depending on the type of application and the environment. Some guidelines are given towards the end of this chapter.

How locking works

When a transaction is reading or writing a row in a database table, the database engine automatically locks the individual row (row level locking) to prevent other transactions from interfering with the data, or from obtaining unreliable data. The transaction that has access to the row is said to hold the lock. Depending on the type of lock, other transactions may have limited access to the locked row, or none at all.

All locks for a transaction are held until the transaction is complete (COMMIT or ROLLBACK), with a single exception noted below.

SQL Anywhere allows users to determine the extent to which transactions can operate concurrently by setting **isolation levels**. Isolation levels are discussed in the next section.

Types of locks

There are three distinct types of locks:

- read lock
- write lock
- phantom lock

Uses for locks

They have the following uses:

- Whenever a transaction inserts, updates, or deletes a row, a **write lock** is set. No other transaction can obtain a lock on the same row when a write lock is set. A write lock is an **exclusive lock**.
- A **read lock** can be set when a transaction reads a row. Several transactions can acquire read locks on the same row (a read lock is a **nonexclusive lock**). However, once a row has been read locked no other transaction may obtain a write lock on it.
- A **phantom lock** is a read lock that prevents phantom rows. Phantom locks for lookups using indexes require a read lock on each row that is read, and one extra read lock to prevent insertions into the index at the end of the result set. Phantom rows for lookups that do not use indexes require a read lock on all rows in a table to prevent insertions from altering the result set, and so can have a bad effect on concurrency.

Isolation levels and consistency

The degree to which the operations in one transaction are visible to the operations in a concurrent transaction is defined by isolation level. SQL Anywhere has four different isolation levels that prevent some or all inconsistent behavior. The isolation level is a database option that can be different for each connection. Database options are changed by using the SET command; the default setting is isolation level 0. For a description of the SET statement syntax, see the chapter "Watcom-SQL Language Reference".

All isolation levels guarantee that each transaction will execute completely or not at all, and that no updates will be lost. SQL Anywhere therefore ensures recoverability at all times, regardless of the isolation level.

Isolation levels and dirty reads, nonrepeatable reads, and phantom rows

The isolation levels are different with respect to dirty reads, nonrepeatable reads, and phantom rows. An **x** means that the behavior is prevented, and a ✓ means that the behavior may occur.

Isolation level	0	1	2	3
Dirty reads	✓	x	x	x
Non-repeatable reads	✓	✓	x	x
Phantom rows	✓	✓	✓	x

If your application is using cursors to perform retrievals and updates, there is an additional level of isolation called **cursor stability**. Cursor stability guarantees that any row that is the current position of a cursor will not be modified by another transaction until the cursor leaves the row. No row fetched through a cursor yields uncommitted data. SQL Anywhere automatically provides cursor stability at isolation levels 1, 2, and 3.

Locks and isolation levels

Write locks are employed at all isolation levels. This ensures that once data is modified by a transaction, no other transaction can modify it until the transaction is either committed or rolled back. As long as the transaction completes, there is no danger of its database update being interfered with by any other user.

Isolation level 0 (Read uncommitted)

The default setting for SQL Anywhere is isolation level 0. Read locks are not employed at this isolation level.

When all transactions are running at isolation level 0, the only time a locking conflict occurs is when one transaction attempts to update or delete a row that has been inserted or updated by a different transaction and not yet committed.

Isolation level 1 (Read committed)

The only distinction between levels 0 and 1 occurs when one transaction has a write lock on a row (because it is modifying the row) and a second transaction attempts to read the row.

At isolation level 0, the second transaction is allowed to read the row, at the risk that the read may be dirty. At isolation level 1, the second transaction checks to see if there is a write lock on the row before reading it, and is not allowed to read it if a write lock is in place.

At isolation level 1, cursor stability is achieved by putting a read lock on the current row of a cursor. This read lock is removed when the cursor is moved. This is the only type of lock that does not persist until the end of a transaction.

Isolation level 2 (Repeatable read)

Read locks that persist until the end of a transaction are introduced at isolation level 2. Level 2 guarantees that there will be no non-repeatable reads. If an application once reads a row, that row will be available to be read again, and will give the same result.

No guarantee is given that the row can be updated, however, as another transaction may also have a read lock on the row.

Isolation level 3 (Serializable)

Isolation level 3 is the most secure level, and is also the one at which concurrency is most affected.

At isolation level 3 a phantom lock is employed to prevent phantom rows. If your application looks up data in tables without using an index, concurrency can suffer considerably under isolation level 3.

Changing the isolation level

SQL Anywhere allows you to change isolation levels at any time, including within a transaction, using the SET OPTION statement.

For more information, see "SET OPTION statement" in the chapter "Watcom-SQL Statements".

When the ISOLATION_LEVEL option is changed in the middle of a transaction, the new setting affects only the following:

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

This may be useful, for example, when just one table or group of tables involved in a transaction requires serialized access, and so a high isolation level.

How SQL Anywhere handles locking conflicts

When a transaction attempts to acquire a lock on a row, but is forbidden by a lock held by another transaction, a locking conflict arises and the transaction attempting to acquire the lock is blocked. The section "Locking conflicts and transaction blocking", next, describes how SQL Anywhere handles transaction blocking.

☞ "Transaction blocking and deadlock" below, describes how SQL Anywhere handles a deadlock when transactions cannot become unblocked.

Locking conflicts and transaction blocking

When a locking conflict occurs, one transaction must wait for another transaction to complete. A transaction becomes blocked on another transaction. If two transactions simultaneously have a read lock on a row, the behavior when one of them attempts to modify that row (acquire a write lock) depends on the database setting BLOCKING.

- If BLOCKING is ON (the default setting), then the transaction that attempts to write is blocked until the other transaction releases its read lock. At that time, the write goes through.
- If BLOCKING has been set to OFF, then the transaction that attempts to write receives an error.

Blocking is more likely to occur with higher isolation levels because more locking and more checking is done. Higher isolation levels provide less concurrency.

☞ For information about the BLOCKING option, see "SET OPTION statement" in the chapter "Watcom-SQL Statements".

Transaction blocking and deadlock

Transaction blocking can lead to **deadlock**, where a set of transactions get into a state where none of them can proceed.

Reasons for deadlocks

A deadlock can arise for two reasons:

- **A cyclical blocking conflict** Transaction A is blocked on transaction B, and transaction B is blocked on transaction A. Clearly, more time will not solve the problem, and one of the transactions must be canceled, allowing the other to proceed. The same situation can arise with more than two transactions blocked in a cycle.
- **All active database threads are blocked** When a transaction becomes blocked, its database thread is not relinquished. If the database is configured with three threads (see the THREAD_COUNT option in "SET OPTION statement" in the chapter "Watcom-SQL Statements") and transactions A, B, and C are blocked on transaction D which is not currently executing a request, then a deadlock situation has arisen since there are no available threads.

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

Choosing an isolation level

The choice of isolation level depends on the kind of task an application is carrying out. This section gives some guidelines for choosing isolation levels.

Typical level 0 transactions

Transactions that involve browsing or performing data entry may last several minutes, and read a large number of rows. If isolation level 2 or 3 is used, concurrency can suffer. An isolation level of 0 or 1 is typically used for this kind of transaction.

For example, a decision support application that reads large amounts of information from the database to produce statistical summaries may not be significantly affected if it reads a few rows that are later modified. If high isolation was required for such an application, it could hold read locks on large amounts of data, not allowing other applications write access to it.

Transactions for which no updates are lost

Some applications require that no updates be lost. The following example typifies the lost update problem.

Example

Consider a sequence of instructions which could occur when two people put money into the same account at about the same time.

The initial account balance is $1000, and two people (Alex and Ben, say) are about to deposit money into it. Alex will deposit $2000, while Ben will deposit $100. We'll call Alex's transaction is transaction A and Ben's transaction is transaction B.

1 A reads the account balance and finds it to be $1000.

2 B reads the account balance and finds it to be $1000.

3 A adds $2000 to the present balance of $1000 to calculate the new account balance. It then updates the account balance to reflect Alex's deposit. It writes a new balance of $3000 into the database.

4 B adds $100 to the present account balance, which it has read as $1000, to calculate the new account balance. It then updates the account balance to reflect Ben's deposit, and writes a balance of $1100 into the database.

5 The final balance recorded after the two deposits is $1100.

While both transactions are perfectly correct in themselves, the interaction between the two creates an invalid result in the database, and Alex's update was lost.

Isolation levels that prevent lost updates

SQL Anywhere's locking mechanism prevents lost updates at isolation level 2 and 3. If your application is using cursors, then **cursor stability** (achieved at isolation level 1) guarantees no lost updates.

Serializable transactions

Some applications require **serializable** transactions due to the nature of the application. Isolation level 3 enforces serializability.

When transactions are serializable, they behave as if they were run one after another even if they were actually run concurrently. For example, banking software must prevent two machines from checking a balance and withdrawing the full amount from the account at the same time.

Transactions of this type should read few or no rows and last at most a few seconds, so that concurrency is not likely to be a problem. Applications that involve a high volume of small transactions can use isolation level 3 without sacrificing concurrency.

Savepoints within transactions

Within a transaction, SQL Anywhere supports **savepoints**. Before Watcom SQL 4.0, a savepoint was referred to as a **subtransaction**.

A SAVEPOINT statement defines a point in a transaction where all changes after the point can be undone by a ROLLBACK TO SAVEPOINT statement. Once a RELEASE SAVEPOINT statement has been executed, the savepoint can no longer be used.

No locks are released by the RELEASE SAVEPOINT or ROLLBACK TO SAVEPOINT commands: locks are released only at the end of a transaction.

Naming and nesting savepoints

Savepoints can be named and they can be nested. By using named, nested savepoints, you can have many active savepoints within a transaction. Changes between a SAVEPOINT and a RELEASE SAVEPOINT can still be canceled by rolling back to a previous savepoint or rolling back the transaction itself. Changes within a transaction are not a permanent part of the database until the transaction is committed. All savepoints are released when a transaction ends.

Savepoints make use of the rollback log. They cannot be used in bulk operations mode. There is very little additional overhead in using savepoints.

Particular concurrency issues

This section discusses the following particular concurrency issues:

- Primary key generation
- Data definition statements and concurrency
- Coordinating transactions with multiple database engines

Primary key generation

Many applications generate primary key values automatically.

Example

For example, invoice numbers could be obtained by adding 1 to the previous invoice number. This will not work when there is more than one person adding invoices to the database. Two people may decide to use the same invoice number.

There is more than one solution to the problem:

- Use a different range of invoice numbers for each person that adds new invoices.
- This could be done by having a table with two columns (user name and invoice number). The table would have one row for each user that adds invoices. Each time a user adds an invoice, the number in the table would be incremented and used for the new invoice. In order to handle all tables in the database, the table should have three columns (table name, user name, and last key value).
- Have a table with two columns (table name and last key value).
- There would be one row in this table for the last invoice number used. Each time a user adds an invoice, establish a new connection, increment the number in the table, and commit. The incremented number can be used for the new invoice. Other users will be able to grab invoice numbers because you updated the row with a separate transaction that only lasted an instant.
- Probably the best solution is to use a column with a default value of AUTOINCREMENT.

For example: CREATE TABLE orders (order_id INTEGER NOT NULL DEFAULT AUTOINCREMENT, order_date DATE, primary key(order_id)). On INSERTs into the table, if a value is not specified for the autoincrement column, a unique value is generated. If a value is specified, it will be used. If the value is larger than the current maximum value for the column, that value will be used as a starting point for subsequent INSERTs. The value of the most recently inserted row in an autoincrement column is available as the global variable @@identity.

Data definition statements and concurrency

The CREATE INDEX statement, ALTER TABLE statement, and DROP statement are prevented whenever the statement affects a table that is currently being used by another connection. These statements can be time consuming and the database server will not process requests referencing the same table while the command is being processed.

The CREATE TABLE statement does not cause any concurrency conflicts.

The GRANT statement, REVOKE statement, and SET OPTION statement also do not cause concurrency conflicts. These commands affect any new SQL statements sent to the database engine, but do not affect existing outstanding statements.

GRANT and REVOKE for a user are not allowed if that user is connected to the database.

Coordinating transactions with multiple database engines

Two-phase commit is a mechanism of coordinating transactions between multiple servers. It is a primary component of most distributed database systems. Most applications do not need to use two-phase commit.

The first phase of a two-phase commit asks the database engine to prepare to commit and report any errors that would occur on a commit. This phase is accomplished with the PREPARE TO COMMIT statement. The second phase actually performs the commit operation using the COMMIT statement.

If you want to coordinate transactions with multiple servers, you can issue the PREPARE TO COMMIT statement to each server. If one of them fails, you can deal with the error or rollback all transactions. If all of the first phase commits are successful, you can commit each transaction knowing that there won't be any errors (except environment errors such as a network connection going down or one of the servers going off line).

Transactions and portable computers

Some of the computers on your network might be portable computers that people take away from the office or which are occasionally connected to the network. There may be several database applications that they would like to use while not connected to the network.

Clearly, they cannot update the database server while they are not connected to the network. They can, however, take a copy of the database file and make updates to the copy using the SQL Anywhere server Standalone database engine running on the portable computer. Later, on returning to the office, the transaction log can be translated into a command file and applied to the database server.

This type of process is fully automated by the new SQL Remote replication system, described in chapters "Introduction to SQL Remote Replication" and "SQL Remote Administration".

Required software
You must purchase a copy of SQL Anywhere Standalone and SQL Remote for each machine on which you wish to deploy a client application running on the SQL Anywhere Standalone database engine.

Applying updates from a portable computer

There are potential problems with applying translated transaction logs. When machines are connected to the network, locking prevents conflicting updates to records in the database. When a user makes changes to a copied database, there is no such protection. Consider what happens when two users update the same record, or one user deletes a record that another user updates. You can design applications to avoid this sort of problem by having each user update an isolated subset of the data, but you need to be aware of the possibility.

One solution is to use the *-v* command line switch when starting the database engine on the portable computer. This causes the engine to record the previous values of every column whenever a row of the database is updated. When the transaction log is translated (using DBTRAN) every UPDATE will have a WHERE clause specifying the value of every column in the row. The row will not be updated if any value has been changed by another user. ISQL will display a message if an UPDATE does not affect any rows, indicating that the particular database row has already been updated or deleted by someone else. When this happens, your update procedure will need manual intervention to resolve the potential conflict.

SQL Remote has full conflict detection capabilities and the ability to resolve conflicts through conflict triggers.

Distributing applications that do not require server updates

If you want to deploy applications that work on information taken from a database file, but which do not require applying updates to a multiuser database, you can install the SQL Anywhere Desktop Runtime System on each computer.

The runtime database engine supports the full range of data manipulation language commands, such as INSERT, DELETE and UPDATE, but does not employ a transaction log. Consequently, updates made on the local database cannot be applied to a multiuser database.

Required software
To use the single-user runtime database engine on an unlimited number of machines, you need to have purchased the SQL Anywhere Desktop Runtime System.

Working with large databases on portable computers

With large database files, it may help to use DBSHRINK to compress the database before making a copy. In this case you will need to use a write file with the copied database (see DBWRITE).

You could also use a subset of the large database by creating an extraction procedure that builds a database that contains only the data needed by one person. As long as the table names and column names are identical, translated transaction logs from the smaller database can be applied to the main database.

CHAPTER 20

Using Procedures, Triggers, and Batches

About this chapter

Procedures and triggers store procedural SQL statements in the database for use by all applications. They enhance the security, efficiency, and standardization of databases. User-defined functions are one kind of procedure that return a value to the calling environment for use in queries and other SQL statements. Batches are sets of SQL statements submitted to the database server as a group. Many features available in procedures and triggers, such as control statements, are also available in batches.

Contents

Topic	Page
Procedure and trigger overview	266
Benefits of procedures and triggers	267
Introduction to procedures	268
Introduction to user-defined functions	273
Introduction to triggers	276
Introduction to batches	281
Control statements	283
The structure of procedures and triggers	286
Returning results from procedures	290
Using cursors in procedures and triggers	295
Errors and warnings in procedures and triggers	300
Using the EXECUTE IMMEDIATE statement in procedures	307
Transactions and savepoints in procedures and triggers	308
Some hints for writing procedures	309
Statements allowed in batches	312
Calling external libraries from stored procedures	314

Procedure and trigger overview

Procedures and triggers store procedural SQL statements in a database for use by all applications.

Procedures and triggers can include control statements that allow repetition (LOOP statement) and conditional execution (IF statement and CASE statement) of SQL statements.

Procedures are invoked with a CALL statement, and use parameters to accept values and return values to the calling environment. Procedures can also return result sets to the caller. Procedures can call other procedures and fire triggers.

Triggers are associated with specific database tables. They are invoked automatically (**fired**) whenever rows of the associated table are inserted, updated or deleted. Triggers do not have parameters and cannot be invoked by a CALL statement. Triggers can call procedures and fire other triggers.

User-defined functions are one kind of stored procedure that returns a single value to the calling environment. User-defined functions do not modify parameters passed to them. They broaden the scope of functions available to queries and other SQL statements.

Benefits of procedures and triggers

Procedures and triggers are defined in the database, separate from any one database application. This separation provides a number of advantages.

Standardization

Procedures and triggers allow standardization of any actions that are performed by more than one application program. The action is coded once and stored in the database. The applications need only CALL the procedure or fire the trigger to achieve the desired result. If the implementation of the action evolves over time, any changes are made in only one place, and all applications that use the action automatically acquire the new functionality.

Efficiency

When used in a database implemented on a network server, procedures and triggers are executed on the database server machine. They can access the data in the database without requiring network communication. This means that they execute faster and with less impact on network performance than if they had been implemented in an application on one of the client machines.

When a procedure or trigger is created, it is checked for correct syntax and then stored in the system tables. The first time it is required by any application, it is retrieved from the system tables and compiled into the virtual memory of the database engine, and executed from there. Subsequent executions of the same procedure or trigger will result in immediate execution, since the compiled copy is retained. A procedure or trigger can be used concurrently by several applications and recursively by one application. Only one copy is compiled and kept in virtual memory.

Security

Procedures, including user-defined functions, execute with the permissions of the procedure owner but can be called by any user that has been granted permission to do so.

Triggers execute under the table permissions of the owner of the associated table but are fired by any user with permission to insert, update or delete rows in the table. This means that a procedure or trigger can (and usually does) have different permissions than the user ID that invoked it. Procedures and triggers provide security by allowing users limited access to data in tables that they cannot directly examine or modify.

Introduction to procedures

In order to use procedures you need to understand how to do the following:

- Create procedures
- Drop, or remove, procedures
- Call procedures from a database application
- Control who has permission to use procedures

This section discusses each of these aspects of using procedures, and also describes some of the different uses of procedures.

Creating procedures

Procedures are created using the CREATE PROCEDURE statement. You must have RESOURCE authority in order to create a procedure.

The following simple example creates a procedure that carries out an insert into the department table of the sample database, creating a new department.

You can create the example procedure new_dept by connecting to the sample database from the ISQL utility as user ID DBA, using password SQL, and typing the statement in the command window.

If you are using a tool other than ISQL or Sybase Central, you may need to change the command delimiter away from the semicolon before entering the CREATE PROCEDURE statement.

```
CREATE PROCEDURE new_dept ( IN id INT,
    IN name CHAR(35),
    IN head_id INT )
BEGIN
INSERT
INTO "dba".department ( dept_id,
    dept_name,
    dept_head_id )
VALUES ( id, name, head_id );
END
```

For a complete description of the CREATE PROCEDURE syntax, see "CREATE PROCEDURE statement" in the chapter "Watcom-SQL Statements".

The body of a procedure is a **compound statement** (see "Using compound statements" on page 283). The compound statement starts with a BEGIN statement and concludes with an END statement. In the case of new_dept, the compound statement is a single INSERT bracketed by BEGIN and END statements.

Parameters to procedures are marked as one of IN, OUT, or INOUT. All parameters to the new_dept procedure are IN parameters, as they are not changed by the procedure.

Calling procedures

A procedure is invoked with a CALL statement (see "CALL statement" in the chapter "Watcom-SQL Statements".) Procedures can be called by an application program or they can be called by other procedures and triggers.

The following statement calls the new_dept procedure to insert an Eastern Sales department:

```
CALL new_dept( 210, 'Eastern Sales', 902 );
```

After this call, you may wish to check the department table to see that the new department has been added.

❖ **To list all departments:**

- Type the following:

```
SELECT *
FROM department
```

dept_id	dept_name	dept_head_id
100	R & D	501
200	Sales	902
300	Finance	1293
400	Marketing	1576
500	Shipping	703
210	Eastern Sales	902

The new_dept procedure can be called by any user who has been granted EXECUTE permission for the procedure, even if they have no permissions on the department table.

Dropping procedures

Once a procedure is created, it remains in the database until it is explicitly removed. Only the owner of the procedure or a user with DBA authority can drop the procedure from the database.

The following statement removes the procedure new_dept from the database:

```
DROP PROCEDURE new_dept
```

Permissions to execute procedures

A procedure is owned by the user who created it and that user can execute it without permission. Permission to execute it can be granted to other users using the GRANT EXECUTE command.

For example, the owner of the procedure new_dept could allow another_user to execute new_dept with the statement:

```
GRANT EXECUTE ON new_dept TO another_user
```

The following statement revokes permission to execute the procedure:

```
REVOKE EXECUTE ON new_dept FROM another_user
```

For more information on managing user permissions on procedures, see "Granting permissions on procedures" in the chapter "Managing User IDs and Permissions".

Returning procedure results in parameters

Procedures can return results to the calling environment in one of the following ways:

- Individual values are returned as OUT or INOUT parameters to the procedure
- Result sets can be returned
- A single result can be returned using a RETURN statement

This section describes how to return results from procedures as parameters.

The following procedure on the sample database returns the average salary of employees as an OUT parameter.

```
CREATE PROCEDURE AverageSalary( OUT avgsal
   DECIMAL(20,3) )
```

```
BEGIN
   SELECT AVG( salary ) INTO avgsal FROM employee;
END
```

To run this procedure and display its output from the ISQL utility, carry out the following steps:

1 Connect to the sample database from the ISQL utility as user ID DBA using password SQL.

2 Create the procedure.

3 Create a variable to hold the procedure output. In this case, the output variable is numeric, with three decimal places, so create a variable as follows:

```
CREATE VARIABLE Average NUMERIC(20,3)
```

4 Call the procedure, using the created variable to hold the result: CALL AverageSalary(Average) The ISQL statistics window displays the message "Procedure completed" if the procedure was created and run properly.

5 Look at the value of the output variable Average. The ISQL data window displays the value 49988.623 for this variable; the average employee salary.

Returning procedure results in result sets

In addition to returning results to the calling environment in individual parameters, procedures can return information in result sets. A result set is typically the result of a query. The following procedure returns a result set containing the salary for each employee in a given department:

```
CREATE PROCEDURE SalaryList ( IN department_id INT)
RESULT ( "Employee ID" INT, "Salary" NUMERIC(20,3) )
BEGIN
   SELECT emp_id, salary
   FROM employee
   WHERE employee.dept_id = department_id;
END
```

If called from ISQL, the names in the RESULT clause are matched to the results of the query and used as column headings in the displayed results.

To test this procedure from within ISQL you can CALL it, specifying one of the departments of the company. The results are displayed in the ISQL data window. For example:

❖ **To list the salaries of employees in the R & D department (department ID 100):**

♦ Type the following:

```
CALL SalaryList (100)
```

Employee ID	Salary
102	45700.000
105	62000.000
160	57490.000
243	72995.000
247	48023.690

To execute a CALL of a procedure that returns a result set, ISQL opens a cursor.

The cursor is left open after the CALL in case a second result set is returned. The ISQL statistics window displays the plan of the SELECT query in the procedure and then displays the line:

Procedure is executing. Use RESUME to continue.

You need to execute the *RESUME* statement from the ISQL command window before you can, for example, DROP the procedure. Alternatively, the ISQL CLEAR statement clears the window.

For more information about using cursors in procedures, see "Using cursors in procedures and triggers" on page 295.

Introduction to user-defined functions

User-defined functions are a class of procedures that return a single value to the calling environment. This section introduces creating, using, and dropping user-defined functions

Creating user-defined functions

User-defined functions are created using the CREATE FUNCTION statement. You must have RESOURCE authority in order to create a user-defined function.

The following simple example creates a function that concatenates two strings, together with a space, to form a full name from a first name and a last name.

You can create the example function fullname by connecting to the sample database from the ISQL utility as user ID DBA, using password SQL, and typing the statement in the command window.

If you are using a tool other than ISQL or Sybase Central, you may need to change the command delimiter away from the semicolon before entering the CREATE FUNCTION statement.

```
CREATE FUNCTION fullname (firstname CHAR(30),
   lastname CHAR(30))
RETURNS CHAR(61)
BEGIN
   DECLARE name CHAR(61);
   SET name = firstname || ' ' || lastname;
   RETURN ( name );
END
```

For a complete description of the CREATE FUNCTION syntax, see "CREATE FUNCTION statement" in the chapter "Watcom-SQL Statements".

The CREATE FUNCTION syntax differs slightly from that of the CREATE PROCEDURE statement. The following are distinctive differences:

- No IN, OUT, or INOUT keywords are required, as all parameters are IN parameters.
- The RETURNS clause is required to specify the data type being returned.
- The RETURN statement is required to specify the value being returned.

Calling user-defined functions

A user-defined function can be used, subject to permissions, in any place that a built-in non-aggregate function is used.

The following statement in ISQL displays a full name from two columns containing a first and last name:

```
SELECT fullname (emp_fname, emp_lname)
FROM employee;
```

fullname (emp_fname, emp_lname)
Phillip Chin

The following statement in ISQL displays a full name from a supplied first and last name:

```
SELECT fullname ('Jane', 'Smith');
```

fullname ('Jane','Smith')
Jane Smith

The fullname function can be used by any user who has been granted EXECUTE permission for the function.

Dropping user-defined functions

Once a user-defined function is created, it remains in the database until it is explicitly removed. Only the owner of the function or a user with DBA authority can drop a function from the database.

The following statement removes the function fullname from the database:

```
DROP FUNCTION fullname
```

Permissions to execute user-defined functions

A user-defined function is owned by the user who created it and that user can execute it without permission. Permission to execute it can be granted to other users using the GRANT EXECUTE command.

For example, the creator of the function fullname could allow another_user to use fullname with the statement:

```
GRANT EXECUTE ON fullname TO another_user
```

The following statement revokes permission to use the function:

```
REVOKE EXECUTE ON fullname FROM another_user
```

For more information on managing user permissions on functions, see "Granting permissions on procedures" in the chapter "Managing User IDs and Permissions".

Introduction to triggers

Triggers are used whenever referential integrity and other declarative constraints are not sufficient (see the chapter "Ensuring Data Integrity" and "CREATE TABLE statement" in the chapter "Watcom-SQL Statements").

You may want to enforce a more complex form of referential integrity involving more detailed checking, or you may want to enforce checking on new data but allow legacy data to violate constraints. Another use for triggers is in logging the activity on database tables, independent of the applications using the database.

Trigger execution permissions
Triggers execute with the permissions of the owner of the associated table, not the user ID whose actions cause the trigger to fire. A trigger can modify rows in a table that a user could not modify directly.

Triggers can be defined on one or more of the following triggering actions:

Action	Description
INSERT	The trigger is invoked whenever a new row is inserted into the table associated with the trigger
DELETE	The trigger is invoked whenever a row of the associated table is deleted.
UPDATE	The trigger is invoked whenever a row of the associated table is updated.
UPDATE OF column-list	The trigger is invoked whenever a row of the associated table is updated such that a column in the *column-list* has been modified

Triggers can be defined as row-level triggers or statement-level triggers. Row-level triggers can execute BEFORE or AFTER each row modified by the triggering insert, update, or delete operation is changed. Statement-level triggers execute after the entire operation is performed.

Flexibility in trigger execution time is particularly useful for triggers that rely on referential integrity actions such as cascaded updates or deletes being carried out, or not, as they execute.

If an error occurs while a trigger is executing, the operation that fired the trigger fails. INSERT, UPDATE, and DELETE are **atomic** operations (see "Atomic compound statements" on page 285). When they fail, all effects of the statement (including the effects of triggers and any procedures called by triggers) are undone.

Creating triggers

You create triggers using the CREATE TRIGGER statement. You must have RESOURCE authority in order to create a trigger and you must have ALTER permissions on the table associated with the trigger. For information about ALTER permissions, see "Granting permissions on tables and views" in the chapter "Managing User IDs and Permissions". For information about RESOURCE permissions, see "Granting DBA and resource authority" in the chapter "Managing User IDs and Permissions".

The body of a trigger consists of a compound statement (see "Using compound statements" on page 283): a set of semicolon-delimited SQL statements bracketed by a BEGIN and an END statement.

COMMIT and ROLLBACK and some ROLLBACK TO SAVEPOINT statements are not permitted within a trigger (see "Transactions and savepoints in procedures and triggers" on page 308).

A row-level INSERT trigger example

The following trigger is an example of a row-level INSERT trigger. It checks that the birthdate entered for a new employee is reasonable:

```
CREATE TRIGGER check_birth_date
   AFTER INSERT ON Employee
REFERENCING NEW AS new_employee
FOR EACH ROW
BEGIN
   DECLARE err_user_error EXCEPTION
   FOR SQLSTATE '99999';
   IF new_employee.birth_date > 'June 6, 1994' THEN
      SIGNAL err_user_error;
   END IF;
END
```

This trigger is fired just after any row is inserted into the employee table. It detects and disallows any new rows that correspond to birth dates later than June 6, 1994.

The phrase REFERENCING NEW AS new_employee allows statements in the trigger code to refer to the data in the new row using the alias new_employee.

Signaling an error causes the triggering statement as well as any previous effects of the trigger to be undone.

For an INSERT statement that adds many rows to the employee table, the check_birth_date trigger is fired once for each new row. If the trigger fails for any of the rows, all effects of the INSERT statement are rolled back.

You can specify that the trigger fire before the row is inserted rather than after by changing the first line of the example to:

```
CREATE TRIGGER mytrigger BEFORE INSERT ON Employee
```

The REFERENCING NEW clause refers to the inserted values of the row; it is independent of the timing (BEFORE or AFTER) of the trigger.

A row-level DELETE trigger example

The following CREATE TRIGGER statement defines a row-level DELETE trigger:

```
CREATE TRIGGER mytrigger BEFORE DELETE ON employee
REFERENCING OLD AS oldtable
FOR EACH ROW
BEGIN
   ...
END
```

The REFERENCING OLD clause enables the delete trigger code to refer to the values in the row being deleted using the alias oldtable.

You can specify that the trigger fire after the row is deleted rather than before, by changing the first line of the example to:

```
CREATE TRIGGER mytrigger BEFORE DELETE ON employee
```

The REFERENCING OLD clause is independent of the timing (BEFORE or AFTER) of the trigger.

A statement-level UPDATE trigger example

The following CREATE TRIGGER statement is appropriate for statement-level UPDATE triggers:

```
CREATE TRIGGER mytrigger AFTER UPDATE ON employee
REFERENCING NEW AS table_after_update
            OLD AS table_before_update
FOR EACH STATEMENT
BEGIN
   ...
END
```

The REFERENCING NEW and REFERENCING OLD clause allows the UPDATE trigger code to refer to both the old and new values of the rows being updated. Columns in the new row are referred to with the table alias table_after_update and columns in the old row are referred to with the table alias table_before_update.

The REFERENCING NEW and REFERENCING OLD clause has a slightly different meaning for statement-level and row-level triggers. For statement-level triggers the REFERENCING OLD or NEW aliases are table aliases, while in row-level triggers they refer to the row being altered.

Executing triggers

Triggers are executed automatically whenever an INSERT, UPDATE, or DELETE operation is performed on the table named in the trigger. A row-level trigger is fired once for each row that is affected, while a statement-level trigger is fired once for the entire statement.

When an INSERT, UPDATE, or DELETE fires a trigger, the order of operation is as follows:

1 Any BEFORE triggers are fired.

2 Any referential actions are performed.

3 The operation itself is performed.

4 Any AFTER triggers are fired.

If any of the steps encounters an error that is not handled within a procedure or trigger, the preceding steps are undone, the subsequent steps are not performed, and the operation that fired the trigger fails.

Dropping triggers

Once a trigger is created, it remains in the database until it is explicitly removed. You must have ALTER permissions on the table associated with the trigger.

The following statement removes the trigger mytrigger from the database:

```
DROP TRIGGER mytrigger
```

Trigger execution permissions

You cannot grant permissions to execute a trigger, as triggers are not executed by users: they are fired by the database engine in response to actions on the database. Nevertheless, a trigger does have permissions associated with it as it executes, defining its right to carry out certain actions.

Triggers execute using the permissions of the owner of the table on which they are defined, not the permissions of the user that caused the trigger to fire, and not the permissions of the user that created the trigger.

When a trigger refers to a table, it uses the group memberships of the table creator to locate tables with no explicit owner name specified. For example, if a trigger on user_1.Table_A references Table_B and does not specify the owner of Table_B, then either Table_B must have been created by user_1 or user_1 must be a member of a group (directly or indirectly) that is the owner of Table_B. If neither condition is met, a table not found message results when the trigger is fired.

Also, user_1 must have permission to carry out the operations specified in the trigger.

Introduction to batches

A simple batch consists of a set of SQL statements, separated by semicolons. For example, the following set of statements form a batch:

Create an Eastern Sales department, and transfer all sales reps from Massachusetts to that department.

```
INSERT
INTO department ( dept_id, dept_name )
VALUES ( 220, 'Eastern Sales' ) ;
UPDATE employee
SET dept_id = 220
WHERE dept_id = 200
AND state = 'MA' ;
COMMIT ;
```

You can include this set of statements in an application and execute them together.

ISQL and batches
A list of semicolon-separated statements, such as the above, is parsed by ISQL prior to sending to the database engine. In this case, ISQL sends each statement individually to the engine, not as a batch. Unless you have such parsing code in your application, the statements would be sent and treated as a batch. Other batches described below are sent by ISQL as a batch, not as a set of individual statements. Putting a BEGIN and END around a set of statements causes ISQL to treat them as a batch.

Many statements used in procedures and triggers can also be used in batches. You can use control statements (CASE, IF, LOOP, and so on), including compound statements (BEGIN and END), in batches. Compound statements can include declarations of variables, exceptions, temporary tables, or cursors inside the compound statement.

The following batch creates a table only if a table of that name does not already exist:

```
BEGIN
   IF NOT EXISTS (
      SELECT * FROM SYSTABLE
      WHERE table_name = 't1' ) THEN
      CREATE TABLE t1 (
            firstcol INT PRIMARY KEY,
            secondcol CHAR( 30 )
      ) ;
   ELSE
      MESSAGE 'Table t1 already exists' ;
   END IF
```

```
END
```

If you run this batch twice from ISQL, it creates the table the first time you run it, and prints the message on the server or engine message window the next time you run it.

Control statements

There are a number of control statements for logical flow and decision making in the body of the procedure or trigger, or in a batch. The following is a list of control statements available.

Control statement	Syntax

Compound statements	`BEGIN [ ATOMIC ]`
	`  statement-list`
	`END`

Conditional execution: IF	`IF condition THEN`
	`  statement-list`
	`ELSEIF condition THEN`
	`  statement-list`
	`ELSE`
	`  statement-list`
	`END IF`

Conditional execution: CASE	`CASE expression`
	`WHEN value THEN`
	`  statement-list`
	`WHEN value THEN`
	`  statement-list`
	`ELSE`
	`  statement-list`
	`END CASE`

Repetition: WHILE, LOOP	`WHILE condition LOOP`
	`  statement-list`
	`END LOOP`

Repetition: FOR cursor loop	`FOR`
	`  statement-list`
	`END FOR`

Break: LEAVE	`LEAVE label`
CALL	`CALL procname( arg, ... )`

For complete descriptions of each, see the entries in "Watcom-SQL Language Reference".

Using compound statements

A compound statement starts with the keyword BEGIN and ends with the keyword END. The body of a procedure or trigger is a **compound statement**. Compound statements can also be used in batches. Compound statements can be nested, and combined with other control statements to define execution flow in procedures and triggers or in batches.

A compound statement allows a set of SQL statements to be grouped together and treated as a unit. SQL statements within a compound statement should be delimited with semicolons.

Declarations in compound statements

Local declarations in a compound statement immediately follow the BEGIN keyword. These local declarations exist only within the compound statement. The following may be declared within a compound statement:

- Variables
- Cursors
- Temporary tables
- Exceptions (error identifiers)

Local declarations can be referenced by any statement in that compound statement, or in any compound statement nested within it. Local declarations are not visible to other procedures called from within the compound statement.

The following user-defined function illustrates local declarations of variables.

The customer table includes some Canadian customers sprinkled among those from the USA, but there is no country column. The user-defined function nationality uses the fact that the US zip code is numeric while the Canadian postal code begins with a letter to distinguish Canadian and US customers.

```
CREATE FUNCTION nationality( cust_id INT )
RETURNS CHAR( 20 )
BEGIN
   DECLARE natl CHAR(20);
   IF cust_id IN ( SELECT id FROM customer
                WHERE LEFT(zip,1) > '9') THEN
         SET natl = 'CDN';
   ELSE
      SET natl = 'USA';
   END IF;
   RETURN ( natl );
END
```

This example declares a variable natl to hold the nationality string, uses a SET statement to set a value for the variable, and returns the value of the natl string to the calling environment.

The following query lists all Canadian customers in the customer table:

```
SELECT *
FROM customer
WHERE nationality(id) = 'CDN'
```

Declarations of cursors and exceptions are discussed in later sections.

Atomic compound statements

An **atomic** statement is a statement that is executed completely or not at all. For example, an UPDATE statement that updates thousands of rows might encounter an error after updating many rows. If the statement does not complete, all changes are undone. The UPDATE statement is atomic.

All noncompound SQL statements are atomic. A compound statement can be made atomic by adding the keyword ATOMIC after the BEGIN keyword.

```
BEGIN ATOMIC
   UPDATE employee
   SET manager_ID = 501
   WHERE emp_ID = 467;
   UPDATE employee
   SET birth_date = 'bad_data';
END
```

In this example, the two update statements are part of an atomic compound statement. They must either succeed or fail as one. The first update statement would succeed. The second one causes a data conversion error since the value being assigned to the birth_date column cannot be converted to a date.

The atomic compound statement fails and the effect of both UPDATE statements is undone. Even if the currently executing transaction is eventually committed, neither statement in the atomic compound statement takes effect.

COMMIT and ROLLBACK and some ROLLBACK TO SAVEPOINT statements are not permitted within an atomic compound statement (see "Transactions and savepoints in procedures and triggers" on page 308).

There is a case where some, but not all, of the statements within an atomic compound statement are executed. This is when an error occurs, and is handled by an exception handler within the compound statement.

For more information, see "Using exception handlers in procedures and triggers" on page 304.

The structure of procedures and triggers

The body of a procedure or trigger consists of a compound statement as discussed in "Using compound statements" on page 283. A compound statement consists of a BEGIN and an END, enclosing a set of SQL statements. The statements must be delimited by semicolons.

☞ The SQL statements that can occur in procedures and triggers are described in "SQL statements allowed in procedures and triggers" next.

☞ Procedures and triggers can contain control statements, which are described in "Control statements" on page 283.

SQL statements allowed in procedures and triggers

Many SQL statements are allowed within procedures and triggers, including the following:

- SELECT, UPDATE, DELETE, INSERT and SET variable.
- The CALL statement to execute other procedures.
- Control statements (see "Control statements" on page 283).
- Cursor statements (see "Using cursors in procedures and triggers" on page 295).
- Exception handling statements (see "Using exception handlers in procedures and triggers" on page 304).
- The EXECUTE IMMEDIATE statement.

Some SQL statements are not allowed within procedures and triggers. These include the following:

- CONNECT statement
- DISCONNECT statement.

COMMIT, ROLLBACK and SAVEPOINT statements are allowed within procedures and triggers with certain restrictions (see "Transactions and savepoints in procedures and triggers" on page 308).

☞ For details, see the Usage for each SQL statement in the chapter "Watcom-SQL Language Reference".

Declaring procedure parameters

Procedure parameters, or arguments, are specified as a list in the CREATE PROCEDURE statement. Parameter names must conform to the rules for other database identifiers such as column names. They must be one of the types supported by SQL Anywhere (see "SQL Anywhere Data Types"), and must be prefixed with one of the keywords IN, OUT or INOUT. These keywords have the following meanings:

- **IN** The argument is an expression that provides a value to the procedure.
- **OUT** The argument is a variable that could be given a value by the procedure.
- **INOUT** The argument is a variable that provides a value to the procedure, and could be given a new value by the procedure.

Default values can be assigned to procedure parameters in the CREATE PROCEDURE statement. The default value must be a constant, which may be NULL. For example, the following procedure uses the NULL default for an IN parameter to avoid executing a query that would have no meaning:

```
CREATE PROCEDURE
CustomerProducts( IN customer_id
                  INTEGER DEFAULT NULL )
RESULT ( product_id INTEGER,
         quantity_ordered INTEGER )
BEGIN
   IF customer_id IS NULL THEN
      RETURN;
   ELSE
      SELECT    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
```

The following statement causes the DEFAULT NULL to be assigned, and the procedure RETURNs instead of executing the query.

```
CALL customer_products();
```

Passing parameters to procedures

You can take advantage of default values of stored procedure parameters with either of two forms of the CALL statement.

If the optional parameters are at the end of the argument list in the CREATE PROCEDURE statement, they may be omitted from the CALL statement. As an example, consider a procedure with three INOUT parameters:

```
CREATE PROCEDURE SampleProc( INOUT var1 INT
                                DEFAULT 1,
                    INOUT var2 int DEFAULT 2,
                    INOUT var3 int DEFAULT 3 )
...
```

We assume that the calling environment has set up three variables to hold the values passed to the procedure:

```
CREATE VARIABLE V1 INT;
CREATE VARIABLE V2 INT;
CREATE VARIABLE V3 INT;
```

The procedure SampleProc may be called supplying only the first parameter as follows:

```
CALL SampleProc( V1 )
```

in which case the default values are used for *var2* and *var3*.

A more flexible method of calling procedures with optional arguments is to pass the parameters by name. The SampleProc procedure may be called as follows:

```
CALL SampleProc( var1 = V1, var3 = V3 )
```

or as follows:

```
CALL SampleProc( var3 = V3, var1 = V1 )
```

and so on.

Passing parameters to functions

User-defined functions are not invoked with the CALL statement, but are used in the same manner that built-in functions are. For example, the following statement uses the fullname function defined in "Creating user-defined functions" on page 273 to retrieve the names of employees:

❖ **To list the names of all employees:**

- Type the following:

```
SELECT fullname(emp_fname, emp_lname) AS Name
FROM employee
```

Name
Fran Whitney
Matthew Cobb
Philip Chin
Julie Jordan
Robert Breault
...

Notes

- Default parameters can be used in calling functions. However, parameters cannot be passed to functions by name.
- Parameters are passed by value, not by reference. Even if the function changes the value of the parameter, this change is not returned to the calling environment.
- Output parameters cannot be used in user-defined functions.
- User-defined functions cannot return result sets.

Returning results from procedures

Procedures can return results that are a single row of data, or multiple rows. In the former case, results can be passed back as arguments to the procedure, in the latter case, results are passed back as result sets. Procedures can also return a single value given in the RETURN statement.

For simple examples of how to return results from procedures, see "Introduction to procedures" on page 268. For information about returning a single row, see "Returning results as procedure parameters" below. For information about returning multiple rows, see "Returning result sets from procedures" on page 292.

Returning a value using the RETURN statement

A single value can be returned to the calling environment using the RETURN statement, which causes an immediate exit from the procedure. The RETURN statement takes the form:

```
RETURN expression
```

The value of the supplied expression is returned to the calling environment. To save the return value in a variable, an extension of the CALL statement is used:

```
CREATE VARIABLE returnval INTEGER ;
returnval = CALL myproc() ;
```

Returning results as procedure parameters

Procedures can return results to the calling environment in the parameters to the procedure.

Within a procedure, parameters and variables can be assigned values in one of the following ways:

- The parameter can be assigned a value using the SET statement.
- The parameter can be assigned a value using a SELECT statement with an INTO clause.

Using the SET statement

The following somewhat artificial procedure returns a value in an OUT parameter that is assigned using a SET statement:

```
CREATE PROCEDURE greater ( IN a INT,
                           IN b INT,
```

```
                                        OUT c INT)
BEGIN
   IF a > b THEN
      SET c = a;
   ELSE
      SET c = b;
   END IF ;
END
```

Using single-row SELECT statements

Single-row queries retrieve at most one row from the database. This type of query is achieved by a SELECT statement with an INTO clause. The INTO clause follows the select list and precedes the FROM clause. It contains a list of variables to receive the value for each select list item. There must be the same number of variables as there are select list items.

When a SELECT statement is executed, the database engine retrieves the results of the select statement and places the results in the variables. If the query results contain more than one row, the database engine returns an error. For queries returning more than one row, **cursors** must be used. For information about returning more than one row from a procedure, see "Returning result sets from procedures" on page 292.

If the query results in no rows being selected, a row not found warning is returned.

The following procedure returns the results of a single-row SELECT statement in the procedure parameters.

❖ **To return the number of orders placed by a given customer:**

- Type the following:

```
CREATE PROCEDURE OrderCount (IN customer_ID INT,
                             OUT Orders INT)
BEGIN
   SELECT COUNT("dba".sales_order.id)
      INTO Orders
   FROM "dba".customer
      KEY LEFT OUTER JOIN "dba".sales_order
   WHERE "dba".customer.id = customer_ID;
END
```

This procedure can be tested in ISQL using the following statements, which show the number of orders placed by the customer with ID 102:

```
CREATE VARIABLE orders INT;
CALL OrderCount ( 102, orders );
SELECT orders;
```

Notes

- The *customer_ID* parameter is declared as an IN parameter. This parameter holds the customer ID that is passed in to the procedure.

- The *Orders* parameter is declared as an OUT parameter. It holds the value of the orders variable that is returned to the calling environment.
- No DECLARE statement is required for the *Orders* variable, as it is declared in the procedure argument list.
- The SELECT statement returns a single row and places it into the variable *Orders*.

Returning result sets from procedures

If a procedure returns more than one row of results to the calling environment, it does so using result sets.

The following procedure returns a list of customers who have placed orders, together with the total value of the orders placed. The procedure does not list customers who have not placed orders.

```
CREATE PROCEDURE ListCustomerValue ()
RESULT ("Company" CHAR(36), "Value" INT)
BEGIN
   SELECT company_name,
      CAST( sum(  sales_order_items.quantity *
                  product.unit_price)
                  AS INTEGER ) AS value
   FROM customer
      INNER JOIN sales_order
      INNER JOIN sales_order_items
      INNER JOIN product
   GROUP BY company_name
   ORDER BY value desc;
END
```

- Type the following:

```
CALL ListCustomerValue ()
```

Company	Value
Chadwicks	8076
Overland Army Navy	8064
Martins Landing	6888
Sterling & Co.	6804
Carmel Industries	6780
...	

Notes

- The number of variables in the RESULT list must match the number of the SELECT list items. Automatic data type conversion is carried out where possible if data types do not match.
- The RESULT clause is part of the CREATE PROCEDURE statement, and is not followed by a command delimiter.
- The names of the SELECT list items do not need to match those of the RESULT list.
- When testing this procedure, ISQL opens a cursor to handle the results. The cursor is left open following the SELECT statement, in case the procedure returns more than one result set. You should type RESUME to complete the procedure and close the cursor.

Returning multiple result sets from procedures

A procedure can return more than one result set to the calling environment. If a RESULT clause is employed, the result sets must be compatible: they must have the same number of items in the SELECT lists, and the data types must all be of types that can be automatically converted to the data types listed in the RESULT list.

The following procedure lists the names of all employees, customers, and contacts listed in the database:

```
CREATE PROCEDURE ListPeople()
RESULT ( lname CHAR(36), fname CHAR(36) )
BEGIN
   SELECT emp_lname, emp_fname
   FROM employee;
   SELECT lname, fname
   FROM customer;
   SELECT last_name, first_name
   FROM contact;
END
```

Notes

- To test this procedure in ISQL, enter the statement CALL ListPeople (); You must enter a RESUME statement after each of the three result sets is displayed in the ISQL data window to continue, and then complete, the procedure.
- A command delimiter is required after the first two SELECT statements. It is optional after the final statement in a statement list.

Returning variable result sets from procedures

The RESULT clause is optional in procedures. Omitting the result clause allows you to write procedures that return different result sets, with different numbers or types of columns, depending on how they are executed.

If you are not using this feature of variable result sets, it is recommended that you employ a RESULT clause, for performance reasons.

For example, the following procedure returns two columns if the input variable is Y, but only one column otherwise:

```
CREATE PROCEDURE names( IN formal char(1))
BEGIN
   IF formal = 'y' THEN
      SELECT emp_lname, emp_fname
      FROM employee
   ELSE
      SELECT emp_fname
      FROM employee
   END IF
END
```

The use of variable result sets in procedures is subject to some limitations, depending on the interface used by the client application.

- **Embedded SQL** You must DESCRIBE the procedure call after the cursor for the result set is opened, but before any rows are returned, in order to get the proper shape of result set.

  For information about the DESCRIBE statement, see "DESCRIBE statement" in the chapter "Watcom-SQL Statements".

- **ODBC** Variable result set procedures can be used by ODBC applications. The proper description of the variable result sets is carried out by the SQL Anywhere ODBC driver.

- **Open Client applications** Variable result set procedures can be used by Open Client applications using the Open Server Gateway. The proper description of the variable result sets is carried out by the Open Server Gateway.

- **ISQL** ISQL does not support calling of variable result set procedures, and so cannot be used for testing this feature.

Using cursors in procedures and triggers

Cursors are used to retrieve rows one at a time from a query or stored procedure that has multiple rows in its result set. A **cursor** is a handle or an identifier for the query or procedure, and for a current position within the result set.

Cursor management overview

Managing a cursor is similar to managing a file in a programming language. The following steps are used to manage cursors:

1 Declare a cursor for a particular select statement or procedure using the DECLARE statement.

2 Open the cursor using the OPEN statement.

3 Use the FETCH statement to retrieve results one row at a time from the cursor.

4 Records are usually fetched until the row not found warning is returned, signaling the end of the result set.

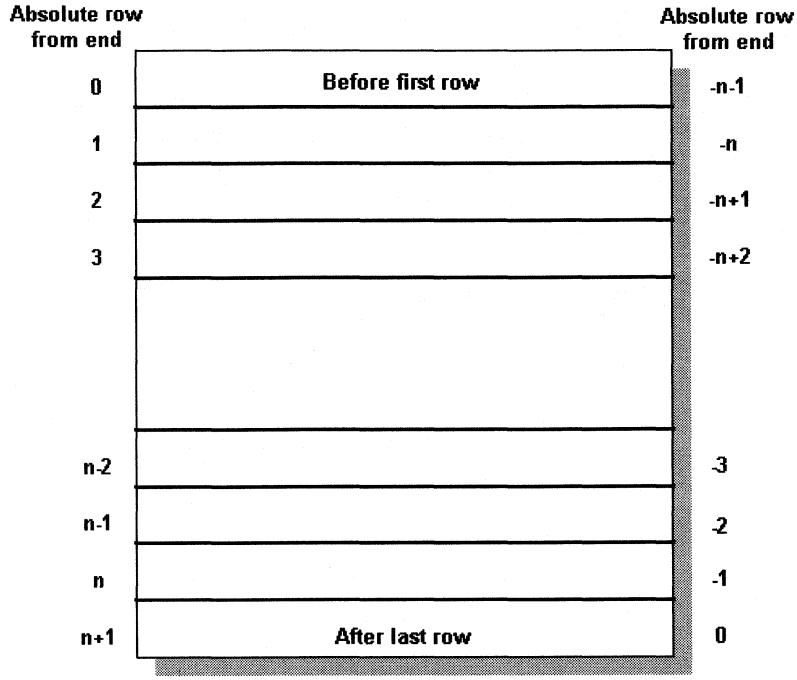
5 Close the cursor using the CLOSE statement.

By default, cursors are automatically closed at the end of a transaction (on COMMIT or ROLLBACK statements). Cursors that are opened using the WITH HOLD clause will be kept open for subsequent transactions until they are explicitly closed.

Cursor positioning

A cursor can be positioned at one of three places:

- On a row
- Before the first row
- After the last row



When a cursor is opened, it is positioned before the first row. The cursor position can be moved using the FETCH command (see "FETCH statement" in the chapter "Watcom-SQL Statements"). It can be positioned to an absolute position either from the start or from the end of the query results (using FETCH ABSOLUTE, FETCH FIRST, or FETCH LAST). It can also be moved relative to the current cursor position (using FETCH RELATIVE, FETCH PRIOR, or FETCH NEXT). The NEXT keyword is the default qualifier for the FETCH statement.

There are special positioned versions of the UPDATE and DELETE statements that can be used to update or delete the row at the current position of the cursor. If the cursor is positioned before the first row or after the last row, a no current row of cursor error will be returned.

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" in the chapter "Monitoring and Improving Performance" 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). Using STATIC SCROLL cursors alleviates these problems but is more expensive.

Using cursors on SELECT statements in procedures

The following procedure uses a cursor on a SELECT statement. It illustrates several features of the stored procedure language. It is based on the same query used in the ListCustomerValue procedure described in "Returning result sets from procedures" on page 292.

```
CREATE PROCEDURE TopCustomerValue
    (  OUT TopCompany CHAR(36),
       OUT TopValue INT )
BEGIN
    -- 1. Declare the "error not found" exception
    DECLARE err_notfound
       EXCEPTION FOR SQLSTATE '02000';
    -- 2. Declare variables to hold
    --    each company name and its value
    DECLARE ThisName CHAR(36);
    DECLARE ThisValue INT;
    -- 3. Declare the cursor ThisCompany
    --    for the query
    DECLARE ThisCompany CURSOR FOR
    SELECT company_name,
          CAST( sum( sales_order_items.quantity *
                product.unit_price ) AS INTEGER )
          AS value
    FROM customer
       INNER JOIN sales_order
       INNER JOIN sales_order_items
       INNER JOIN product
    GROUP BY company_name;
```

```
    -- 4. Initialize the values of TopValue
    SET TopValue = 0;
    -- 5. Open the cursor
    OPEN ThisCompany;
    -- 6. Loop over the rows of the query
    CompanyLoop:
    LOOP
       FETCH NEXT ThisCompany
          INTO ThisName, ThisValue;
       IF SQLSTATE = err_notfound THEN
          LEAVE CompanyLoop;
       END IF;
       IF ThisValue > TopValue THEN
          SET TopCompany = ThisName;
          SET TopValue = ThisValue;
       END IF;
    END LOOP CompanyLoop;
    -- 7. Close the cursor
    CLOSE ThisCompany;
END
```

Notes

The TopCustomerValue procedure has the following notable features:

- The "error not found" exception is declared. This exception is used to detect when a loop over the results of a query has completed.

  For more information about exceptions, see "Errors and warnings in procedures and triggers" on page 300.

- Two local variables ThisName and ThisValue are declared to hold the results from each row of the query.
- The cursor ThisCompany is declared. The SELECT statement produces a list of company names and the total value of the orders placed by that company.
- The value of TopValue is set to an initial value of 0, for later use in the loop.
- The ThisCompany cursor is opened.
- The LOOP statement loops over each row of the query, placing each company name in turn into the variables ThisName and ThisValue. If ThisValue is greater than the current top value, TopCompany and TopValue are reset to ThisName and ThisValue.
- The cursor is closed at the end of the procedure.

The LOOP construct in the TopCompanyValue procedure is a standard form, exiting after the last row is processed. You can rewrite this procedure in a more compact form using a FOR loop. The FOR statement combines several aspects of the above procedure into a single statement.

```
CREATE PROCEDURE TopCustomerValue2(
      OUT TopCompany CHAR(36),
      OUT TopValue INT )
BEGIN
   -- Initialize the TopValue variable
   SET TopValue = 0;
   -- Do the For Loop
   CompanyLoop:
   FOR CompanyFor AS ThisCompany
      CURSOR FOR
      SELECT company_name AS ThisName ,
         CAST( sum( sales_order_items.quantity *
               product.unit_price ) AS INTEGER )
         AS ThisValue
      FROM customer
         INNER JOIN sales_order
         INNER JOIN sales_order_items
         INNER JOIN product
      GROUP BY ThisName
   DO
      IF ThisValue > TopValue THEN
         SET TopCompany = ThisName;
         SET TopValue = ThisValue;
         END IF;
   END FOR CompanyLoop;
END
```

Errors and warnings in procedures and triggers

After an application program executes a SQL statement, it can examine a **return code**. This return code indicates whether the statement executed successfully or failed and gives the reason for the failure. The same mechanism can be used to indicate the success or failure of a CALL statement to a procedure.

SQL Anywhere supports the SQLCODE and SQLSTATE status descriptions. For full descriptions of SQLCODE and SQLSTATE error and warning values and their meanings, see the chapter "SQL Anywhere Database Error Messages". Whenever a SQL statement is executed, a value is placed in special procedure variables called SQLSTATE and SQLCODE. That value indicates whether or not there were any unusual conditions encountered while the statement was being performed. You can check the value of SQLSTATE or SQLCODE in an IF statement following a SQL statement, and take actions depending on whether the statement succeeded or failed.

For example, the SQLSTATE variable can be used to indicate if a row is successfully fetched. The TopCustomerValue procedure presented in section "Using cursors on SELECT statements in procedures" on page 297 used the SQLSTATE test to detect that all rows of a SELECT statement had been processed.

 Possible values for the SQLSTATE and SQLCODE variables are listed in the chapter "SQL Anywhere Database Error Messages".

Default error handling in procedures and triggers

This section describes how SQL Anywhere handles errors that occur during a procedure execution, if you have no error handling built in to the procedure.

 If you want to have different behavior from that described in this section, you can use exception handlers, described in "Using exception handlers in procedures and triggers" on page 304. Warnings are handled in a slightly different manner from errors: for a description, see "Default handling of warnings in procedures and triggers" on page 303.

There are two ways of handling errors without using explicit error handling:

- **Default error handling** The procedure or trigger fails and returns an error code to the calling environment.

- **ON EXCEPTION RESUME** If the ON EXCEPTION RESUME clause is included in the CREATE PROCEDURE statement, the procedure carries on executing after an error, resuming at the statement following the one causing the error.

Default error handling

Generally, if a SQL statement in a procedure or trigger fails, the procedure or trigger terminates execution and control is returned to the application program with an appropriate setting for the SQLSTATE and SQLCODE values. This is true even if the error occurred in a procedure or trigger invoked directly or indirectly from the first one. In the case of a trigger, the operation causing the trigger is also undone and the error is returned to the application.

The following demonstration procedures show what happens when an application calls the procedure OuterProc and OuterProc in turn calls the procedure InnerProc, which then encounters an error.

```
CREATE PROCEDURE OuterProc()
BEGIN
   MESSAGE 'Hello from OuterProc.';
   CALL InnerProc();
   MESSAGE 'SQLSTATE set to ',
      SQLSTATE,' in OuterProc.'
END
CREATE PROCEDURE InnerProc()
   BEGIN
      DECLARE column_not_found
         EXCEPTION FOR SQLSTATE '52003';
      MESSAGE 'Hello from InnerProc.';
      SIGNAL column_not_found;
   MESSAGE 'SQLSTATE set to ',
      SQLSTATE, ' in InnerProc.';
END
```

Notes

- The DECLARE statement in InnerProc declares a symbolic name for one of the predefined SQLSTATE values associated with error conditions already known to the database engine. The DECLARE statement does not take any other action.
- The MESSAGE statement sends a message to the database engine or server message window.
- The SIGNAL statement generates an error condition from within the InnerProc procedure.

The following statement executes the OuterProc procedure:

```
CALL OuterProc();
```

The message window of the database engine then displays the following:

```
Hello from OuterProc.
```

Hello from InnerProc.

No statements following the SIGNAL statement in InnerProc are executed: InnerProc immediately passes control back to the calling environment, which in this case is the procedure OuterProc. The error condition is returned to the calling environment to be handled there. For example, the ISQL utility handles the error by displaying a message window describing the error.

The TRACEBACK function provides a list of the statements that were executing when the error occurred. You can use the TRACEBACK function from ISQL by typing the following statement:

```
SELECT TRACEBACK(*)
```

Error handling with ON EXCEPTION RESUME

If the ON EXCEPTION RESUME clause is included in the CREATE PROCEDURE statement, the procedure does not return control to the calling environment when an error occurs. Instead, it continues executing, resuming at the statement after the one causing the error.

The following demonstration procedures show what happens when an application calls the procedure OuterProc; and OuterProc in turn calls the procedure InnerProc, which then encounters an error. These demonstration procedures are based on those used earlier in this section:

```
CREATE PROCEDURE OuterProc()
ON EXCEPTION RESUME
BEGIN
   MESSAGE 'Hello from OuterProc.';
   CALL InnerProc();
   MESSAGE 'SQLSTATE set to ',
      SQLSTATE,' in OuterProc.';
END;

CREATE PROCEDURE InnerProc()
ON EXCEPTION RESUME
BEGIN
   DECLARE column_not_found
      EXCEPTION FOR SQLSTATE '52003';
   MESSAGE 'Hello from InnerProc.';
   SIGNAL column_not_found;
   MESSAGE 'SQLSTATE set to ',
      SQLSTATE, ' in InnerProc.';
END
```

The following statement executes the OuterProc procedure:

```
CALL OuterProc();
```

The message window of the database engine then displays the following:

Hello from OuterProc.

Hello from InnerProc.

SQLSTATE set to 52003 in InnerProc.

SQLSTATE set to 52003 in OuterProc.

Default handling of warnings in procedures and triggers

Warnings are handled differently from errors. While the default action for errors is to set a value for the SQLSTATE and SQLCODE variables, and return control to the calling environment, the default action for warnings is to set the SQLSTATE and SQLCODE values and continue execution of the procedure.

The following demonstration procedures illustrate default handling of warnings. These demonstration procedures are based on those used in "Default error handling in procedures and triggers" on page 300. In this case, the SIGNAL statement generates a row not found condition, which is a warning rather than an error.

```
CREATE PROCEDURE OuterProc()
BEGIN
   MESSAGE 'Hello from OuterProc.';
   CALL InnerProc();
   MESSAGE 'SQLSTATE set to ',
      SQLSTATE,' in OuterProc.';
END
CREATE PROCEDURE InnerProc()
BEGIN
   DECLARE row_not_found
      EXCEPTION FOR SQLSTATE '02000';
   MESSAGE 'Hello from InnerProc.';
   SIGNAL row_not_found;
   MESSAGE 'SQLSTATE set to ',
   SQLSTATE, ' in InnerProc.';
END
```

The following statement executes the OuterProc procedure:

```
CALL OuterProc();
```

The message window of the database engine then displays the following:

Hello from OuterProc.

Hello from InnerProc.

SQLSTATE set to 02000 in InnerProc.

SQLSTATE set to 02000 in OuterProc.

The procedures both continued executing after the warning was generated, with SQLSTATE set to the value set by the warning (02000).

Using exception handlers in procedures and triggers

It is often desirable to intercept certain types of errors and handle them within a procedure or trigger, rather than pass the error back to the calling environment. This is done through the use of an **exception handler**.

An exception handler is defined with the EXCEPTION part of a compound statement (see "Using compound statements" on page 283). The exception handler is executed whenever an error occurs in the compound statement. Unlike errors, warnings do not cause exception handling code to be executed. Exception handling code is also executed if an error is encountered in a nested compound statement or in a procedure or trigger that has been invoked anywhere within the compound statement.

The demonstration procedures used to illustrate exception handling are based on those used in "Default error handling in procedures and triggers" on page 300. In this case, additional code is added to handle the column not found error in the InnerProc procedure.

```
CREATE PROCEDURE OuterProc()
BEGIN
   MESSAGE 'Hello from OuterProc.';
   CALL InnerProc();
   MESSAGE 'SQLSTATE set to ',
      SQLSTATE,' in OuterProc.'
END
CREATE PROCEDURE InnerProc()
BEGIN
   DECLARE column_not_found
      EXCEPTION FOR SQLSTATE '52003';
   MESSAGE 'Hello from InnerProc.';
   SIGNAL column_not_found;
   MESSAGE 'Line following SIGNAL.';
   EXCEPTION
      WHEN column_not_found THEN
         MESSAGE 'Column not found handling.';
      WHEN OTHERS THEN
   RESIGNAL ;
END
```

The EXCEPTION statement declares the exception handler itself. The lines following the EXCEPTION statement are not executed unless an error occurs. Each WHEN clause specifies an exception name (declared with a DECLARE statement) and the statement or statements to be executed in the event of that exception. The WHEN OTHERS THEN clause specifies the statement(s) to be executed when the exception that occurred is not in the preceding WHEN clauses.

In this example, the statement RESIGNAL passes the exception on to a higher-level exception handler. RESIGNAL is the default action if WHEN OTHERS THEN is not specified in an exception handler.

The following statement executes the OuterProc procedure:

```
CALL OuterProc();
```

The message window of the database engine then displays the following:

Hello from OuterProc.

Hello from InnerProc.

Column not found handling.

SQLSTATE set to 00000 in OuterProc.

Notes

- The lines following the SIGNAL statement in InnerProc are not executed; instead, the EXCEPTION statements are executed.
- As the error encountered was a column not found error, the MESSAGE statement included to handle the error is executed, and SQLSTATE is reset to zero (indicating no errors).
- After the exception handling code is executed, control is passed back to OuterProc, which proceeds as if no error was encountered.
- You should not use ON EXCEPTION RESUME together with explicit exception handling. The exception handling code is not executed if ON EXCEPTION RESUME is included.
- If the error handling code for the column not found exception is simply a RESIGNAL statement, control is passed back to the OuterProc procedure with SQLSTATE still set at the value 52003. This is just as if there were no error handling code in InnerProc. As there is no error handling code in OuterProc, the procedure fails.

Exception handling and atomic compound statements

When an exception is handled inside a compound statement, the compound statement completes without an active exception and the changes before the exception are not undone. This is true even for atomic compound statements. If an error occurs within an atomic compound statement and is explicitly handled, some but not all of the statements in the atomic compound statement are executed.

Nested compound statements and exception handlers

The code following a statement that causes an error is not executed unless an ON EXCEPTION RESUME clause is included in a procedure definition.

You can use nested compound statements to give you more control over which statements are and are not executed following an error.

The following demonstration procedure illustrates how nested compound statements can be used to control flow. The procedure is based on that used as an example in "Default error handling in procedures and triggers" on page 300.

```
CREATE PROCEDURE InnerProc()
BEGIN
   DECLARE column_not_found
      EXCEPTION FOR SQLSTATE VALUE '52003';
      MESSAGE 'Hello from InnerProc';
      SIGNAL column_not_found;
         MESSAGE 'Line following SIGNAL'
   EXCEPTION
      WHEN column_not_found THEN
         MESSAGE 'Column not found handling';
      WHEN OTHERS THEN
         RESIGNAL;
      MESSAGE 'Outer compound statement';
END
```

The following statement executes the InnerProc procedure:

```
CALL InnerProc();
```

The message window of the database engine then displays the following:

Hello from InnerProc

Column not found handling

Outer compound statement

When the SIGNAL statement that causes the error is encountered, control passes to the exception handler for the compound statement, and the Column not found handling message is printed. Control then passes back to the outer compound statement and the Outer compound statement message is printed.

If an error other than column not found is encountered in the inner compound statement, the exception handler executes the RESIGNAL statement. The RESIGNAL statement passes control directly back to the calling environment, and the remainder of the outer compound statement is not executed.

Using the EXECUTE IMMEDIATE statement in procedures

The EXECUTE IMMEDIATE statement allows statements to be built up inside procedures using a combination of literal strings (in quotes) and variables.

For example, the following procedure includes an EXECUTE IMMEDIATE statement that creates a table.

```
CREATE PROCEDURE CreateTableProc(
      IN tablename char(30) )
BEGIN
   EXECUTE IMMEDIATE 'CREATE TABLE ' || tablename
||' ( column1 INT PRIMARY KEY)'
END
```

In ATOMIC compound statements, you cannot use an EXECUTE IMMEDIATE statement that causes a COMMIT, as COMMITs are not allowed in that context.

Transactions and savepoints in procedures and triggers

SQL statements in a procedure or trigger are part of the current transaction (see the chapter "Using Transactions and Locks"). You can call several procedures within one transaction or have several transactions in one procedure.

COMMIT and ROLLBACK are not allowed within any atomic statement (see "Atomic compound statements" on page 285). Note that triggers are fired due to an INSERT, UPDATE, or DELETE which are atomic statements. COMMIT and ROLLBACK are not allowed in a trigger or in any procedures called by a trigger.

Savepoints (see "Savepoints within transactions" in the chapter "Using Transactions and Locks") can be used within a procedure or trigger, but a ROLLBACK TO SAVEPOINT statement can never refer to a savepoint before the atomic operation started. Also, all savepoints within an atomic operation are released when the atomic operation completes.

Some hints for writing procedures

This section provides some pointers for developing procedures.

Check if you need to change the command delimiter

You do not need to change the command delimiter in ISQL or Sybase Central when you are writing procedures. However, if you are creating and testing procedures and triggers from some other browsing tool, you may need to change the command delimiter from the semicolon to another character. Each statement within the procedure ends with a semicolon. For some browsing applications to parse the CREATE PROCEDURE statement itself, you need the command delimiter to be something other than a semicolon. If you are using an application that requires changing the command delimiter, a good choice is to use two semicolons as the command delimiter (;;) or a question mark (*?*) if the system does not permit a multicharacter delimiter.

Remember to delimit statements within your procedure

You should terminate each statement within the procedure with a semicolon. Although you can leave off semicolons for the last statement in a statement list, it is good practice to use semicolons after each statement.

The CREATE PROCEDURE statement itself contains both the RESULT specification and the compound statement that forms its body. No semicolon is needed after the BEGIN or END keywords, or after the RESULT clause.

Use fully-qualified names for tables in procedures

If a procedure has references to tables in it, you should always preface the table name with the name of the owner (creator) of the table.

When a procedure refers to a table, it uses the group memberships of the procedure creator to locate tables with no explicit owner name specified. For example, if a procedure created by user_1 references Table_B and does not specify the owner of Table_B, then either Table_B must have been created by user_1 or user_1 must be a member of a group (directly or indirectly) that is the owner of Table_B. If neither condition is met, a table not found message results when the procedure is called.

You can minimize the inconvenience of long fully qualified names by using a correlation name to provide a convenient name to use for the table within a statement. Correlation names are described in "FROM clause" in the chapter "Watcom-SQL Statements".

Specifying dates and times in procedures

When dates and times are sent to the database from procedures, they are sent as strings. The string is interpreted according to the current setting of the DATE_ORDER database option. As different connections may set this option to different values, some strings may be converted incorrectly to dates, or the database may not be able to convert the string to a date.

You should use the unambiguous date format *yyyy-mm-dd* or *yyyy/mm/dd* when sending dates to the database from procedures. These strings are interpreted unambiguously as dates by the database, regardless of the DATE_ORDER database option setting.

For more information on dates and times, see "Date and time data types" in the chapter "SQL Anywhere Data Types".

Verifying that procedure input arguments are passed correctly

You can verify that input arguments to a procedure are passed correctly in several ways. For example:

- Display the value of the parameter on the message window of the database engine or server using the MESSAGE statement. For example, the following procedure simply displays the value of the input parameter *var*:

```
CREATE PROCEDURE message_test (IN var char(40))
BEGIN
   MESSAGE var;
END
```

- From ISQL:

  1 Create a variable to hold the argument (say GBLVAR).

  2 Assign the argument to GBLVAR inside the procedure:

```
SET GBLVAR = argument
```

  3 After calling the procedure, display the value of GBLVAR:

```
SELECT GBLVAR
```

4 Assign the input value to an output parameter, and test the value of the output parameter on completion.

5 Insert the argument into a table, and SELECT from the table.

Statements allowed in batches

The following statements are not allowed in batches:

- CONNECT or DISCONNECT statement
- ALTER PROCEDURE or ALTER FUNCTION statement
- CREATE TRIGGER statement
- ISQL commands such as INPUT or OUTPUT

Otherwise, any SQL statement is allowed, including data definition statements such as CREATE TABLE, ALTER TABLE, and so on.

The CREATE PROCEDURE statement is allowed, but must be the final statement of the batch.

Using SELECT statements in batches

You can include one or more SELECT statements in a batch. There are limitations on the use of more than one SELECT statements, just as in stored procedures: multiple SELECT statements are allowed only if they return the same result columns.

The following is a valid batch:

```
IF EXISTS(  SELECT *
            FROM systable
            WHERE table_name='employee' )
THEN
   SELECT   emp_lname AS LastName,
            emp_fname AS FirstName
   FROM employee;
   SELECT lname, fname
   FROM customer;
   SELECT last_name, first_name
   FROM contact;
END IF
```

The alias for the result set is required only in the first SELECT statement, as the database engine uses the first SELECT statement in the batch to describe the result set.

A RESUME is required following each query to retrieve the next result set.

The following is not a valid batch, as the two queries return different result sets:

```
IF EXISTS( SELECT * FROM systable
      WHERE table_name='employee' )
```

```
THEN
   SELECT   emp_lname AS LastName,
            emp_fname AS FirstName
   FROM employee;
   SELECT id, lname, fname
   FROM customer;
END IF
```

Calling external libraries from stored procedures

You can call a function in an external dynamic link library (DLL) from a stored procedure (including user-defined functions) under operating systems that support DLLs. You can also call functions in an NLM under NetWare. SQL Anywhere includes a set of system procedures that make use of this capability to send MAPI e-mail messages and carry out other functions. This section describes how to use the external library calls in procedures.

> ***Caution: external libraries can corrupt your database***
> *External libraries called from procedures share the memory of the database engine. If you call a DLL from a procedure and the DLL contains memory-handling errors, you can crash the database engine or corrupt your database. Ensure your libraries are thoroughly tested before deploying them on production databases.*

For information on MAPI and other system procedures, see the chapter "SQL Anywhere System Procedures and Functions".

Creating procedures and functions with external calls

This section presents some examples of procedures and functions with external calls..

For a full description of CREATE PROCEDURE statement syntax, see "CREATE PROCEDURE statement" in the chapter "Watcom-SQL Statements".

For a full description of the CREATE FUNCTION statement syntax for external calls, see "CREATE FUNCTION statement" in the chapter "Watcom-SQL Statements".

A procedure that calls a function function_name in DLL library.dll can be created as follows:

```
CREATE PROCEDURE dll_proc ( parameter-list )
EXTERNAL NAME 'function_name@library.dll'
```

Such a procedure is called an **external stored procedure**. If you call an external DLL from a procedure, the procedure cannot carry out any other tasks; it just forms a wrapper around the DLL.

An analogous CREATE FUNCTION statement is as follows:

```
CREATE FUNCTION dll_func ( parameter-list )
RETURNS data-type
EXTERNAL NAME 'function_name@library.dll'
```

In these statements, function_name is the name of a function in the dynamic link library, and library.dll is the name of the library. The arguments in the procedure argument list must correspond in type and order to the arguments to the library function; they are passed to the external DLL function in the order in which they are listed. Any value returned by the external function is in turn returned by the procedure to the calling environment.

A procedure that calls an external function can include no other statements: its sole purposes are to take arguments for a function, call the function, and return any value and returned arguments from the function to the calling environment. You can use IN, INOUT, or OUT parameters in the procedure call in the same way as for other procedures: the input values get passed to the external function, and any parameters modified by the function are returned to the calling environment in OUT or INOUT parameters.

You can specify operating-system dependent calls, so that a procedure calls one function when run on one operating system, and another function (presumably analogous) on another operating system. The syntax for such calls is to prefix the function name with the operating system name. For example:

```
CREATE PROCEDURE dll_proc ( parameter-list )
EXTERNAL NAME
'OS2:os2_fn@os2_lib.dll;WindowsNT:nt_fn@nt_lib.dll'
```

The operating system identifier must be one of OS2, WindowsNT, Windows95, Windows3X, or NetWare.

If no system identifier for the current operating system is provided, and a function with no system identifier is provided, that function is called.

NetWare calls have a slightly different format than the other operating systems. All symbols are globally known under NetWare, so that if a symbol (such as a function name) is exported, it must be unique to all NLMs on the system. Consequently, the NLM name is not necessary in the call, and it has the following syntax:

```
CREATE PROCEDURE dll_proc ( parameter-list )
EXTERNAL NAME 'NetWare:nw_fn'
```

No library name needs to be provided.

External function declarations

When an external function is called, a stack is fabricated with the arguments (or argument references in the case of INOUT or OUT parameters) and the DLL is called. Only the following data types can be passed to an external library:

- CHARACTER data types, but INOUT and OUT parameters must be no more than 255 bytes in length
- SMALLINT and INT data types
- REAL and DOUBLE data types

This section describes the format of the function declaration.

For information about passing parameters to external functions, see "How parameters are passed to the external function", next.

In the external library, function declarations should follow the following guidelines:

- **Windows NT and Windows 95** The function declaration should be of the following form for the Watcom C/C++ compiler:

  ```
  return-type * __export __stdcall function-name(
  argument-list )
  ```

  and the following form for Microsoft Visual C++:

  ```
  return-type * __declspec(dllexport) function-
  name( argument-list )
  ```

- **Windows 3.x** All pointers are far pointers, so the DLL must be at least compiled under the large model. The function declaration should be of the following form for the Watcom C/C++ compiler:

  ```
  return-type * __far __export __pascal function-
  name( argument-list )
  ```

For the 32-bit Windows 3.x engine or server, no more than 256 parameters can be used, of any type.

- **OS/2** The function declaration should be of the following form for the Watcom C/C++ compiler:

  ```
  return-type * __export __syscall function-name(
  argument-list )
  ```

- **NetWare** The function declaration should be of the following form for the Watcom C/C++ compiler:

  ```
  return-type  *
  ```

How parameters are passed to the external function

SQL data types are mapped to their C equivalents as follows:

SQL data type	C data type
INTEGER	long
SMALLINT	short
REAL	float
DOUBLE	double
CHAR(n) or VARCHAR (n)	char *

These are the only SQL data types you can use: using others produces an error.

Procedure parameters that are INOUT or OUT parameters are passed to the external function by reference. For example, the procedure

```
CREATE PROCEDURE dll_proc( INOUT xvar REAL )
EXTERNAL NAME 'function_name@library.dll'
```

has an associated C function parameter declaration of

```
function_name( float * xvar )
```

Procedure parameters that are IN parameters are passed to the external function by value. For example, the procedure

```
CREATE PROCEDURE dll_proc( IN xvar REAL )
EXTERNAL NAME 'function_name@library.dll'
```

has an associated external function parameter declaration of

```
function_name( float xvar )
```

Character data types are an exception to IN parameters being passed. They are always passed by reference, whether they are IN, OUT, or INOUT parameters. For example, the procedure

```
CREATE PROCEDURE dll_proc ( IN invar CHAR( 128 ) )
EXTERNAL NAME 'function_name@library.dll'
```

has the following external function parameter declaration

```
function_name( char * invar )
```

Special considerations when passing character types

For character data types (CHAR or VARCHAR), the database engine allocates a 255-byte buffer (including one for the null terminator) for each parameter. If the parameter is an INOUT parameter, the existing value is copied into the buffer and null terminated, and a pointer to this buffer is passed to the external function. The external function should therefore not allocate a buffer of its own for OUT or INOUT character parameters: the engine has already allocated the space. If the external function writes beyond the 255 bytes (including the ending null character), it is writing over data structures in the database engine.

When the entry point returns, the parameter buffers are translated back into their engine data structure string equivalents based on the strlen() value of the buffer.

The external function should be sure to null-terminate any output string parameters. OUT parameters follow the same procedure except that as there is no initial data, no initial value of the output buffer parameter is guaranteed.

CHAPTER 21

Monitoring and Improving Performance

About this chapter

This chapter describes some of the methods available in SQL Anywhere to monitor and improve the performance of your database.

Contents

Topic	Page
Factors affecting database performance	320
Using keys to improve query performance	322
Using indexes to improve query performance	326
Search strategies for queries from more than one table	328
Sorting query results	331
Temporary tables used in query processing	332
How the optimizer works	333
Monitoring database performance	336

Factors affecting database performance

There are many factors that can affect database performance. This chapter describes ways of improving performance from a SQL perspective, and assumes a reasonably designed database running with sufficient resources. Disk and file management, as well as hardware configuration, also play an important part in determining performance. For example, the following can lead to poor performance:

Factors leading to poor database performance

- You are not running with a transaction log (see "The transaction log" in the chapter "Backup and Data Recovery"). A transaction log improves commit time for transactions that insert, update, or delete rows.
- You are using the 16-bit version of the Windows 3.x database engine. The 32-bit version has better performance, especially for larger databases.
- You are loading huge amounts of data into a database. See "Tuning bulk operations" in the chapter "Importing and Exporting Data" for methods to improve performance.
- The database engine does not have an adequate amount of memory for caching database pages. See "The database engine" in the chapter "SQL Anywhere Components" for command line options for controlling the cache size. Extra memory for your computer could improve database performance dramatically.
- Your hard disk is excessively fragmented. This becomes more important as your database increases in size. The DOS and Windows database engines cannot do direct (fast) reading and writing when the database file is very fragmented. There are several utilities available for DOS and Windows to defragment your hard disk. One of these should be run periodically. You could put the database on a DOS disk partition by itself to eliminate fragmentation problems.
- You are using a small page size for a large database. The page size is determined when the database is created by the initialization utility. You can find out the page size by using DBINFO. To change page size, you need to unload and reload your database.
- The database table design is not good. A bad database design can result in time-consuming queries to get information from the database. If indexes will not solve your performance problem, consider alternative database designs.

- Your hard disk is slow. A faster hard disk, a caching disk controller or a disk array can improve performance considerably.
- In a multiuser environment, your network performance is slow.
- You are fetching or inserting many rows of data. Consider using the multi-row operations.

  For more information, see "FETCH statement" in the chapter "Watcom-SQL Statements" and "PUT statement" in the chapter "Watcom-SQL Statements".

Using keys to improve query performance

The foreign key and the primary key are used for validation purposes. However, these keys are also used by SQL Anywhere to improve performance where possible.

Example

The following example illustrates how keys are used to make commands execute faster.

```
SELECT *
FROM employee
WHERE emp_id = 390
```

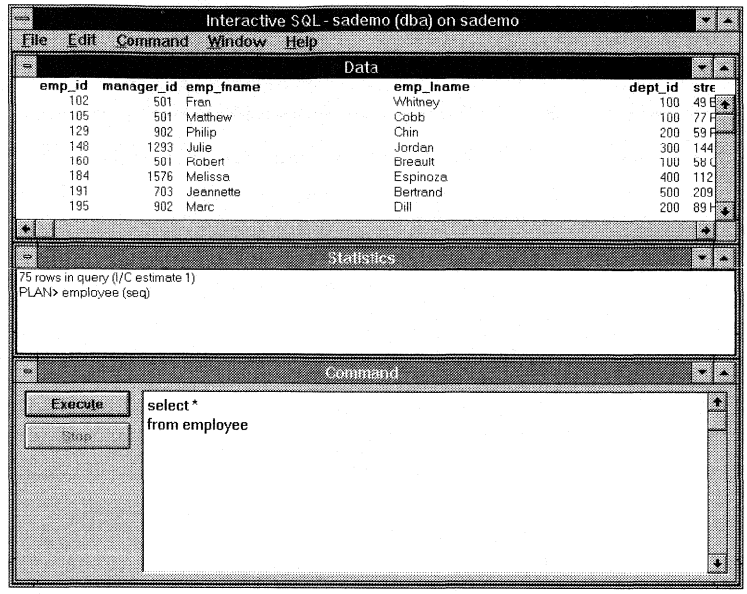
The simplest way for SQL Anywhere to perform this query would be to look at all 75 rows in the employee table and check the employee ID number in each row to see if it is 390. This does not take very long since there are only 75 employees, but for tables with many thousands of entries the search can take a long time.

The emp_id column is the primary key for the employee table. SQL Anywhere has a built-in index mechanism for finding primary and foreign key values quickly. (This mechanism is used for the validation you saw in "Validity checking" in the chapter "Updating the Database".)

SQL Anywhere automatically uses the same mechanism to find the employee number 390 quickly. This quick search takes almost the same amount of time whether there are 100 rows or 1,000,000 rows in the table.

Using ISQL to examine query performance

The ISQL statistics window tells you when keys are being used to improve performance.



The number of lines in the statistics window can be adjusted or the window can be turned off altogether.

Changing the size of the statistics window

Under Windows, Windows NT and OS/2, the window can be resized in the standard manner. Click the mouse on the window border and drag it to the desired size. The window can be turned off by minimizing it. Restore the window by choosing Window➤Statistics from the menu bar.

Under DOS and QNX, you can control the size of the statistics window with the SET OPTION STATISTICS command. The statistics window can also be turned on or off from the configuration window (choose Options➤Configure from the menu bar). The Statistics option contains the number of lines in the statistics window. Use 0 to turn off the statistics window.

Information in the statistics window

If you execute a query to look at every row in the employee table:

```
SELECT *
FROM employee
```

two lines appear in the Statistics window:

75 rows in query (I/O estimate 1)

PLAN> employee (seq)

The first line indicates the number of rows in the query. Sometimes the database knows exactly, as in this case where there are 75 rows; other times it estimates the number of rows. The first line also indicates an internal I/O estimate of how many times SQL Anywhere will have to look at the database on your hard disk to examine the entire employee table.

The second line summarizes the execution plan for the query: the tables that are searched, any indexes used to search through a table. This plan says that SQL Anywhere will look at the employee table sequentially (that is, one page at a time, in the order that the rows appear on the pages). The letters seq inside parentheses mean that SQL Anywhere will examine all the rows of the table. This makes sense since the query fetches the entire table.

Resetting statistics

You may notice that when working through the tutorial yourself that the statistics window contains estimates than are different from what is given here. This may happen because the SQL Anywhere optimizer has decided to optimize a query differently. The optimizer maintains statistics as it evaluates queries and uses these statistics to optimize subsequent queries. These statistics can be reset by executing the following statement:

```
DROP OPTIMIZER STATISTICS
```

Note that you must have DBA authority to execute this statement.

Using primary keys to improve query performance

SQL Anywhere uses a primary key to improve performance on the following statement:

```
SELECT *
FROM employee
WHERE emp_id = 390
```

Statistic window information

The statistics window contains the following two lines:

```
Estimated 1 rows in query (I/O estimate 13)

PLAN> employee (employee)
```

Whenever the name inside the parentheses in the ISQL statistics window PLAN description is the same as the name of the table, it means that SQL Anywhere will use the primary key for the table to improve performance. Also, the ISQL statistics window shows that the database optimizer estimates that there will be one row in the query and that it will have to go to the disk 13 times.

Using foreign keys to improve query performance

The following query lists the orders from customer with customer ID 113:

```
SELECT *
FROM sales_order
WHERE cust_id = 113
```

Statistic window information

The statistics window contains the following information:

Estimated 5 rows in query (I/O estimate 10)

PLAN> sales_order (ky_so_customer)

Here ky_so_customer refers to the foreign key that the sales_order table has for the customer table.

Primary and foreign keys are just special indexes that also maintain entity and referential integrity. The integrity is maintained by extra information that is placed in the indexes.

Using indexes to improve query performance

Sometimes you need to search for something which is not in a primary or foreign key. Hence, SQL Anywhere cannot use a key to improve performance. Creating indexes speeds up searches on particular columns. For example, suppose you wanted to look up all the employees with a last name beginning with M.

A query for this is as follows:

```
SELECT *
FROM employee
WHERE emp_lname LIKE 'M%'
```

If you execute this command, the plan description in the ISQL statistics window shows that the table is searched sequentially.

Creating an index

If a search by employee last names is common, you may wish to create an index on the emp_lname column in order to speed up the queries. You can do this with a CREATE INDEX statement.

```
CREATE INDEX lname
ON employee ( emp_lname )
```

The column name emp_lname indicates the column that is indexed. An index can contain one, two, or more columns. However, if you create a multiple-column index, and then do a search with a condition using only the second column in the index, the index cannot be used to speed up the search. An index is similar to a telephone book which first sorts people by their last name, and then all the people with the same last name by their first name. A telephone book is useful if you know the last name, even more useful if you know both the first name and last name, but worthless if you only know the first name and not the last name.

Once you have created the index, rerunning the query produces the following plan description in the Statistics window:

```
PLAN> employee (lname)
```

How AQL Anywhere uses an index

SQL Anywhere uses the index automatically. Once an index is created, SQL Anywhere automatically keeps it up to date and uses it to improve performance whenever it can.

You could create an index for every column of every table in the database. But that would make data modifications slow, since SQL Anywhere would have to update all indexes affected by the change. Further, each index requires space in the database. For these reasons, you should only create indexes that are used frequently.

Since you will not be using this index again, you should delete it by entering the following statement:

```
DROP INDEX lname
```

How indexes work

This section provides a technical description of how the SQL Anywhere database engine uses indexes when searching databases.

Index page structure

The SQL Anywhere engine uses modified B+ trees. Each index page is a node in the tree and each node has many index entries. Leaf page index entries have a reference to a row of the indexed table. Indexes are kept balanced (uniform depth) and pages are kept close to full.

An index lookup

An index lookup starts with the root page. The index entries on a nonleaf page determine which child page has the correct range of values. The index lookup moves down to the appropriate child page. This continues until a leaf page is reached. An index with N levels will require N reads for index pages and 1 read for the data page that contains the actual row. Index pages tend to be cached due to the frequency of use.

Recommended page sizes

About the first 10 bytes of data for each index entry are stored in the index pages. This allows for a fan-out of roughly 200 using 4K pages, meaning that 200 rows can be indexed on one page, and 40,000 rows can be indexed with a two-level index. Each new level of an index allows for a table 200 times larger. Page size can significantly affect fan-out, in turn affecting the depth of index required for a table. 4K pages are recommended for large databases.

The leaf nodes of the index are linked together. Once a row has been looked up, the rows of the table can be scanned in index order. Scanning all rows with a given value requires only one index lookup, followed by scanning the leaf nodes of the index until the value changes. This occurs when you have a WHERE clause that filters out rows with a certain value or a range of values. It also occurs when joining rows in a one-to-many relationship.

Search strategies for queries from more than one table

This section uses sample queries to illustrate how SQL Anywhere selects an optimal processing route for each query. If you execute each of the commands in this section in ISQL, the statistics window display shows you the execution plan chosen by SQL Anywhere to process each query.

Using a key join

The following simple query uses a **key join** to search more than one table:

```
SELECT customer.company_name, sales_order.id
FROM sales_order
KEY JOIN customer
```

The Statistics window displays the following:

```
Estimated 711 rows in query (I/O estimate 2)

PLAN> customer(seq), sales_order(ky_so_customer)
```

When this query is executed, the ISQL statistics window display indicates that SQL Anywhere first examines each row in the customer table, then finds the corresponding sales order number in the sales_order table using the ky_so_customer foreign key joining the sales_order and customer tables.

The order that the tables are listed in the statistics window is the order that the tables are accessed by the database.

Adding a WHERE clause

If you modify the query by adding a WHERE clause, as follows, SQL Anywhere carries out the search in a different order:

```
SELECT customer.company_name, sales_order.id
FROM sales_order
KEY JOIN customer
WHERE sales_order.id = 2583
```

The Statistics windows displays the following plan:

```
PLAN> sales_order(sales_order), customer(customer)
```

Now, SQL Anywhere looks in the sales_order table first, using the primary key index. Then, for each sales order numbered 2583 (there is only one), it looks up the company_name in the customer table using the customer table primary key to identify the row. The primary key can be used here because the row in the sales_order table is linked to the rows of the customer table by the customer id number, which is the primary key of the customer table.

The tables are examined in a different order depending on the query. The SQL Anywhere built-in query optimizer estimates the cost of different possible execution plans, and chooses the plan with the least estimated cost.

For some more complicated examples, try the following commands which each join four tables. The ISQL statistics window shows that SQL Anywhere processes each query in a different order.

Example 1

❖ **To list the customers and the sales reps they have dealt with.**

♦ Type the following:

```
SELECT  customer.lname,
   employee.emp_lname

FROM customer
   KEY JOIN sales_order
   KEY JOIN sales_order_items
   KEY JOIN employee
```

lname	emp_lname
Colburn	Chin
Smith	Chin
Sinnot	Chin
Piper	Chin
Phipps	Chin

The plan for this query is as follows:

PLAN> employee (seq), sales_order (ky_so_employee_id),

customer (customer), sales_order_items (id_fk)

Example 2

The following command restricts the results to list all sales reps that the customer named Piper has dealt with:

```
SELECT customer.lname,
employee.emp_lname
FROM customer
   KEY JOIN sales_order
   KEY JOIN sales_order_items
   KEY JOIN employee
   WHERE   customer.lname = 'Piper'
```

The plan for this query is as follows:

PLAN> customer (ix_cust_name), sales_order (ky_so_customer),

employee (employee), sales_order_items (id_fk)

Example 3

The third example shows all customers who have dealt with sales reps of the same name:

```
SELECT  customer.lname,
    employee.emp_lname
FROM customer
    KEY JOIN sales_order
    KEY JOIN sales_order_items
    KEY JOIN employee
WHERE  customer.lname = employee.emp_lname
```

The plan for this query is as follows:

```
PLAN> employee (seq), customer (ix_cust_name),
sales_order (ky_so_employee_id), sales_order_items (id_fk)
```

For information on how SQL Anywhere's optimizer selects a strategy for each search, see "How the optimizer works" on page 333.

Sorting query results

Many queries against a database have an ORDER BY clause so that the rows come out in a predictable order. SQL Anywhere uses indexes to accomplish the ordering quickly. For example,

```
SELECT *
FROM customer
ORDER BY customer.lname
```

can use the index on the lname column of the customer table to access the rows of the customer table in alphabetical order by last name.

Queries with WHERE and ORDER BY clauses

A potential problem arises when a query has both a WHERE clause and an ORDER BY clause.

```
SELECT *
FROM customer
WHERE id > 300
ORDER BY company_name
```

The database engine must decide between two strategies:

1 Go through the entire customer table in order by company name, checking each row to see if the customer id is greater than 300.

2 Use the key on the id column to read only the companies with id greater than 300. The results would then need to be sorted by company name.

If there are very few id values greater than 300, the second strategy is better because only a few rows are scanned and quickly sorted. If most of the id values are greater than 300, the first strategy is much better because no sorting is necessary.

Solving the problem

The example above could be solved by creating a two-column index on id and company_name. (The order of the two columns is important.) The database engine could then use this index to select rows from the table and have them in the correct order. However, keep in mind that indexes take up space in the database file and involve some overhead to keep up to date. Do not create indexes indiscriminately.

Temporary tables used in query processing

Sometimes SQL Anywhere needs to make a **temporary table** for a query. This occurs in the following cases:

When temporary tables occur

- When a query has an ORDER BY or a GROUP BY and SQL Anywhere does not use an index for sorting the rows. Either no index was found or the optimizer chose a strategy that did not use the appropriate index for sorting.
- When a multiple-row update is being performed and the column being updated is used in the where clause of the update or in an index that is being used for the update.
- When a multiple-row update or delete has a subquery in the where clause that references the table being modified.
- When an insert from a select statement is being performed and the select statement references the insert table.
- When a multiple row INSERT, UPDATE, or DELETE is performed, and there are triggers defined on the table that the operation causes to fire.

In these cases, SQL Anywhere makes a temporary table before the operation begins. The records affected by the operation are put into the temporary table and a temporary index is built on the temporary table. This operation of extracting the required records into a temporary table can take a significant amount of time before any rows at all are retrieved from the query. Thus, creating indexes that can be used to do the sorting in case 1 above will improve the performance of these queries since it will not be necessary to build a temporary table.

Notes

The insert, update and delete cases above are usually not a performance problem since it is usually a one-time operation. However, if it does cause problems, the only thing that can be done to avoid building a temporary table is to rephrase the command to avoid the conflict. This is not always possible.

In ISQL, the statistics window displays "TEMPORARY TABLE" before the optimization strategy is listed if a temporary table is created by SQL Anywhere in carrying out the search.

How the optimizer works

The database engine has an optimizer that attempts to pick the best strategy for executing each query. The best strategy is the one that gets the results in the shortest period of time. The optimizer determines the **cost** of each strategy by estimating the number of disk reads and writes required. The strategy with the lowest cost is chosen.

The optimizer must decide which order to access the tables in a query, and whether or not to use an index for each table. If a query joins N tables, there are N factorial possible ways to access the tables. The optimizer will estimate the cost of executing the query in the different ways and use the ordering with the lowest cost estimate. The query execution plan in the ISQL statistics window shows the table ordering for the current query and indicates in parentheses the index that was used for each table.

Optimizer estimates

The optimizer uses heuristics (educated guesses) to help decide the best strategy.

For each table in a potential execution plan, the optimizer must estimate the number of rows that will be part of the results. The number of rows will depend on the size of the table and the restrictions in the WHERE clause or the ON clause of the query.

Percentage of rows that some comparison operations select

The table below shows the simplest guess at the percentage of rows that some of the comparison operations will select. The other comparison operations such as LIKE, IS NULL, and EXISTS are handled in a similar way.

Comparison	Percentage
=	5 (see below)
<>	95
<, <=, >, >=	25
between	6

In many cases, the optimizer uses more sophisticated heuristics. For example, the estimate of 5percent for equality is only used in cases where these other heuristics do not apply.

The optimizer makes use of indexes and keys to improve its guess of the number of rows. Here are a few single-column examples:

Single-column examples

- Equating a column to a value: estimate one row when the column has a unique index or is the primary key.
- A comparison of an indexed column to a constant: use the index to estimate the percentage of rows that will satisfy the comparison.
- Equating a foreign key to a primary key (key join): use relative table sizes in determining an estimate. For example, if a 5000 row table has a foreign key to a 1000 row table, the optimizer guesses that there are five foreign rows for each primary row.

Self tuning of the query optimizer

One of the most common constraints in a query is equality with a column value. For example,

```
SELECT *
FROM employee
WHERE sex = 'f'
```

tests for equality of the sex column. For this type of constraint, the SQL Anywhere optimizer learns from its experience. A query will not always be optimized the same way the second time it is executed. The estimate for an equality constraint will be modified for columns that have an unusual distribution of values. This information is stored permanently in the database. If needed, the statistics can be deleted with the DROP OPTIMIZER STATISTICS command.

Providing estimates to improve query performance

Since the query optimizer is guessing at the number of rows in a result based on the size of tables and particular restrictions used in the WHERE clause, it almost always makes inexact guesses. In many cases, the guess that the query optimizer makes is close enough to the real number of rows that the optimizer will have chosen the best search strategy. However, in some cases this does not occur.

The following query displays a list of order items that shipped later than the end of June, 1994:

```
SELECT  ship_date
FROM  sales_order_items
WHERE ship_date > '1994/06/30'
```

```
ORDER BY ship_date DESC
```

The estimated number of rows is 274. However, the actual number of rows returned is only 12. This estimate is wrong because the query optimizer guesses that a test for greater than will succeed 25 percent of the time. In this example, the condition on the ship_date column:

```
ship_date > '1994/06/30'
```

is assumed to choose 25 percent of rows in the sales_order_items table.

Using an estimate

If you know that a condition has a success rate that differs from the optimizer rule, you can tell the database this information by using an estimate. An estimate is formed by enclosing in brackets the expression followed by a comma and a number. The number represents the percentage of rows that the expression is estimated to select. In this case, you could estimate a success rate of one percent:

```
SELECT  ship_date
FROM  sales_order_items
WHERE ( ship_date > '1994/06/30', 1 )
ORDER BY ship_date DESC
```

With this estimate, the optimizer estimates ten rows in the query.

Note

Incorrect estimates are only a problem when they lead to poorly optimized queries.

Monitoring database performance

SQL Anywhere provides a set of statistics that can be used to monitor database performance. These are accessible from Sybase Central, and client applications can access the statistics as functions. In addition, these statistics are made available by the database engine to the Windows NT performance monitor. The NT performance monitor can be used only to monitor Windows NT database engines or network servers.

This section describes how to access performance and related statistics from client applications, how to monitor database performance using Sybase Central, and how to monitor database performance using the Windows NT performance monitor.

Obtaining database statistics from a client application

SQL Anywhere provides a set of system functions that can access information on a per-connection, per-database, or engine-wide basis. The kind of information available ranges from static information such as the server name, the database file associated with a database, or the page-size of a database, to detailed performance-related statistics concerning disk and memory usage. The performance-related statistics are also available, along with some other statistics, for the Windows NT engine and server in the Windows NT Performance Monitor.

For more information on the Performance Monitor, see "Monitoring database statistics from the NT Performance Monitor" on page 338.

This section illustrates how to use the functions.

A complete list of system functions and of the properties available using the system functions is provided in the section "System functions" in the chapter "Watcom-SQL Functions".

Functions that retrieve system information

The following functions are used to retrieve system information:

property Provides the value of a given property on an engine-wide basis.

connection_property Provides the value of a given property for a given connection, or for the current connection by default.

db_property Provides the value of a given property for a given database, or for the current database by default. If you supply as argument only the name of the property you wish to retrieve, the functions return the value for the current server, connection, or database.

Examples

For example:

- The following statement sets a variable named server_name to the name of the current server:

```
SET server_name = property( 'name' )
```

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

```
SELECT connection_property( 'userid' )
```

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

```
SELECT db_property( 'file' )
```

Improving query efficiency

For maximum efficiency, a client application monitoring database activity should use the property_number function to identify a named property, and then use the number to repeatedly retrieve the statistic. The following set of statements illustrates the process from ISQL:

```
CREATE VARIABLE propnum INT ;
CREATE VARIABLE propval INT ;
SET propnum = property_number( 'cacheread' );
SET propval = property( propnum )
```

Property names obtained in this way are available for many different database statistics, from the number of transaction log page write operations and the number of checkpoints carried out to the number of reads of index leaf pages from the memory cache.

Many of these statistics are made available in graphical form from the Sybase Central database management tool.

Monitoring database statistics from Sybase Central

You can monitor database statistics from Sybase Central. The Sybase Central Performance Monitor is a graphing tool allowing database statistics to be plotted as a line graph or a bar graph.

❖ **To start the Sybase Central Performance Monitor:**

1. Click the icon for the server you wish to monitor in the left panel.
2. Double-click the Statistics folder underneath the server.
3. Select a statistic to graph, and drag it to the Performance Monitor icon to start graphing that statistic.

About the performance monitor

The Performance Monitor uses the regular SQL Anywhere communication mechanisms to gather statistics. This means some statistics (most notably Cache Reads) are affected by Sybase Central. For example, graphing Cache Reads/sec in Sybase Central shows a steady rate, even when nothing apart from the monitoring is going on.

If you have an NT client and server, the NT Performance monitor is preferable since it offers more statistics, and is not intrusive: updating the statistics will not affect the measurements. The extra statistics the NT performance monitor offers deal mainly with network communications—packets received, network buffers used, and so on.

Monitoring database statistics from the NT Performance Monitor

The NT performance monitor is an application for viewing the behavior of objects such as processors, memory, and applications. SQL Anywhere provides many statistics for the performance monitor to display.

The NT performance monitor allows unintrusive monitoring of statistics: updating the statistics does not affect the measurements.

❖ **To start the NT performance monitor:**

1. Open the Administrative Tools program group.
2. Double click Performance Monitor.

For information about Performance Monitor, see the Performance Monitor online Help.

❖ **To display SQL Anywhere statistics:**

1. With Performance Monitor running, select Add To Chart from the Edit menu, or click the Plus sign on the toolbar.

   The Add To Chart dialog appears.
2. From the Object list, select SQL Anywhere.

   The Counter list then displays a list of the statistics provided.
3. From the Counter list, click a statistic to be displayed.
4. For a description of the selected counter, click Explain.
5. To display the counter, click Add.

6 When you have selected all the counters you wish to display, click Done.

Performance Monitor statistics

The statistics made available for Performance Monitor by SQL Anywhere are as follows:

Statistic	Description
Active Requests	Active Requests is the number of engine threads that are currently handling a request.
Asynchronous Reads/sec	Asynchronous Reads/sec is the rate at which pages are being read asynchronously from disk.
Asynchronous Writes/sec	Asynchronous Writes/sec is the rate at which pages are being written asynchronously to disk.
Bytes Received/sec	Bytes Received/sec is the rate at which network data (in bytes) are being received.
Bytes Transmitted/sec	Bytes Transmitted/sec is the rate at which bytes are being transmitted over the network.
Cache Hits/sec	Cache Hits/sec is the rate at which database page lookups are satisfied by finding the page in the cache.
Cache Index Internal Reads/sec	Cache Index Internal Reads/sec is the rate at which index internal-node pages are being read from the cache.
Cache Index Leaf Reads/sec	Cache Index Leaf Reads/sec is the rate at which index leaf pages are being read from the cache.
Cache Reads/sec	Cache Reads/sec is the rate at which database pages are being looked up in the cache.
Cache Table Reads/sec	Cache Table Reads/sec is the rate at which table pages are being read from the cache.
Cache Writes/sec	Cache Writes/sec is the rate at which pages in the cache are being modified (in pages/sec).
Checkpoint Flushes/sec	Checkpoint Flushes/sec is the rate at which ranges of adjacent pages are being written out during a checkpoint.
Checkpoint Log/sec	Checkpoint Log/sec is the rate at which the transaction log is being checkpointed.
Checkpoint Urgency	Checkpoint Urgency is expressed as a percentage. See documentation for details.
Checkpoints/sec	Checkpoints/sec is the rate at which checkpoints are being performed.
Commit files/sec	Commit files/sec is the rate at which the engine is forcing a flush of the disk cache. On NT and NetWare platforms, the

Statistic	Description
	disk cache does not need to be flushed since unbuffered (direct) IO is used.
Commits/sec	Commits/sec is the rate at which Commit requests are being handled.
Context Switch Checks/sec	Context Switch Checks/sec is the rate at which the current engine thread is volunteering to give up the CPU to another engine thread.
Context Switches/sec	Context Switches/sec is the rate at which the current engine thread is being changed.
Continue Requests/sec	Continue Requests/sec is the rate at which "CONTINUE" requests are being issued to the engine.
Corrupt Packets/sec	Corrupt Packets/sec is the rate at which corrupt network packets are being received.
Current IO	Current IO is the current number of file IOs issued by the engine which have not yet completed.
Current Reads	Current Reads is the current number of file reads issued by the engine which have not yet completed.
Current Writes	Current Writes is the current number of file writes issued by the engine which have not yet completed.
Cursor	Cursor is the number of declared cursors that are currently being maintained by the engine.
Dirty Pages	Dirty Pages is the number of pages in the cache which must be written out and which do not belong to temporary files.
Disk Index Internal Reads/sec	Disk Index Internal Reads/sec is the rate at which index internal-node pages are being read from disk.
Disk Index Leaf Reads/sec	Disk Index Leaf Reads/sec is the rate at which index leaf pages are being read from disk.
Disk Reads/sec	Disk Reads/sec is the rate at which pages are being read from file.
Disk SyncReads/sec	Disk SyncReads/sec is the rate at which pages are being read synchronously from disk.
Disk SyncWrite Other/sec	Disk SyncWrite Other/sec is the rate at which pages are being written synchronously to disk for a reason not covered by other "Disk SyncWrites _____/sec" counters.
Disk SyncWrites Checkpoint/sec	Disk SyncWrites Checkpoint/sec is the rate at which pages are being written synchronously to disk for a checkpoint.
Disk SyncWrites	Disk SyncWrites Extend/sec is the rate at which pages are being written synchronously to disk while extending a

Statistic	Description
Extend/sec	database file.
Disk SyncWrites Free Current/sec	Disk SyncWrites Free Current/sec is the rate at which pages are being written synchronously to disk to free a page that cannot remain in the in-memory free list.
Disk SyncWrites Free Push/sec	Disk SyncWrites Free Push/sec is the rate at which pages are being written synchronously to disk to free a page that can remain in the in-memory free list.
Disk SyncWrites Log/sec	Disk SyncWrites Log/sec is the rate at which pages are being written synchronously to the transaction log.
Disk SyncWrites Rollback/sec	Disk SyncWrites Rollback/sec is the rate at which pages are being written synchronously to the rollback log.
Disk SyncWrites/sec	Disk SyncWrites/sec is the rate at which pages are being written synchronously to disk. It is the sum of all the other "Disk SyncWrites _____/sec" counters.
Disk Table Reads/sec	Disk Table Reads/sec is the rate at which table pages are being read from disk.
Disk Waitreads/sec	Disk Waitreads/sec is the rate at which the engine is waiting 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.
Disk Waitwrites/sec	Disk Waitwrites/sec is the rate at which the engine is waiting synchronously for the completion of a write IO operation which was originally issued as an asynchronous write.
Disk Writes/sec	Disk Writes/sec is the rate at which modified pages are being written to disk.
Dropped Packets/sec	Dropped Packets/sec is the rate at which network packets are being dropped due to lack of buffer space.
Extend Database/sec	Extend Database/sec is the rate (in pages/sec) at which the database file is being extended.
Extend Temporary File/sec	Extend Temporary File/sec is the rate (in pages/sec) at which temporary files are being extended.
Free Buffers	Number of free network buffers.
Freelist Write Current/sec	Freelist Write Current/sec is the rate at which pages that cannot remain in the in-memory free list are being freed.
Freelist Write Push/sec	Freelist Write Push/sec is the rate at which pages that can remain in the in-memory free list are being freed.

Statistic	Description
Full compares/sec	Full compares/sec is the rate at which comparisons beyond the hash value in an index must be performed.
IO to Recover	IO to Recover is the estimated number of IO operations required to recover the database.
Idle Active/sec	Idle Active/sec is the rate at which the engine's idle thread becomes active to do idle writes, idle checkpoints, etc.
Idle Checkpoints/sec	Idle Checkpoints/sec is the rate at which checkpoints are completed by the engine's idle thread. An idle checkpoint occurs whenever the idle thread writes out the last dirty page in the cache.
Idle Waits/sec	Idle Waits/sec is the number of times per second that the server goes idle waiting for IO completion or a new request.
Idle Writes/sec	Idle Writes/sec is the rate at which disk writes are being issued by the engine's idle thread.
Index Fills	Index Fills is the number of times a new temporary merge index is created.
Index Merges	Index Merges is the number of times a temp index has been merged into a main index
Index adds/sec	Index adds/sec is the rate at which entries are being added to indexes.
Index lookups/sec	Index lookups/sec is the rate at which entries are being looked up in indexes.
Lock Table Pages	Lock Table Pages is the number of pages used to store lock information.
Main Heap Pages	Main Heap Pages is the number of pages used for global engine data structures.
Map Pages	Map Pages is the number of map pages used for accessing the lock table, frequency table, and table layout.
Maximum IO	Maximum IO is the maximum value that "Current IO" has reached.
Maximum Reads	Maximum Reads is the maximum value that "Current Reads" has reached.
Maximum Writes	Maximum Writes is the maximum value that "Current Writes" has reached.
Multi-packets Received/sec	Multi-packets Received/sec is the rate at which multi-packet deliveries are being received.
Multi-packets	Multi-packets Transmitted/sec is the rate at which multi-

Statistic	Description
Transmitted/sec	packet deliveries are being transmitted.
Open cursors	Open cursors is the number of open cursors that are currently being maintained by the engine.
Packets Received/sec	Packets Received/sec is the rate at which network packets are being received.
Packets Transmitted/sec	Packets Transmitted/sec is the rate at which network packets are being transmitted.
Page Relocations/sec	Page Relocations/sec is the rate at which relocatable heap pages are being read from the temporary file.
Pending requests/sec	Pending requests/sec is the rate at which the engine is detecting the arrival of new requests.
Ping1/sec	Ping1/sec is the rate at which ping requests which go all the way down into the engine are serviced.
Ping2/sec	Ping2/sec is the rate at which ping requests which are turned around at the top of the protocol stack are serviced.
Procedure Pages	Procedure Pages is the number of relocatable heap pages used for procedures.
Read Hints Used/sec	Read Hints Used/sec is the rate at which page-read operations are being satisfied immediately from cache thanks to a earlier read hint.
Read Hints/sec	A read hint is an asynchronous read operation for a page that the database engine is likely to need soon. Read Hints/sec is the rate at which such read operations are being issued.
Recovery Urgency	Recovery Urgency is expressed as a percentage. See documentation for details.
Redo Free Commits/sec	A "Redo Free Commit" occurs when a commit of the transaction (redo) log is requested but the log has already been written (so the commit was done for "free").
Redo Rewrites/sec	Redo Rewrites/sec is the rate at which pages that were previously written to the transaction log (but were not full) are being written to the transaction log again (but with more data added).
Redo Writes/sec	Redo Writes/sec is the rate at which pages are being written to the transaction (redo) log.
Relocatable Heap Pages	Relocatable Heap Pages is the number of pages used for relocatable heaps (cursors, statements, procedures, triggers, views, etc.).

Statistic	Description
Remoteput Wait/sec	Remoteput Wait/sec is the rate at which the communication link must wait because it does not have buffers available to send information. This statistic is collected for NetBIOS (both sessions and datagrams) and IPX protocols only.
Requests/sec	Requests/sec is the rate at which the engine is being entered to allow it to handle a new request or continue processing an existing request.
Rereads Queued/sec	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. Rereads Queued/sec is the rate at which this condition is occurring.
Rereceived Packets/sec	Rereceived Packets/sec is the rate at which duplicate network packets are being received.
Retransmitted Packets/sec	Retransmitted Packets/sec is the rate at which network packets are being retransmitted.
Rollback Log Pages	Rollback Log Pages is the number of pages in the rollback log.
Rollback/sec	Rollbacks/sec is the rate at which Rollback requests are being handled.
SQL Anywhere	The SQL Anywhere object provides information about the Sybase SQL Anywhere Server, Client or Engine.
Sends Failed/sec	Sends Failed/sec is the rate at which the underlying protocol(s) failed to send a packet.
Statement	Statements is the number of prepared statements that are currently being maintained by the engine.
TotalBuffers	Total number of network buffers.
Trigger Pages	Trigger Pages is the number of relocatable heap pages used for triggers.
Unscheduled requests	Unscheduled requests is the number of requests that are currently queued up waiting for an available engine thread.
View Pages	View Pages is the number of relocatable heap pages used for views.
Voluntary blocks/sec	Voluntary blocks/sec is the rate at which engine threads voluntarily block on pending disk IO.
Waitread Full Compare/sec	Waitread Full Compare/sec is the rate at which read requests associated with a full comparison (a comparison beyond the hash value in an index) must be satisfied by a

Statistic	Description
	synchronous read operation.
Waitread Optimizer/sec	Waitread Optimizer/sec is the rate at which read requests posted by the optimizer must be satisfied by a synchronous read operation.
Waitread Other/sec	Waitread Other/sec is the rate at which read requests from other sources must be satisfied by a synchronous read operation.
Waitread SysConnection/sec	Waitread SysConnection/sec is the rate at which read requests posted from the system connection 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 a any client connection.
Waitread Temporary Table/sec	Waitread Temporary Table/sec is the rate at which read requests for a temporary table must be satisfied by a synchronous read operation.

CHAPTER 22

Database Collations

About this chapter

This chapter documents the built-in collations provided with SQL Anywhere and describes how to construct custom collations.

A collation is an ordering for a particular character set. Choosing the proper collation ensures that the sorting and comparison operations produce the proper results for the native language of the user.

Contents

Topic	Page
Collation overview	348
Support for multibyte character sets	352
Choosing a character set	354
Creating custom collations	357
The collation file format	358

Collation overview

When you create a SQL Anywhere database, you specify a collating sequence or **collation** to be used by the database. A collation is a sorting order for characters in the database. Whenever the database compares strings, sorts strings, or carries out other string operations such as case conversion, it does so using the collation sequence. The database carries out sorting and string comparison when statements such as the following are submitted:

- Queries with an ORDER BY clause.
- Expressions that use string functions, such as LOCATE, SIMILAR, SOUNDEX.
- Conditions using the LIKE keyword.

The database also uses character sets in identifiers (column names and so on). In deciding whether a string is a valid or unique identifier, the database is using the database collation.

Character sets in applications and databases

For character strings to be both sorted properly by the database and displayed properly in an application, it is important that the application and the database are using the same, or at least compatible, character sets. This section describes how the responsibilities for handling characters are divided between the database and the application.

Important components

Different aspects of character storage and display are separated out by operating systems. The following aspects are treated distinctly:

- Each operating system has a **character set** available to it. A character set is a set of symbols, including letters, digits, spaces and other symbols.
- Each operating system employs a character **encoding**, in which each character is mapped onto one or more bytes of information, typically represented as hexadecimal numbers.
- Characters are displayed on a screen using a **font**, which is a mapping between characters in the character set and their appearance.
- Operating systems also use a **keyboard mapping** to map keys or key combinations on the keyboard to characters in the character set.

- A **collation** is a combination of a character encoding (a map between characters and hexadecimal numbers) and a sorting order for the characters.

The database engine receives strings as a stream of hexadecimal numbers, which it associates with characters and sorts according to the collation specified when the database was created.

Character strings

It is up to the operating system of the computer on which the client application is running to handle the following aspects of character strings:

- Which character is stored when a particular key on the keyboard is typed.
- What a character looks like on your computer screen.
- What characters are available to the application.
- What hexadecimal encoding is stored for each character.

Notes

- It is important that the operating system and the database be using compatible character sets, character set encodings, and (if the application itself does any string sorting or comparison) collation sequences if information is to be handled and displayed in a consistent manner by the database engine and the client application.
- The ODBC interface provides mechanisms for translating strings on their way to the database engine, and on the way back, using a translation driver. For more information, see "Files needed for ODBC connections" in the chapter "Connecting to a Database".

Character encodings

There are several different systems for encoding character sets as hexadecimal numbers. This section lists some of the more common.

Single-byte character sets and code pages

Many languages have few enough characters to be represented in a single-byte character set. In such a character set, each character is represented by a single two-digit hexadecimal number.

At most 256 characters can be represented in a single-byte set. No single single-byte character set can hold all the characters used, including accented characters, internationally. IBM developed a set of code pages in which each code page describes a set of characters appropriate for one national language. For example, code page 869 contains the Greek character set, code page 850 contains an international character set suitable for representing many characters in a variety of languages.

SQL Anywhere supports a set of single-byte collations (code pages and collation orderings) suitable for many languages of European origin.

For information about choosing a single-byte collation for your database, see "Choosing a character set" on page 354.

Multibyte character sets

Some languages have many more than 256 characters, and these can be represented in multibyte character sets. In addition, there are character sets that use the much larger number of characters available in a multibyte representation to represent characters from many languages in a single, more universal, character set.

Multibyte character sets are of two types. Some are variable width, in which some characters are single-width characters, others are double-byte, and so on. Other sets are fixed width, in which all characters in the set have the same number of bytes. SQL Anywhere supports variable-width character sets.

For information on the multibyte character sets supported by SQL Anywhere, see "Support for multibyte character sets" on page 352.

Displaying your current character settings

Each operating system has its own system for handling character sets, encodings, and collation sequences. To find out information about the current settings on your operating system, you can:

- In DOS or OS/2, type chcp at the command prompt to display the current code page.
- In Windows and Windows NT, see the International Settings in the Control Panel.

Character sets are stored by the operating system and sent to a database as a set of hexadecimal numbers.

Collation sequences

Roughly speaking, a collation sequence is a sorting order for characters in a character set encoding or code page. The collation sequence is based on the encoded value of the characters.

The collation sequence includes the notion of alphabetic ordering of letters, and extends it to include all characters in the character set, including digits and space characters.

The collation sequence includes more information than a simple ordering. Each character is assigned to a sort position, and the sort position defines the position of that character in any comparison or sorting of character strings.

Associating more than one character with each sort position

More than one character can be associated with each sort position. This is useful if you wish, for example, to treat an accented character the same as the character without an accent. Two characters with the same sort position are considered to be identical in all ways by the database. Therefore, if a collation assigned the characters *a* and *e* to the same sort order, then a query with the following search condition:

```
WHERE col1 = 'want'.
```

is satisfied by a row for which col1 contains the entry "went".

At each sort position, lower- and uppercase forms of a character can be indicated. For case-sensitive databases, the lower- and uppercase characters are not treated as equivalent. For case-insensitive databases, the lower- and uppercase versions of the character are considered equivalent.

Support for multibyte character sets

SQL Anywhere supports several multibyte character sets, including the following:

Multibyte character set	Description
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	UTF8 is a variable-byte encoding of 4-byte unicode (UCS-4). SQL Anywhere supports UTF8 characters up to 4 bytes in length, while UTF8 may use up to 6 bytes.

This section describes how SQL Anywhere handles multibyte character sets. The description applies to the supported collations and to any custom collations.

Variable length character sets

SQL Anywhere supports variable length character sets. In these sets, some characters are represented by one byte, and some by more than one, to a maximum of four bytes. The value of the first byte in any character indicates the number of bytes used for that character, and also indicates whether the character is a space character, a digit, or an alphabetic (alpha) character. SQL Anywhere does not support fixed-length multibyte character sets such as 2-byte UNICODE or 4-byte UNICODE.

Example

As an example, characters in the Shift-JIS character set are of either one or two bytes in length. If the hex value of the first byte is in the range 81-9F or E0-EF (decimal values 129-159 or 224-239) then the character is a two-byte character and the subsequent byte (called a **follow byte**) completes the character. If the first byte is outside this range, the character is a single-byte character and the next byte is the first byte of the following character.

The properties of any Shift-JIS character can be read from its first byte also. Characters with a first byte in the (hex) range 09 to 0D, or 20, are space characters, those in the ranges 41 to 5A, 61 to 7A, 81 to 9F or E0 to EF are alpha characters (letters), and those in the range 30 to 39 are digits.

In building custom collations, you can specify which ranges of values for the first byte signify single and double byte (or more) characters, and which specify space, alpha, and digit characters. However, all first bytes of value less than 40 (hex 28) must be single byte characters, and no follow bytes may have values less than 40. This restriction is satisfied by all known current encodings.

First-byte collation orderings

A sorting order for characters in a multibyte character set can be specified only for the first byte. Characters that have the same first byte are sorted according to the hexadecimal value of the following bytes. If the two characters are the same up to the length of the shorter of the two, the longer character is greater than the shorter.

Choosing a character set

The best collation to use varies depending primarily on the language of the user and the code pages available in the user's machine. This will, of course, vary from one country or region to another.

The following table shows the built-in collations provided with SQL Anywhere. The table and the corresponding collations were derived from several manuals from IBM concerning National Language Support, subject to the restrictions mentioned above. (This table represents the best information available at the time of writing. Due to recent rapid geopolitical changes, the table may contain names for countries that no longer exist.)

Country	Language	Primary Code Page	Primary Collation	Secondary Code Page	Secondary Collation
Argentina	Spanish	850	850ESP	437	437ESP
Australia	English	437	437LATIN1	850	850LATIN1
Austria	German	850	850LATIN1	437	437LATIN1
Belgium	Belgian Dutch	850	850LATIN1	437	437LATIN1
Belgium	Belgian French	850	850LATIN1	437	437LATIN1
Belarus	Belarussian	855	855CYR		
Brazil	Portuguese	850	850LATIN1	437	437LATIN1
Bulgaria	Bulgarian	855	855CYR	850	850CYR
Canada	Cdn French	850	850LATIN1	863	863LATIN1
Canada	English	437	437LATIN1	850	850LATIN1
Croatia	Croatian	852	852LATIN2	850	850LATIN2
Czech Republic	Czech	852	852LATIN2	850	850LATIN2
Denmark	Danish	850	850DAN		
Finland	Finnish	850	850SVE	437	437SVE
France	French	850	850LATIN1	437	437LATIN1
Germany	German	850	850LATIN1	437	437LATIN1
Greece	Greek	869	869ELL	850	850ELL

Country	Language	Primary Code Page	Primary Collation	Secondary Code Page	Secondary Collation
Hungary	Hungarian	852	852LATIN2	850	850LATIN2
Iceland	Icelandic	850	850ISL	861	861ISL
Ireland	English	850	850LATIN1	437	437LATIN1
Israel	Hebrew	862	862HEB	856	856HEB
Italy	Italian	850	850LATIN1	437	437LATIN1
Mexico	Spanish	850	850ESP	437	437ESP
Nether-lands	Dutch	850	850LATIN1	437	437LATIN1
New Zealand	English	437	437LATIN1	850	850LATIN1
Norway	Norwegian	865	865NOR	850	850NOR
Peru	Spanish	850	850ESP	437	437ESP
Poland	Polish	852	852LATIN2	850	850LATIN2
Portugal	Portuguese	850	850LATIN1	860	860LATIN1
Romania	Romanian	852	852LATIN2	850	850LATIN2
Russia	Russian	866	866RUS	850	850RUS
S. Africa	Afrikaans	437	437LATIN1	850	850LATIN1
S. Africa	English	437	437LATIN1	850	850LATIN1
Slovak Republic	Slovakian	852	852LATIN2	850	850LATIN2
Slovenia	Slovenian	852	852LATIN2	850	850LATIN2
Spain	Spanish	850	850ESP	437	437ESP
Sweden	Swedish	850	850SVE	437	437SVE
Switzer-land	French	850	850LATIN1	437	437LATIN1
Switzer-land	German	850	850LATIN1	437	437LATIN1
Switzer-land	Italian	850	850LATIN1	437	437LATIN1
Turkey	Turkish	857	857TRK	850	850TRK
UK	English	850	850LATIN1	437	437LATIN1

Country	Language	Primary Code Page	Primary Collation	Secondary Code Page	Secondary Collation
USA	English	437	437LATIN1	850	850LATIN1
Venezuela	Spanish	850	850ESP	437	437ESP
Yugoslavia	Maced-onian	852	852LATIN2	850	850LATIN2
Yugoslavia	Serbian Cyrillic	855	855CYR	852	852CYR
Yugoslavia	Serbian Latin	852	852LATIN2	850	850LATIN2

When creating a new database

A user creating a new database should find the line with the country/language that they wish to use, then pick either the primary or secondary collation, depending on which code page is in use in their computer. (The chcp command will display the current code page number.) If their particular combination is not present, then another line with a satisfactory combination may be used, or a custom collation may be required.

Creating custom collations

To create a custom collation file, first use the DBCOLLAT utility to extract a collation from an existing database. A collation as close as possible to the wanted collation should be chosen.

Example 1

The following statement extracts the collation from a database named collat.db into a file named CUSTOM.COL:

```
dbcollat -c "dbf=collat.db;uid=dba;pwd=sql"
custom.col
```

When you use DBCOLLAT to extract a collation from an existing database, the database does not need to contain any information. If you do not currently have a database using the collation on which you want to base a custom collation, you can create such as database using the Sybase Central administration tool or the DBINIT command line utility.

You can see a listing of available collations using DBINIT with the *-l* (lowercase L) option.

```
dbinit -l
```

Example 2

The following statement creates a database named COLLAT.DB using the Shift-JIS collation:

```
dbinit -z SJIS collat.db
```

Once you have extracted a collation file, you need to edit the file using a text editor to produce your custom collation file. The format of the custom collation file is described below.

To build a database that uses the custom collation file, use DBINIT with the *-z* switch. The following statement builds a database named COLLAT2.DB using the CUSTOM.COL custom collation file.

```
dbinit -z custom.col collat2.db
```

The collation file format

This section describes the collation file format. Collation files may include the following elements:

- Comment lines, which are ignored by the database.
- A title line.
- A collation sequence section.
- An Encodings section (multibyte character sets only).
- A Properties section (multibyte character sets only).

Comment lines

In the collation file, spaces are generally ignored. Comment lines start with either % or --.

The title line

The first non-comment line must be of the form:

```
Collation label (name)
```

In this statement:

Item	Description
Collation	a keyword and is required
label	is the collation label and appears in the system tables as SYS.SYSCOLLATION.collation_label and SYS.SYSINFO.default_collation. The label must be no more than 10 characters, and must not be the same as one of the built-in collations. (In particular, do not leave the collation label unchanged.)
name	a descriptive term, used for documentation purposes. The name should be no more than 128 characters For example, the Shift-JIS collation file contains the following collation line, with label SJIS and name (Japanese Shift-JIS Encoding): `Collation SJIS (Japanese Shift-JIS Encoding)`

The collation sequence section

After the title line, each noncomment line describes one position in the collation. The ordering of the lines determines the sort ordering used by the database, and also determines the result of comparisons. Characters on lines appearing higher in the file (closer to the beginning) sort before characters that appear later.

The form of each line in the sequence is:

```
[sort-position] : character
```

or

```
[sort-position] : character [lowercase uppercase]
```

where:

Descriptions of arguments

Argument	Description
sort-position	is optional and specifies the position at which the characters on that line will sort. Smaller numbers represent a lesser value, so will sort closer to the beginning of the sorted item. Typically, the sort-position is omitted, and the characters sort immediately following the characters from the previous sort position
character	is the character whose sort-position is being specified
lowercase	is optional and specifies the lowercase equivalent of the character. If not specified, then the character has no lowercase equivalent
uppercase	is optional and specifies the uppercase equivalent of the character. If not specified, then the character has no uppercase equivalent

Multiple characters may appear on one line, separated by commas (,). In this case, these characters are sorted and compared as if they were the same character.

Specifying character and sort-position

Each character and sortposition is specified in one of the following ways:

Specification	Description
\dnnn	Decimal number, using digits 0-9 (such as \d001)
\xhh	Hexadecimal number, using 2 digits 0-9 and/or letters a-f or A-F (such as \xB4)
'c'	Any character in place of c (such as ',')

Specification	Description
c	Any character other than quote ('), back-slash (\), colon (:) or comma (,). These characters must use one of the previous forms. The following are some sample lines for a collation:

```
% Sort some letters in alphabetical order
: A a A
: a a A
: B b B
: b b B
% Sort some E's from code page 850,
% including some accented extended
characters:
: e e E, \x82 \x82 \x90, \x8A \x8A \xD4
: E e E, \x90 \x82 \x90, \xD4 \x8A \xD4
% Sort some special characters at the end:
: ' '
: _
: \xF2
: \xEE
: \xF0
: -
: ','
: ;
: ':'
: !
```

Other syntax notes

For databases using case-insensitive sorting and comparing (no *-c* specified on the DBINIT command line), the lowercase and uppercase mappings are used to find the lowercase and uppercase characters that will be sorted together.

For multibyte character sets, the first byte of a character is listed in the collation sequence, and all characters with the same first byte are sorted together, and ordered according to the value of the second byte. For example, the following is part of a Shift-JIS collation file:

```
:  \xfb
:  \xfc
:  \xfd
```

In this collation, all characters with first byte \xfc come after all characters with first byte \xfb and before all characters with first byte \xfd. The two-byte character \xfc \x01 would be ordered before the two-byte character \xfc \x02.

Any characters omitted from the collation will be added to the collation at the position equal to their binary value. DBINIT issues a message for each omitted character. However, it is recommended that any collation contain all 256 characters (first bytes).

The Encodings section

The Encodings section is optional, and follows the collation sequence. It is not useful for single-byte character sets.

The Encodings section lists those combinations of bytes which are valid characters. The format of the section may be described by example.

The Shift-JIS Encodings section is as follows:

```
Encodings:
[\x00-\x80,\xa0-\xdf,\xf0-\xff]
[\x81-\x9f,\xe0-\xef][\x00-\xff]
```

The first line following the section title lists valid single-byte characters. The square brackets enclose a comma-separated list of ranges. Each range is listed as a hyphen-separated pair of values. In the Shift-JIS collation, values \x00 to \x80 are valid single-byte characters, but \x81 is not a valid single-byte character.

The second line following the section title lists valid double-byte characters. Any combination of bytes from ranges in the first pair of brackets with those in the second are valid characters. Therefore \x81\x00 is a valid double-byte character, but \xd0 \x00 is not.

The Properties section

The Properties section is optional, and follows the Encodings section. It is not useful for single-byte character sets.

If a Properties section is supplied, an Encodings section must be supplied also.

The Properties section lists values for the first-byte of each character that represent characters, digits, or spaces.

The Shift-JIS Properties section is as follows:

```
Properties:
space: [\x09-\x0d,\x20]
digit: [\x30-\x39]
alpha: [\x41-\x5a,\x61-\x7a,\x81-\x9f,\xe0-\xef]
```

This indicates that characters with first bytes \x09 to \x0d, as well as \x20, are to be treated as space characters, digits are found in the range \x30 to \x39 inclusive, and alphabetic characters in the four ranges \x41-\x5a, \x61-\x7a, \x81-\x9f, and \xe0-\xef.

CHAPTER 23

Importing and Exporting Data

About this chapter

Transferring large amounts of data into and out of your database may be necessary in several situations. For example:

- Importing an initial set of data into a new database
- Exporting data into other applications, such as spreadsheets
- Building new copies of a database with modified structure
- Creating extractions of a database

This chapter describes how to import data to and export data from SQL Anywhere databases, both in text form and in other formats.

Contents

Topic	Page
Import and export overview	364
Exporting data from a database	366
Importing data into a database	370
Tuning bulk operations	373

Import and export overview

SQL Anywhere supports import and export of individual tables or a complete database using text files. Importing and exporting data to other formats, such as spreadsheet program formats, is available from the ISQL utility.

The LOAD TABLE statement and the UNLOAD TABLE statement are the SQL statements for loading and unloading tables using text files. The INPUT statement and the OUTPUT statement are ISQL commands for importing and exporting data using a variety of formats.

You can unload individual tables or a complete database from SQL Central, using the Unload Database wizard. You can also unload individual tables or a complete database using the DBUNLOAD command line utility. The DBUNLOAD utility is accessible from ISQL.

This chapter describes the LOAD TABLE and UNLOAD TABLE SQL statements, as well as the use of SQL Central, DBUNLOAD, and ISQL to import and export data.

Input and output data formats

The LOAD TABLE and UNLOAD TABLE import and export text files, with one row per line, and values separated by a delimiter.

The ISQL INPUT and OUTPUT statements support the following file formats:

File Format	Description
ASCII	A text file, one row per line, with values separated by a delimiter. String values are optionally enclosed in apostrophes (single quotes). This is the same as the format used by LOAD TABLE and UNLOAD TABLE
DBASEII	DBASE II format
DBASEIII	DBASE III format
DIF	Data Interchange Format
FIXED	Data records are in fixed format with the width of each column either the same as defined by the column's type or specified as a parameter
FOXPRO	FoxPro format
LOTUS	Lotus workspace format

File Format	Description
WATFILE	WATFILE format. There is an ISQL option for the default input format and one for the default output format. For details on how to specify this option, see "SET OPTION statement" in the chapter "Watcom-SQL Statements". Alternatively, the file format can be specified on each INPUT statement or OUTPUT statement

Exporting data from a database

You can export data from your database using one of the following methods:

- The UNLOAD TABLE statement, for efficient export of text files
- The ISQL OUTPUT statement, for slower but more flexible export to a variety of file formats
- The DBUNLOAD utility, for text export of more than one table

This section describes each of these methods, and also describes some tips for dealing with NULL output.

Unloading data using the UNLOAD TABLE statement

The UNLOAD TABLE statement is for efficient export of data from a database table to a text file. You must have SELECT permission on the table to use the UNLOAD TABLE statement.

Example 1

The following statement unloads the department table from the sample database into the file DEPT.TXT in the current directory of the drive on which the database engine is running. If you are running against a network server, the command unloads the data into a file on the server machine, not the client machine.

```
UNLOAD TABLE department
TO 'dept.txt'
```

The dept.txt file has the following contents:

```
100,'R & D',501
200,'Sales',902
300,'Finance',1293
400,'Marketing',1576
500,'Shipping',703
```

Notes

- Each row of the table is output on a single line of the output file.
- No column names are exported.
- The columns are separated, or *delimited*, by a comma. The delimiter character can be changed using the DELIMITED BY clause.
- The fields are not fixed-width fields. Only the number of characters in each entry are exported, not the full width of the column.
- The character data in the dept_name column is enclosed in single quotes. The single quotes can be turned off using the QUOTES clause.

- The data is exported by primary key values. This makes reloading quicker. You can export data in the order in which it is stored using the ORDER OFF clause.
- The file name is relative to the database engine's current directory, not the current directory of the client application. Also, the file name is passed to the database engine as a string, so that special characters need to be preceded by a backslash (\) in order to be recognized. For example, the following statement unloads a table into the file c:.temp.output.dat:

```
UNLOAD TABLE employee TO 'c:\temp\output.dat'
```

For more information on the syntax, see "UNLOAD TABLE statement" in the chapter "Watcom-SQL Statements".

Example 2

The following example uses explicit settings for the DELIMITED BY and QUOTES clauses:

```
UNLOAD TABLE department
TO 'dept.txt'
DELIMITED BY '$'
QUOTES OFF
```

The resulting dept.txt file has the following contents:

```
100$R & D$501
200$Sales$902
300$Finance$1293
400$Marketing$1576
500$Shipping$703
```

If a delimiter character appears in a value in a column and the QUOTES option is turned off, the character is replaced by its hexadecimal value preceded by \x to prevent ambiguity as to the end of the value. For example:

```
UNLOAD TABLE department
TO 'dept.txt'
DELIMITED BY '&'
QUOTES OFF
```

yields the following output file.

```
100&R \x26 D&501
200&Sales&902
300&Finance&1293
400&Marketing&1576
500&Shipping&703
```

Exporting data using the ISQL OUTPUT statement

Data can be exported from a database to a variety of file formats using the ISQL OUTPUT statement. This statement exports the results of the current query (the one displayed in the ISQL. data window) and puts the results into a specified file. The output format can be specified on the output command.

For example, the following commands extract the employee table to a dBaseIII format file:

```
SELECT *
FROM employee;
OUTPUT TO employee.dbf
FORMAT dbaseiii;
```

Output redirection

Output redirection can be used to export data as an alternative to the OUTPUT statement.

The output of any command can be redirected to a file or device by putting the ># redirection symbol anywhere on the command. The redirection symbol must be followed by a file name, as follows:

```
># filename
```

Output redirection is most useful on the SELECT statement. The SET OUTPUT_FORMAT command can be used to control the format of the output file.

Example

The >& redirection symbol redirects all output including error messages and statistics for the command on which it appears. For example:

```
SELECT *
FROM employee >& filename
```

outputs the SELECT statement to the file, followed by the output from the SELECT statement and some statistics pertaining to the command.

The >& redirection is useful for getting a log of what happens during a READ command. The statistics and errors of each command are written following the command in the redirected output file.

If two > characters are used in a redirection symbol instead of one (>>#, >>&.), then the output is appended to the specified file instead of replacing the contents of the file. For output from the SELECT command, headings are output if and only if the output starts at the beginning of the specified file and the output format supports headings.

NULL value output

The most common reason to extract data is for use in other software products. When your data includes NULL values, the other software package may not understand the NULL values.

There is an ISQL option (NULLS) that allows you to choose how NULL values are output. Alternatively, you can use the IFNULL function to output a specific value whenever there is a NULL value.

For information on setting ISQL options, see "SET OPTION statement" in the chapter "Watcom-SQL Statements".

Unloading a database using DBUNLOAD

The DBUNLOAD utility supplied with SQL Anywhere is used to unload an entire database in ASCII comma-delimited format and to create the necessary ISQL. command files to completely recreate your database. 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.

It is particularly useful if you want to rearrange your tables in the database. Use DBUNLOAD to create the necessary command files and modify them as needed.

Exporting a list of tables

The DBUNLOAD utility can also export a list of tables, rather than the entire database. This is useful for retrieving data from a corrupted database that cannot be entirely unloaded.

The following statement unloads the data from the sample database (assumed to be running on the default database server with the default database name) into a set of files in the c: temp directory. A command file to rebuild the database from the data files is created with the default name reload.sql in the current directory.

```
dbunload -c "dbn=sademo;uid=dba;pwd=sql" c:\temp
```

For Windows 3.x, the DBUNLOAD executable is named DBUNLOAW.EXE. For a full description of dbunload utility command-line switches, see the section "The Unload utility" in the chapter "SQL Anywhere Components".

Importing data into a database

You can import data into your database using one of the following methods:

- The LOAD TABLE statement, for efficient import of text files.
- The ISQL INPUT statement, for slower but more flexible import of a variety of file formats.
- Interactive input using the INSERT Statement or the ISQL INPUT statement.

This section describes each of these methods, and also describes some tips for dealing with incompatible data structure and conversion errors.

Loading data using the LOAD TABLE statement

The LOAD TABLE statement is for efficient importing of data from a text file into a database table. To use the LOAD TABLE statement, the table must exist and have the same number of columns as the input file has fields, defined on compatible data types. In order to use the LOAD TABLE statement, the user must have INSERT permission on the table.

Example

If the department table had all its rows deleted, the following statement would load the data from the file dept.txt into the department table:

```
LOAD TABLE department
FROM 'dept.txt'
```

The LOAD TABLE statement appends the contents of the file to the existing rows of the table; it does not replace the existing rows in the table. You can use the TRUNCATE TABLE statement to remove all the rows from a table.

Neither the TRUNCATE TABLE statement nor the LOAD TABLE statement fires triggers, including referential integrity actions such as cascaded deletes.

The LOAD TABLE statement has many of the same options as the UNLOAD TABLE statement.

For a description of column delimiters, use of quotes, and file names, see the section "Unloading data using the UNLOAD TABLE statement" on page 366.

The LOAD TABLE statement has the additional STRIP clause. The default setting (STRIP ON) strips trailing blanks from values before they are inserted. To keep trailing blanks, use the STRIP OFF clause in your LOAD TABLE statement.

For a full description of the LOAD TABLE statement syntax, see "LOAD TABLE statement" in the chapter "Watcom-SQL Statements".

Importing data using the ISQL INPUT statement

Data with the same structure as existing database tables can be loaded into your database from a file using the ISQL INPUT statement.

The ISQL INPUT statement is less efficient than the LOAD TABLE statement for importing text files. However, the INPUT statement supports several different file formats, whereas the LOAD TABLE statement can be used only for text files.

The INPUT command can be entered in ISQL as follows:

```
INPUT INTO t1
FROM file1
FORMAT ASCII;

INPUT INTO t2
FROM file2
FORMAT FIXED
COLUMN WIDTHS (5, 10, 40, 40 );
...
```

These statements could be put in a command file which can then be executed in ISQL for modification and reference.

For more information about command files, see the tutorial chapter "Command Files".

Loading data interactively

There are two commands that can be used to input data interactively. You can use the insert command:

```
INSERT INTO T1 VALUES ( ... )
```

to insert a single row at a time or you can use the input command:

```
INPUT INTO T1 PROMPT
```

which gives you a full screen to type in data in the current input format (controlled by the ISQL INPUT_FORMAT option).

Handling conversion errors on data import

When you are loading data from external sources, there may be errors in the data. For example, there may be dates that are not valid dates and numbers that are not valid numbers. There is an ISQL database option (CONVERSION_ERROR) that allows you to ignore conversion errors by converting them to NULL values.

For information on setting ISQL database options, see "SET OPTION statement" in the chapter "Watcom-SQL Statements".

Loading data that does not match the table structure

The structure of the data to be loaded into a table does not always match the structure of the destination table itself. For example, the column data types may be different, or in different order, or there may be extra values in the data to be imported that do not match columns in the destination table.

You can use a stepwise approach to load data that has a different structure to the destination table:

- Using the LOAD TABLE statement, load the data into a temporary table that has a structure matching that of the input file.
- Use the INSERT statement with a FROM SELECT clause to extract and summarize data from the temporary table and put it into one or more of the permanent database tables.

DECLARE TEMPORARY TABLE statement

If you are loading a set of data once and for all, you should make the temporary table using the DECLARE TEMPORARY TABLE statement. A declared temporary table exists only for the duration of a connection or, if defined inside a compound statement, of the compound statement.

CREATE TABLE statement

If you are loading data of a similar structure repeatedly, you should make the temporary table using the CREATE TABLE statement, specifying a global temporary table. The definition of a created temporary table is held in the database permanently, but the rows exist only within a given connection.

Tuning bulk operations

Loading large volumes of data into a database can be very time consuming. There are a few things you can do to save time.

- Run ISQL or the client application on the same machine as the database engine. Loading data over the network adds extra communication overhead. This might mean loading new data during off hours.
- Place data files on a separate physical disk drive from the database file. This could avoid excessive disk head movement during the load.
- In Windows 3.x, if your computer has a 386, 486, or higher processor, use the 32-bit database engine rather than the 16-bit database engine.
- Increase the size of the database cache. See the chapter "SQL Anywhere Components" for a description of the database engine command line option -c. Eliminate disk cache in favor of database cache if the machine is a dedicated SQL Anywhere server.
- Start the database engine or network server with the -b switch for bulk operations mode. In this mode, the database engine does not keep a rollback log or a transaction log, it does not perform an automatic COMMIT before data definition commands, and it does not lock any records. Without a rollback log, you cannot use savepoints and aborting a command always causes transactions to roll back. Without automatic COMMIT, a ROLLBACK undoes everything since the last explicit COMMIT. Without a transaction log, there is no log of the changes. You should back up the database file before and after using bulk operations mode because, in this mode, your database is not protected against media failure (see the chapter "Backup and Data Recovery").
- The network server allows only one connection when you use the -b switch.
- If you have data that requires many COMMITs, running with the *-b* option may slow down database operation. At each COMMIT, the engine carries out a checkpoint; this frequent checkpointing can slow down the database engine.
- Extend the size of the database file using ALTER DBSPACE (see the chapter "Watcom-SQL Language Reference"). This command allows a database file 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. As well as improving performance for loading large amounts of data, it also serves to keep the database files more contiguous within the file system.

CHAPTER 24

Managing User IDs and Permissions

About this chapter

Each user of a database must be assigned a user ID: the name they type when connecting to the database. This chapter describes how to manage user IDs.

Contents

Topic	Page
An overview of database permissions	376
Managing individual user IDs and permissions	380
Managing groups	386
Database object names and prefixes	391
Using views and procedures for extra security	393
How SQL Anywhere assesses user permissions	396
Users and permissions in the system tables	397

An overview of database permissions

Proper management of user IDs and permissions is essential for users of a database to carry out their jobs effectively while maintaining the security and privacy of appropriate information within the database.

SQL Anywhere provides commands for assigning user IDs to new users of a database, granting and revoking permissions for database users, and finding out the current permissions of users.

Database permissions are assigned to user IDs. Throughout this chapter, the term **user** is used as a synonym for user ID. You should remember, however, that it is always on the basis of a particular user ID that permissions are granted and revoked.

Setting up individual user IDs

Even if there are no security concerns regarding a multiuser database, there are good reasons for setting up an individual user ID for each user, and the administrative overhead is very low if a group with the appropriate permissions is set up. Groups of users are discussed in this chapter. Among other reasons for using individual user IDs are the following:

- The log translation utility can selectively extract the changes made by individual users from a transaction log. This is very useful when troubleshooting or piecing together what happened if data is incorrect.
- The network server screen and the listing of connections in Sybase Central are both much more useful with individual user IDs, as you can tell which connections are which users.
- Row locking messages (with the BLOCKING option set to OFF) are more informative.

DBA authority overview

When a SQL Anywhere database is created using the DBINIT tool, a single usable user ID is created. This first user ID is DBA and the password is initially set to SQL. The DBA user ID is automatically given DBA permissions, also called DBA authority, within the database. This level of permission enables the DBA user ID to carry out any activity in the database: create tables, change table structures, create new user IDs, revoke permissions from users, and so on.

Users with DBA authority

A user with DBA authority is referred to as the database administrator or database owner. In this chapter frequent reference is made to the database administrator, or *the DBA*. This is a shorthand for *any user or users with DBA authority*.

Although DBA authority may be granted or transferred to other user IDs, in this chapter it is assumed that the DBA user ID is the database administrator, and the abbreviation *DBA* is used interchangeably to mean both the DBA user ID and any user ID with DBA authority.

Adding new users

The DBA has the authority to add new users to the database. As users are added, they are also granted permissions to carry out tasks on the database. Some users may need to simply look at the database information using SQL queries, others may need to add information to the database, and others may need to modify the structure of the database itself. Although some of the responsibilities of the DBA may be handed over to other user IDs, the DBA is responsible for the overall management of the database by virtue of the DBA authority.

The DBA has authority to create database objects and assign ownership of these objects to other user IDs

See the syntax of the commands for creating database objects, in the chapter "Watcom-SQL Language Reference".

Resource authority overview

Resource authority is the permission to create database objects, such as tables, views, stored procedures, and triggers. Resource authority may be granted only by the DBA to other users.

In order to create a trigger, a user needs ALTER permissions on the table to which the trigger applies, in addition to RESOURCE authority.

Ownership permissions overview

The creator of a database object becomes the owner of that object. Ownership of a database object carries with it permissions to carry out actions on that object. These are not assigned to users in the same way that other permissions in this chapter are assigned.

Owners

A user who creates a new object within the database is called the **owner** of that object, and automatically has permission to carry out any operation on that object. The owner of a table may modify the structure of that table, for instance, or may grant permissions to other database users to update the information within the table.

The DBA has permission to modify any component within the database, and so could delete a table created by another user, for instance. The DBA has all the permissions regarding database objects that the owners of each object has.

The DBA is also able to create database objects for other users, and in this case the owner of an object is not the user ID that executed the CREATE statement. A use for this ability is discussed in "Groups without passwords" on page 389. Despite this possibility, this chapter refers interchangeably to the owner and creator of database objects.

Table and views permissions overview

There are several distinct permissions that may be granted to user IDs concerning tables:

Permission	Description
ALTER	Permission to alter the structure of a table or create a trigger on a table
DELETE	Permission to delete rows from a table or view
INSERT	Permission to insert rows into a table or view
REFERENCES	Permission to create indexes on a table, and to create foreign keys that reference a table
SELECT	Permission to look at information in a table or view
UPDATE	Permission to update rows in a table or view. This may be granted on a set of columns in a table only
ALL	All the above permissions

Convenient and secure access to data in each table is set by granting combinations of these permissions to different sets of users.

Procedures permissions overview

There is only one permission that may be granted on a procedure, and that is the EXECUTE permission to execute (or CALL) the procedure.

Group permissions overview

Setting permissions individually for each user of a database can be a time-consuming and error-prone process. For most databases, permission management based on groups, rather than on individual user IDs, is a much more efficient approach.

You can assign permissions to a group in exactly the same way as to an individual user. You can then assign membership in appropriate groups to each new user of the database, and they gain a set of permissions by virtue of their group membership.

Example

For example, you may create groups for different departments in a company database (sales, marketing, and so on) and assign these groups permissions. Each salesperson is made a member of the sales group, and automatically gains access to the appropriate areas of the database.

Any user ID can be a member of several groups, and inherits all permissions from each of the groups.

Managing individual user IDs and permissions

This section describes how to create new users and grant permissions to them. For most databases, the bulk of permission management should be carried out using **groups**, rather than by assigning permissions to individual users one at a time. However, as groups are simply a user ID with special properties attached, you should read and understand this section before moving on to the discussion of managing groups.

Creating new users

A new user is added to a database by the DBA using the GRANT CONNECT statement. For example:

❖ **To add a new user to a database, with user ID M_Haneef and password welcome:**

1 From ISQL, connect to the database as a user with DBA authority.

2 Issue the SQL statement:

```
GRANT CONNECT TO M_Haneef
IDENTIFIED BY welcome
```

Only the DBA has the authority to add new users to a database.

Initial permissions for new users

By default, new users are not assigned any permissions beyond connecting to the database and viewing the system tables. In order to access tables in the database they need to be assigned permissions.

The DBA can set the permissions granted automatically to new users by assigning permissions to the special PUBLIC user group, as discussed in "Special groups" on page 389.

Creating users in Sybase Central

❖ **To create a user in Sybase Central:**

1 Connect to the database.

2 Click the Users and Groups folder for that database.

3 Double-click Add User. A Wizard is displayed, which leads you through the process.

For more information, see the Sybase Central online Help.

Changing a password

Changing a users password

You can change your password, or that of another user if you have DBA authority, using the GRANT statement. For example, the following command changes the password for user ID M_Haneef to new_password:

```
GRANT CONNECT TO M_Haneef
IDENTIFIED BY new_password
```

Changing the DBA password

The default password for the DBA user ID for all SQL Anywhere databases is SQL. You should change this password to prevent unauthorized access to your database. The following command changes the password for user ID DBA to new_password:

```
GRANT CONNECT TO "DBA"
IDENTIFIED BY new_password
```

DBA must be enclosed in double quotes in this expression as it is a SQL Anywhere keyword.

If you are using ISQL, it is a good idea to put your permission grants into a command file for reference and so that it can be modified and run again if it is necessary to recreate the permissions.

Granting DBA and resource authority

DBA and RESOURCE authority are granted in exactly the same manner as each other.

❖ **To grant resource permissions to user ID M_Haneef:**

1. Connect to the database as a user with DBA authority.
2. Type and execute the SQL statement:

```
GRANT RESOURCE TO M_Haneef.
```

For DBA authority, the appropriate SQL statement is:

```
GRANT DBA TO M_Haneef
```

Notes

- Only the DBA may grant DBA or RESOURCE authority to database users.
- DBA authority is very powerful, granting the ability to carry out any action on the database and access to all the information in the database. It is generally inadvisable to grant DBA authority to more than a very few people.

- You should consider giving users with DBA authority two user IDs, one with DBA authority and one without, so that they connect as DBA only when necessary.
- RESOURCE authority allows the user to create new database objects, such as tables, views, indexes, procedures, or triggers.

Granting permissions on tables and views

SQL Anywhere provides a set of permissions on individual tables and views. Users can be granted combinations of these permissions to define their access to a table or view.

Combinations of permissions

- The ALTER (permission to alter the structure of a table or to create triggers on a table) and REFERENCES (permission to create indexes on a table, and to create foreign keys) permissions grant the authority to modify the database schema., and so will not be assigned to most users. These permissions do not apply to views.
- The DELETE, INSERT, and UPDATE permissions grant the authority to modify the data in a table or view. Of these, the UPDATE permission may be restricted to a set of columns in the table or view.
- The SELECT permission grants authority to look at data in a table or view, but does not give permission to alter it.
- Granting ALL permissions grants all the above permissions.

Example 1

All table and view permissions are granted in a very similar fashion. You can grant permission to M_Haneef to delete rows from the table named sample_table as follows:

1. Connect to the database as a user with DBA authority, or as the owner of sample_table.
2. Type and execute the SQL statement:

```
GRANT DELETE
ON sample_table
TO M_Haneef
```

Example 2

You can grant permission to M_Haneef to update the column_1 and column_2 columns only in the table named sample_table as follows:

1. Connect to the database as a user with DBA authority, or as the owner of sample_table.
2. Type and execute the SQL statement: GRANT UPDATE column_1, column_2 ON sample_table TO M_Haneef

One limitation of table and view permissions is that they apply to all the data in a table or view (except for the UPDATE permission which may be restricted). Finer tuning of user permissions can be accomplished by creating procedures that carry out actions on tables, and then granting users the permission to execute the procedure.

Procedure permissions are discussed in "Granting permissions on procedures" on page 384.

Granting users the right to grant permissions

Each of the table and view permissions described in "Granting permissions on tables and views" on page 382 can be assigned WITH GRANT OPTION. This option gives the right to pass on the permission to other users. This feature is discussed in the context of groups in section "Permissions of groups" on page 388.

You can grant permission to M_Haneef to delete rows from the table named sample_table, and the right to pass on this permission to other users, as follows:

1 Connect to the database as a user with DBA authority, or as the owner of sample_table:

2 Type and execute the SQL statement:

```
GRANT DELETE ON sample_table
TO M_Haneef
WITH GRANT OPTION
```

Granting user permissions on tables in Sybase Central

One way to grant a user permissions on a table in Sybase Central is as follows:

1 Connect to the database.

2 Double-click the Tables folder for that database to display the tables in the left panel.

3 Click the Users and Groups folder, and locate the user you want to grant permissions to.

4 Drag the user to the table for which you want to grant permissions.

For more information, see the Sybase Central online Help.

Granting permissions on procedures

Permission to execute stored procedures may be granted by the DBA or by the owner of the procedure (the user ID that created the procedure).

The method for granting permissions to execute a procedure is similar to that for granting permissions on tables and views, discussed in "Granting permissions on tables and views" on page 382.

Example

You can grant M_Haneef permission to execute a procedure named my_procedure, as follows:

1. Connect to the database as a user with DBA authority or as owner of my_procedure procedure.
2. Execute the SQL statement:

```
GRANT EXECUTE
ON my_procedure
TO M_Haneef
```

Execution permissions of procedures

Procedures execute with the permissions of their owner. Any procedure that updates information on a table will execute successfully only if the owner of the procedure has UPDATE permissions on the table. As long as the procedure owner does have the proper permissions, the procedure will execute successfully when called by any user assigned permission to execute it, whether or not they have permissions on the underlying table. You can use procedures to allow users to carry out well-defined activities on a table, without having any general permissions on the table.

Granting user permissions on procedures in Sybase Central

One way to grant a user permissions on a table in Sybase Central is as follows:

1. Connect to the database.
2. Click the Users and Groups folder, and locate the user you want to grant permissions to.
3. Right-click the user, and select Copy from the popup menu.
4. Locate the procedure you want to allow the user to execute, in the Stored Procedures folder.
5. Click the procedure, and choose Edit➤Paste from the main menu to grant permissions.

For more information, see the Sybase Central online Help.

Execution permissions of triggers

Triggers are executed by the database engine in response to a user action; no permissions are required for triggers to be executed. When a trigger executes, it does have permissions associated with it that determine its ability to carry out actions.

Triggers execute with the permissions of the creator of the table with which they are associated.

For more information on trigger permissions, see "Trigger execution permissions" in the chapter "Using Procedures, Triggers, and Batches".

Revoking user permissions

The ability to revoke permissions is useful in order to take away permissions that have been explicitly granted to a user, but which are no longer appropriate.

More than this, any user's permissions are a combination of those that have been granted and those that have been revoked. Revoking and granting permissions work together in managing the pattern of user permissions on a database.

The REVOKE statement is the exact converse of the GRANT statement. To disallow M_Haneef from executing my_procedure, the command is:

```
REVOKE EXECUTE ON my_procedure FROM M_Haneef
```

This command must be issued by the DBA or by the owner of the procedure. Permission to delete rows from sample_table may be revoked by issuing the command:

```
REVOKE DELETE ON sample_table FROM M_Haneef
```

Managing groups

Once you understand how to manage permissions for individual users (as described in the previous section) working with groups is straightforward. A group is identified by a user ID, just like a single user, but this user ID is granted the permission to have members.

DBA, RESOURCE, and GROUP permissions

When permissions on tables, views, and procedures are granted to or revoked from a group, all members of the group inherit those changes. The DBA, RESOURCE, and GROUP permissions are not inherited: they must be assigned individually to each individual user ID requiring them.

A group is simply a user ID with special permissions. Granting permissions to a group and revoking permissions from a group are done in exactly the same manner as any other user, using the commands described in "Managing individual user IDs and permissions" on page 380.

A group can also be a member of a group. A hierarchy of groups may be constructed, each inheriting permissions from its parent group.

A user ID may be granted membership in more than one group, so the user-to-group relationship is many-to-many.

The ability to create a group without a password enables you to prevent anybody from signing on using the group user ID. This security feature is discussed in "Groups without passwords" on page 389.

Creating groups

❖ **To create a group with name personnel and password group_password:**

1. Connect to the database as a user with DBA authority.
2. Create the group's user ID just as you would any other user ID, using the following SQL statement:

```
GRANT CONNECT
TO personnel
IDENTIFIED BY group_password
```

3. Give the personnel user ID the permission to have members, with the following SQL statement:

```
GRANT GROUP TO personnel
```

The GROUP permission, which gives the user ID the ability to have members, is not inherited by members of a group. If this were not the case, then every user ID would automatically be a group as a consequence of membership in the special PUBLIC group.

Creating groups in Sybase Central

❖ **To create a group in Sybase Central:**

1 Connect to the database.

2 Click the Users and Groups folder for that database.

3 Double-click Add Group. A Wizard is displayed, which leads you through the process.

For more information, see the Sybase Central online Help.

Granting group membership to users

Making a user a member of a group is done with the GRANT statement. Membership in a group can be granted either by the DBA or by the group user ID. You can grant user M_Haneef membership in a group personnel as follows:

1 Connect to the database as a user with DBA authority, or as the group user ID personnel.

2 Grant membership in the group to M_Haneef with the following SQL statement:

```
GRANT MEMBERSHIP IN GROUP personnel
TO M_Haneef
```

When a user is assigned membership in a group, they inherit all the permissions on tables, views, and procedures associated with that group.

Adding users to groups in Sybase Central

One way to add a user to a group in Sybase Central is as follows:

1 Connect to the database.

2 Double-click the Users and Groups folder for that database to open it. Groups are displayed in the left panel, and both users and groups are displayed in the right panel.

3 In the right panel, select the users you want to add to a group, and drag them to the group.

For more information, see the Sybase Central online Help.

Permissions of groups

Permissions may be granted to groups in exactly the same way as to any other user ID, and permissions on tables, views, and procedures are inherited by members of the group, including other groups and their members. There are some complexities to group permissions that database administrators need to keep in mind.

Notes

The DBA, RESOURCE, and GROUP permissions are not inherited by the members of a group. Even if the personnel user ID is granted RESOURCE permissions, the members of personnel do not have RESOURCE permissions.

Ownership of database objects is associated with a single user ID and is not inherited by group members. If the user ID personnel creates a table, then the personnel user ID is the owner of that table and has the authority to make any changes to the table, as well as to grant privileges concerning the table to other users. Other user IDs who are members of personnel are not the owners of this table, and do not have these rights. If, however, SELECT authority is explicitly granted to the personnel user ID by the DBA or by the personnel user ID itself, all group members do have select access to the table. In other words, only granted permissions are inherited.

Referring to tables owned by groups

Groups are used for finding tables and procedures in the database. For example, the query

```
SELECT * FROM SYSGROUPS
```

will always find the table SYSGROUPS, because all users belong to the PUBLIC group and PUBLIC belongs to the SYS group which owns the SYSGROUPS table. (The SYSGROUPS table contains a list of group_name, member_name pairs representing the group memberships in your database.)

If a table employees is owned by the personnel user ID, and if M_Haneef is a member of the personnel group, then M_Haneef can refer to the employees table simply as employees in SQL statements. Users who are not members of the personnel group need to use the qualified name personnel.employees.

Creating a group to own the tables

It is advisable that you create a group whose only purpose is to own the tables. Do not grant any permissions to this group, but make all users members of the group. This allows everyone to access the tables without qualifying names. You can then create permission groups and grant users membership in these permission groups as warranted. For an example of this, see the section "Database object names and prefixes" on page 391.

Groups without passwords

Users connected to a group's user ID have certain permissions. This user ID can grant and revoke membership in the group. Also, this user would have ownership permissions over any tables in the database created in the name of the group's user ID.

It is possible to set up a database so that all handling of groups and their database objects is done by the DBA, rather than permitting other user IDs to make changes to group membership.

This is done by explicitly disallowing connection as the group's user ID when creating the group. To do this, the GRANT CONNECT statement is typed without a password. Thus:

```
GRANT CONNECT TO personnel
```

creates a user ID personnel. This user ID can be granted group permissions, and other user IDs can be granted membership in the group, inheriting any permissions that have been given to personnel, but nobody can connect to the database using the personnel user ID, because it has no valid password.

The user ID personnel can be an owner of database objects, even though no user can connect to the database using this user ID. The CREATE TABLE statement, CREATE PROCEDURE statement, and CREATE VIEW statement all allow the owner of the object to be specified as a user other than that executing the statement. This assignment of ownership can be carried out only by the DBA.

Special groups

When a database is created, two groups are also automatically created. These are SYS and PUBLIC. Neither of these groups has passwords, so it is not possible to connect to the database as either SYS or as PUBLIC. The two groups serve important functions in the database.

The SYS group

The SYS group is owner of the system tables and views for the database, which contain the full description of database structure, including all database objects and all user IDs.

For a description of the system tables and views, together with a description of access to the tables, see the chapters "SQL Anywhere System Tables" and "SQL Anywhere System Views".

The PUBLIC group

When a database is created, the PUBLIC group is automatically created, with CONNECT permissions to the database and SELECT permission on the system tables.

The PUBLIC group is a member of the SYS group, and has read access for some of the system tables and views, so that any user of the database can find out information about the database schema. If you wish to restrict this access, you can REVOKE PUBLIC's membership in the SYS group.

Any new user ID is automatically a member of the PUBLIC group and inherits any permissions specifically granted to that group by the DBA. You can also REVOKE membership in PUBLIC for users if you wish.

Database object names and prefixes

The name of every database object must be an identifier. The rules for valid identifiers are described in "Watcom-SQL language elements" in the chapter "Watcom-SQL Language Reference". In queries and sample SQL statements throughout this guide, database objects from the sample database are generally referred to using their simple name. For example:

```
SELECT *
FROM employee
```

Tables, procedures, and views all have an owner. The owner of the tables in the sample database is the user ID, DBA. In some circumstances you must prefix the object name with the owner user ID, as in the following statement.

```
SELECT *
FROM "DBA".employee
```

The employee table reference is said to be **qualified**. (In this case the owner name is enclosed in double quotes, as DBA is a SQL keyword.) In other circumstances it is sufficient to give the object name. This section describes when you need to use the owner prefix to identify tables, view and procedures, and when you do not.

When referring to a database object, a prefix is required unless:

- You are the owner of the database object.
- The database object is owned by a group ID of which you are a member.

Example

Consider the following example of a corporate database. All the tables are created by the user ID company. This user ID is used by the database administrator and is therefore given DBA authority.

```
GRANT CONNECT TO company
IDENTIFIED BY secret;
GRANT DBA TO company;
```

The tables in the database are created by the company user ID.

```
CONNECT USER company IDENTIFIED BY secret;
CREATE TABLE company.Customers ( ... );
CREATE TABLE company.Products ( ... );
CREATE TABLE company.Orders ( ... );
CREATE TABLE company.Invoices ( ... );
CREATE TABLE company.Employees ( ... );
CREATE TABLE company.Salaries ( ... );
```

Not everybody in the company should have access to all information. Consider two user IDs in the sales department, Joe and Sally, who should have access to the Customers, Products and Orders tables. To do this, you create a Sales group.

```
GRANT CONNECT TO Sally IDENTIFIED BY xxxxx;
GRANT CONNECT TO Joe IDENTIFIED BY xxxxx;
GRANT CONNECT TO Sales IDENTIFIED BY xxxxx;
GRANT GROUP TO Sales;
GRANT ALL ON Customers TO Sales;
GRANT ALL ON Orders TO Sales;
GRANT SELECT ON Products TO Sales;
GRANT MEMBERSHIP IN GROUP Sales TO Sally;
GRANT MEMBERSHIP IN GROUP Sales TO Joe;
```

Now Joe and Sally have permission to use these tables, but they still have to qualify their table references because the table owner is company, and Sally and Joe are not members of the company group:

```
SELECT *
FROM company.customers
```

To rectify the situation, make the Sales group a member of the company group.

```
GRANT GROUP TO company;
GRANT MEMBERSHIP IN GROUP company TO Sales;
```

Now Joe and Sally, being members of the Sales group, are indirectly members of the company group, and can reference their tables without qualifiers. The following command will now work:

```
SELECT *
FROM Customers
```

Note

Joe and Sally do not have any extra permissions because of their membership in the company group. The company group has not been explicitly granted any table permissions. (The company user ID has implicit permission to look at tables like Salaries because it created the tables and has DBA authority.) Thus, Joe and Sally still get an error executing either of these commands:

```
SELECT *
FROM Salaries;
```

```
SELECT *
FROM company.Salaries
```

In either case, Joe and Sally do not have permission to look at the Salaries table.

Using views and procedures for extra security

For databases that require a high level of security, defining permissions directly on tables has limitations. Any permission granted to a user on a table applies to the whole table. There are many cases when users' permissions need to be shaped more precisely than on a table-by-table basis. For example:

- It is not desirable to give access to personal or sensitive information stored in an employee table to users who need access to other parts of the table.
- You may wish to give sales representatives update permissions on a table containing descriptions of their sales calls, but limit such permissions to their own calls.

In these cases, you can use views and stored procedures to tailor permissions to suit the needs of your organization. This section describes some of the uses of views and procedures for permission management.

For information on how to create views, see "Working with views" in the chapter "Working with Database Objects".

Using views for tailored security

Views are computed tables that contain a selection of rows and columns from base tables. Views are useful for security when it is appropriate to give a user access to just one portion of a table. The portion can be defined in terms of rows or in terms of columns. For example, you may wish to disallow a group of users from seeing the salary column of an employee table, or you may wish to limit a user to see only the rows of a table that they have created.

Example

The Sales manager needs access to information in the database concerning employees in the department. However, there is no reason for the manager to have access to information about employees in other departments.

This example describes how to create a user ID for the sales manager, create views that provides the information she needs, and grants the appropriate permissions to the sales manager user ID.

The first example gives the sales manager permission to see the employees in the Sales department by defining a view on the employee table.

1 Create the new user ID using the GRANT command, from a user ID with DBA authority. Enter the following:

```
CONNECT "dba" IDENTIFIED by sql ;
GRANT CONNECT TO SalesManager IDENTIFIED BY sales
```

(You must enclose DBA in quotation marks because it is a SQL keyword, just like SELECT and FROM.)

2 Define a view which only looks at sales employees as follows:

```
CREATE VIEW emp_sales AS
SELECT emp_id, emp_fname, emp_lname
FROM "dba".employee
WHERE dept_id = 200
```

The table should be identified as "dba".employee, with the owner of the table explicitly identified, for the SalesManager user ID to be able to use the view.

3 Give SalesManager permission to look at the view: GRANT SELECT ON emp_sales TO SalesManager Exactly the same command is used to grant permission on a view as to grant permission on a table.

Example 2

The next example creates a view which allows the Sales Manager to look at a summary of sales orders. This view requires information from more than one table for its definition:

1 Create the view.

```
CREATE VIEW order_summary AS
SELECT order_date, region, sales_rep,
company_name
FROM "dba".sales_order
   KEY JOIN "dba".customer
```

2 Grant permission for the Sales Manager to examine this view.

```
GRANT SELECT
ON order_summary
TO SalesManager
```

3 To check that the process has worked properly, connect to the SalesManager user ID and look at the views you have created:

```
CONNECT SalesManager IDENTIFIED BY sales ;
SELECT * FROM "dba".emp_sales ;
SELECT * FROM "dba".order_summary ;
```

No permissions have been granted to the Sales Manager to look at the underlying tables. The following commands produce permission errors.

```
SELECT * FROM "dba".employee ;
SELECT * FROM "dba".sales_order
```

Update permissions on views

The example shows how to use views to tailor SELECT permissions. INSERT, DELETE, and UPDATE permissions can also be granted on views.

For information on allowing data modification on views, see "Using views" in the chapter "Working with Database Objects".

Using procedures for tailored security

While views restrict access on the basis of data, procedures restrict the actions a user may take. As described in "Granting permissions on procedures" on page 384, a user may have EXECUTE permission on a procedure without having any permissions on the table or tables on which the procedure acts.

Strict security

For strict security, you can disallow all access to the underlying tables, and grant permissions to users or groups of users to execute certain stored procedures. With this approach, the manner in which data in the database can be modified is strictly defined.

Stored procedures
Stored procedures are not supported by the SQL Anywhere Desktop Runtime system.

How SQL Anywhere assesses user permissions

Groups do introduce complexities in the permissions of individual users. Suppose user M_Haneef has been granted select and update permissions on a specific table individually, but is also a member of two groups, one of which has no access to the table at all, and one of which has only select access. What are the permissions in effect for this user?

SQL Anywhere decides whether a user ID has permission to carry out a specific action in the following manner:

1. SQL Anywhere looks to see if the user ID has DBA permissions: if so, the user ID can carry out any action in the database.
2. If the user ID does not have DBA permissions, SQL Anywhere looks at the permissions assigned to the individual user. If the user ID has been granted permission to carry out the action, then the action is allowed to proceed.
3. If no individual settings have been made for that user, SQL Anywhere looks at the permissions of each of the groups of which the user is a member. If any of these groups has permission to carry out the action, then the user ID has permission by virtue of membership in that group, and the action is allowed to proceed.

This approach minimizes problems associated with the order in which permissions are set.

Users and permissions in the system tables

Information about the current users of a database and about their permissions is stored in the database system tables and system views.

For a description of each of these tables, see "SQL Anywhere System Tables".

The system tables are owned by the special user ID SYS. It is not possible to connect to the SYS user ID.

The DBA has SELECT access to all system tables, just as to any other tables in the database. The access of other users to some of the tables is limited. For example, only the DBA has access to the SYS.SYSUSERPERM table, which contains all information about the permissions of users of the database, as well as the passwords of each user ID. However, SYS.SYSUSERPERMS is a view containing all information in SYS.SYSUSERPERM except for the password, and by default all users have SELECT access to this view. All permissions and group memberships set up in a new database for SYS, PUBLIC, and DBA can be fully modified.

The following table summarizes the system tables containing information about user IDs, groups, and permissions. All tables and views are owned by user ID SYS, and so their qualified names are SYS.SYSUSERPERM and so on.

Appropriate SELECT queries on these tables generates all the user ID and permission information stored in the database.

Table	Default	Contents
SYSUSERPERM	DBA only	Database-level permissions and password for each user ID
SYSGROUP	PUBLIC	One row for each member of each group
SYSTABLEPERM	PUBLIC	All permissions on table given by the GRANT command s
SYSCOLPERM	PUBLIC	All columns with UPDATE permission given by the GRANT command
SYSDUMMY	PUBLIC	Dummy table, can be used to find the current user ID
SYSPROCPERM	PUBLIC	Each row holds one user granted permission to use one procedure

The following table summarizes the system views containing information about user IDs, groups, and permissions

Views	Default	Contents
SYSUSERAUTH	DBA only	All information in SYSUSERPERM except for user numbers
SYSYUSERPERMS	PUBLIC	All information in SYSUSERPERM except for passwords
SYSUSERLIST	PUBLIC	All information in SYSUSERAUTH except for passwords
SYSGROUPS	PUBLIC	Information from SYSGROUP in a more readable format
SYSTABAUTH	PUBLIC	Information from SYSTABLEPERM in a more readable format
SYSCOLAUTH	PUBLIC	Information from SYSCOLPERM in a more readable format
SYSPROCAUTH	PUBLIC	Information from SYSPROCPERM in a more readable format

In addition to these tables and views, there are tables and views containing information about each of the objects of the database that include the owner of each object.

CHAPTER 25

Backup and Data Recovery

About this chapter

This chapter explains how to use SQL Anywhere transaction log files to protect your data, how to make backup copies of your database files and the SQL Anywhere logs, and the recovery procedures for system and media failures.

Contents

Topic	Page
System and media failures	400
The SQL Anywhere logs	401
Using a transaction log mirror	406
Backing up your database	411
Recovery from system failure	414
Recovery from media failure	416

System and media failures

SQL Anywhere has features to protect your data from two categories of computer failure: **system failure** and **media failure**.

System Failure A system failure occurs when a power failure or some other failure causes the computer or operating system to go down while there are partially completed transactions. This could occur when the computer is inappropriately turned off or rebooted, or when another application causes the operating system to crash.

Media Failure A media failure occurs when the database file, the file system, or the device storing the database file becomes unusable.

Recovery from failure

When failures occur, the SQL Anywhere recovery mechanism treats transactions properly, as **atomic** units of work: any incomplete transaction is rolled back and any committed transaction is preserved. This ensures that even in the event of failure, the data in your database remains in a consistent state.

SQL Anywhere uses three logs to protect your data from system. These log files exist for each database running on a database engine or server.

Make regular backups

You should make regular backups of your database files so that you can recover your database in the case of a media failure. SQL Anywhere uses the transaction log (which you should store on a separate device from the database for greater security) to recover information put into the database since the last full backup.

The SQL Anywhere Desktop Runtime database engine does not support a transaction log, so the work carried out between the time of the last backup and the time of the media failure is lost, and you will have to re-enter it into the database.

The SQL Anywhere logs

SQL Anywhere uses three logs to protect your data from system and media failure.

All these logs play a role in data recovery. Each log exists for each database running on a database engine or server. Optionally, you can maintain a mirror of the transaction log for greater protection of vital data.

The checkpoint log

Checkpoint log purpose

The checkpoint log is used during database recovery after a system failure or improper shutdown of the database engine.

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.

Dirty pages are not written immediately to the disk. For improved performance, they are cached in memory and written to disk when the cache is full or the server has no pending requests. A **checkpoint** is a point at which all dirty pages are written to disk. Once all dirty pages are written to disk, the checkpoint log is deleted.

Reasons for a checkpoint

A checkpoint can occur for several reasons:

- The database engine is shut down
- The amount of time since the last checkpoint exceeds the database option CHECKPOINT_TIME
- The estimated time to do a recovery operation exceeds the database option RECOVERY_TIME
- The database engine is idle long enough to write all dirty pages
- A transaction issues a CHECKPOINT statement
- The database engine is running without a transaction log and a transaction is committed

Checkpoint priority

The priority of writing dirty pages to the disk increases as the time and the amount of work since the last checkpoint grows. The priority is determined by the following factors:

- **Checkpoint Urgency** The time that has elapsed since the last checkpoint, as a percentage of the CHECKPOINT_TIME setting of the database. The database option CHECKPOINT_TIME controls the maximum desired time, in minutes, between checkpoints.
- **Recovery Urgency** A heuristic to estimate the amount of time required to recover the database if it fails right now. The database option RECOVERY_TIME controls the maximum desired time, in minutes, for recovery in the event of system failure.

The checkpoint and recovery urgencies are important when the database engine does not have enough idle time to write dirty pages.

When the database server is running with multiple databases, the CHECKPOINT_TIME and RECOVERY_TIME specified by the first database started is used, unless overridden by command line switches.

For a description of the command-line options, see "The database engine" in the chapter "SQL Anywhere Components".

How the database decides when to checkpoint

Optional information
You do not need to know the information in this section for most purposes. It is provided as background information for those who wish to understand more about how the database engine works.

The idle I/O task

The writing of dirty pages to disk is carried out by a task within the database engine called the idle I/O task. This task shares processing time with other database tasks, according to a priority. The lower the priority of the idle I/O task, the less time it gets.

There is a threshold for the number of dirty pages below which writing of database pages does not take place.

When the database is busy, the urgency is low, and the cache only has a few dirty pages, the idle I/O task runs at a very low priority and no writing of dirty pages takes place.

Once the urgency exceeds 30%, the priority of the idle I/O task is increased. At intervals, the priority is increased again. As the urgency becomes high, the engine shifts its primary focus to writing dirty pages until the number gets below the threshold again. However, the engine only writes out pages during the idle I/O task if the number of dirty pages is greater than the threshold.

If, because of other activity in the database, the number of dirty pages falls to zero, and if the urgency is 50% or more, then a checkpoint takes place automatically, since it is a convenient time.

Both the checkpoint urgency and recovery urgency values increase in value until the checkpoint occurs, at which point they drop to zero. They do not decrease otherwise.

The rollback log

As changes are made to the contents of tables, a **rollback log** is kept for the purpose of canceling changes. It is used to process the ROLLBACK statement for recovering from system failure. There is a separate rollback log for each transaction. When a transaction is complete, its rollback log is deleted.

The transaction log

All changes to the database are stored in the transaction log in the order that they occur. Inserts, updates, deletes, commits, rollbacks, and database schema changes are all logged. The transaction log is called a **forward log file**.

What you should do

The transaction log is optional. If you run SQL Anywhere with no transaction log, a checkpoint is carried out whenever a transaction is committed. The checkpoint ensures that all committed transactions are written to the disk. Checkpoints can be time consuming, so you should run with a transaction log for improved performance as well as protection against media failure and corrupted databases.

For greater protection, SQL Anywhere allows you to maintain two identical transaction logs in tandem. This is called transaction log mirroring.

For information on creating a database with a mirrored transaction log, see "The Initialization utility" in the chapter "SQL Anywhere Components". For information on changing an existing database to use a mirrored transaction log, see "The Transaction Log utility" in the chapter "SQL Anywhere Components".

Keep the transaction log on a separate device

The transaction log is not kept in the main database file. The filename of the transaction log can be set when the database is initialized (with DBINIT) or at any other time (with DBLOG) when the database engine is not running. To protect against media failure, the transaction log should be written to a different device than the database file. Some machines with two or more hard drives only have one physical disk drive with several logical drives or partitions. If you want protection against media failure, make sure that you have a machine with two storage devices or use a storage device on a network file server. Note that by default, the transaction log is put on the same device and in the same directory as the database—this does not protect against media failure.

Primary key definitions affect transaction log size

Updates and deletes on tables that do not have a primary key or unique index will cause the entire contents of the rows affected to be logged in the transaction log. If a primary key is defined, the engine needs only to record the primary key column values to uniquely identify a row. If the table contains many columns or wide columns, the transaction log pages will fill up much faster (reducing performance) if no primary key is defined. And if DBTRAN is used on the transaction log, it produces a very large command file.

This affects UPDATEs and DELETEs but not INSERTs, which must always log all column values.

If a primary key does not exist, the engine will look for a UNIQUE NOT NULL index on the table (or a UNIQUE constraint). A UNIQUE index that allows null values is not sufficient.

> **Performance tip**
> Placing the transaction log on a separate device can also result in improved performance by eliminating the need for disk head movement between the transaction log and the main database file.

Converting transaction logs to SQL

The transaction log is not human-readable. The command-line utility DBTRAN can be used to convert a transaction log into a SQL command file, which can serve as an audit trail of changes made to the database. The following command uses DBTRAN to convert a transaction log:

```
dbtran sample.log changes.sql
```

You can also convert a transaction log to a SQL command file from Sybase Central or from ISQL.

For more information on the log translation utility, see "The Log Translation utility" in the chapter "SQL Anywhere Components".

Recovering uncommitted database changes

The transaction log contains a record of everything, including transactions that were never committed. By converting the transaction log to a SQL command file using the log translation utility and choosing to include uncommitted transactions (for example by running the DBTRAN command-line utility with the -a switch) you can recover transactions that were accidentally canceled by a user. (If this option is not chosen, the log translation utility omits transactions that were rolled back.) While this is not a common procedure, it can prove useful for exceptional cases.

Using a transaction log mirror

A transaction log mirror is an identical copy of the transaction log, maintained at the same time as the transaction log. Transaction log mirrors are used to allow complete recovery in the case of media failure on the log device.

Every time a database change is written to the transaction log, it is also written to the transaction log mirror file. By default, SQL Anywhere does not use a mirrored transaction log, but you can choose to use one when creating a database or you can make an existing database use a mirrored transaction log.

Why use a transaction log mirror?

A mirrored transaction log provides extra protection of critical data. For example, at a consolidated database in a SQL Remote setup, replication relies on the transaction log, and if the transaction log is damaged or becomes corrupt, data replication can fail.

There is a performance penalty for using a mirrored log, as each database log write operation must be carried out twice. The performance penalty depends on the nature and volume of database traffic and on the physical configuration of the database and logs.

If SQL Anywhere is using a mirrored transaction log, and gets an error while trying to write to one of them (for example, if the disk is full), the database engine or server stops. The purpose of a transaction log mirror is to ensure complete recoverability in the case of media failure on either log device; this purpose would be lost if SQL Anywhere continued with a single log.

Where to use a transaction log mirror

A transaction log mirror should be kept on a separate device from the transaction log, so that if either device fails, the other copy of the log keeps the data safe for recovery.

Requirements

You can only use a mirrored transaction log on a database created with SQL Anywhere Release 5.0 or later.

Creating and dropping a transaction log mirror

Transaction log mirrors can be created at the following times:

- When you create a database, using the initialization utility
- At any other time that the database is not running, using the transaction log utility.

- A mirror for a write file transaction log can be created along with the write file using the write file utility (mirroring can be added later, using the transaction log utility)

Notes

- You cannot choose to use a transaction log mirror without using a transaction log
- The default file extension for transaction log mirrors is .MLG.

Creating a database with a transaction log mirror

You can choose to maintain a transaction log mirror when you create a database. This option is available either from Sybase Central or from the DBINIT command-line utility.

From Sybase Central, the transaction log mirror option is part of the Create Database utility.

For more information, see the Sybase Central online Help.

The following command line (which should be entered on one line) initializes a database named company.db, with a transaction log kept on a different device and a mirror on a different device still.

```
dbinit -t d:\log_dir\company.log -m
e:\mirr_dir\company.mlg c:\db_dir\company.db
```

By default, a transaction log is used but no transaction log mirror is created. For a full description of DBINIT command-line options, see "Initialization utility options" in the chapter "SQL Anywhere Components".

Starting a transaction log mirror for an existing database

You can choose to maintain a transaction log mirror for an existing database any time the database is not running by using the transaction log utility. This option is available from either Sybase Central or the DBLOG command-line utility.

From Sybase Central, the transaction log mirror option is part of the Change Log File utility.

For more information, see the Sybase Central online Help.

The following command line starts a transaction log mirror for a database named company.db, which is already using one transaction log.

```
dblog -m e:\mirr_dir\company.mlg
c:\db_dir\company.db
```

The following command line stops the company.db database from using a transaction log mirror, but continues maintaining a transaction log:

```
dblog -r c:\db_dir\company.db
```

The following command line stops the company.db database from using a transaction log mirror or a transaction log:

```
dblog -n c:\db_dir\company.db
```

With the transaction log utility you can also alter the name or directory of the transaction log and mirror. For a full description of DBLOG command-line options, see "Transaction log utility options" in the chapter "SQL Anywhere Components".

Starting a transaction log mirror for a write file

You can choose to maintain a transaction log mirror for a write file when you create the write file using the write file utility, or at a later time using the transaction log utility.

The option to create a transaction log mirror when creating a write file is available from Sybase Central or from the DBWRITE command-line utility.

From Sybase Central, the transaction log mirror option is part of the Create Write File utility.

For more information, see the Sybase Central online Help.

The following command line (which should be entered on one line) creates a write file for a database named company.db, which is already using a transaction log. The write file has default extension .WRT, the write file transaction log has the default extension .WLG, and the write file transaction log mirror has the default extension .WML.

```
dbwrite -c -t d:\log_dir\company.wlg -m
e:\mirr_dir\company.wml c:\db_dir\company.db
c:\db_dir\company.wrt
```

For a full description of DBWRITE command-line options, see "Write file utility options" in the chapter "SQL Anywhere Components".

You can change the transaction log and log mirror settings of a write file using the transaction log utility, in exactly the same way as described above for a standard database file.

Erasing transaction log mirrors

You can erase transaction log mirrors using the Erase utility in Sybase Central or the DBERASE command-line utility.

The Erase utility is available in Sybase Central as the Erase Database utility or from the command line as the DBERASE utility.

❖ **To delete a mirror log file only:**

- Enter the following command:

```
dberase e:\mirr_dir\company.wml
```

❖ **To delete a transaction log file but not its mirror:**

- Enter the following command:

```
dberase e:\log_dir\company.log
```

Validating the transaction log on database startup

When a database that is using a mirror starts up, the database engine or server carries out a series of checks and automatic recovery operations to confirm that the transaction log and its mirror are not corrupted, and to correct some problems if corruption is detected.

On startup, the database engine checks that the transaction log and its mirror are identical by carrying out a full comparison of the two files; if they are identical, the database starts as usual. The comparison of log and mirror adds to database startup time when you are maintaining a log mirror.

If the database stopped because of a system failure, it is possible that some operations were written into the transaction log but not into the mirror. If the database engine finds that the transaction log and the mirror are identical up to the end of the shorter of the two files, then the remainder of the longer file is copied over into the shorter file to produce identical log and mirror. After this automatic recovery step, the database engine starts as usual.

If the check finds that the log and the mirror are different in the body of the shorter of the two, one of the two files is corrupt. In this case, the database does not start, and an error message is generated saying that the transaction log or its mirror is invalid.

Recovering from a corrupt transaction log or mirror

When a database engine or server detects a difference between the transaction log and its mirror in the body of the file, the engine does not start. You must take the following steps before starting the engine:

1 Identify which of the two files is corrupt.

2 Copy the correct file over the corrupt file so that you have two identical files again.

3 Restart the database engine.

When a database engine detects a difference between the transaction log and its mirror, it has no means of knowing which is intact and which is corrupt.

❖ **To identify which file is corrupt using the database utilities:**

1. Make a copy of the backup of your database file taken at the time the transaction log was started.
2. Run the log translation utility on the transaction log and on its mirror, to see which one generates an error message. (The log translation utility is accessible from Sybase Central or as the DBTRAN command-line utility.)
3. The following command-line translates a transaction log named SADEMO.LOG, placing the translated output into SADEMO.SQL:

   ```
   dbtran sademo.log
   ```

   The translation utility properly translates the intact file, and will report an error while translating the corrupt file.
4. If the DBTRAN test does not identify the incorrect log, you may want to compare the two translated logs (SQL files) to see which one contains an error, or use a disk utility to inspect the two files and detect which is corrupt.
5. Once you have identified the corrupt file, you can copy the intact log file over the corrupt file, and restart the production database engine.

Backing up your database

A **full backup** makes a copy of the database file and (optionally) a copy of the transaction log. Full backups are described in "Performing a full backup" below.

An **incremental backup** makes a copy of the transaction log.

Both full and incremental backups can be carried out online or offline. You can use any means of backing up the files onto diskette, magnetic tape, optical disk, or any other device. Incremental backups are described in "Performing an incremental backup" on page 413.

Online backups

Backups can be made without stopping the database engine. The backup utility can be run against a standalone engine or network database server. Using the backup utility on a running database is equivalent to copying the database files when the database is not running. In other words, it provides a snapshot of a consistent database, even while it is being modified by other users.

For a full description of the online backup facility, see "The Backup utility" in the chapter "SQL Anywhere Components".

Offline backups

The database engine should not be running when you do offline backups by copying database files. Moreover, it should be taken down cleanly.

If you are running a multiuser database, you can use the database server -t command line option to shut down at a specified time. This way, you can have your offline backup procedure start late at night automatically.

Performing a full backup

Check the validity of the database

Before doing a full backup, it is a good idea to verify that the database file is not corrupt. File system errors or software errors (bugs) in any software you are running on your machine could corrupt a small portion of the database file without you ever knowing.

With the database engine running on the database you want to check, execute the validation utility that comes with SQL Anywhere. For example, you could run the DBVALID command-line utility:

```
dbvalid -c "uid=dba;pwd=sql"
```

You can also run the validation utility from Sybase Central or ISQL.

The validation utility scans every record in every table and looks up each record in each index on the table. If the database file is corrupt, you need to recover from your previous backup.

For more information on running the validation utility, see "The Validation utility" in the chapter "SQL Anywhere Components".

Back up the database files

A full backup is completed offline by copying the database file(s) and optionally the transaction log to the backup media. A full backup should be completed according to a regular schedule that you follow carefully. Once a week works well for most situations.

To do a backup while the database engine is running, you use the backup utility. You require DBA authority in order to run the backup utility on a database. The backup utility can be run from Sybase Central, ISQL, or using the DBBACKUP command-line utility.

For more information, see "The Backup utility" in the chapter "SQL Anywhere Components".

For example, you could carry out a full backup of the sample database, held in C:\SQLANY50\SADEMO.DB, to a directory E:\BACKUP, using user ID *dba*, and password *sql*.

❖ **To complete a full backup:**

- Enter the following command line:

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

  As neither of -d or -t is specified, both the database files and transaction log are backed up.

Transaction log options

Whenever the database file is backed up, the transaction log can be archived and/or deleted (use the Erase utility). Provided the backup can be restored, you will never need the transaction log. Archiving transaction logs provides you with a history of all changes to your database and also provides protection if you are unable to restore the most recent full backup. The backup utility has command line options to delete and restart the transaction log (DBBACKUP -x) or backup and restart the transaction log (DBBACKUP -r) while the database engine is running.

Keep several full backups

You should keep several previous full backups. If you back up on top of the previous backup, and you get a media failure in the middle of the backup, you are left with no backup at all. You should also keep some of your full backups offsite to protect against fire, flood, earthquake, theft, or vandalism.

If your transaction log tends to grow to an unmanageable size between full backups, you should consider getting a larger storage device or doing full backups more frequently.

Performing an incremental backup

An incremental backup is a copy of the transaction log. The transaction log has all changes since the most recent full backup.

You can carry out an offline incremental backup by making a copy of the transaction log file. Alternatively, you can carry out an online incremental backup by running the backup utility and backing up just the transaction log. You can do this from the command line using the DBBACKUP utility with the -t switch, or you can use the backup utility from Sybase Central or ISQL. You require DBA authority in order to run the backup utility on a database file.

For example, you could carry out an incremental backup of the sample database, held in C:\SQLANY50\SADEMO\DB, to a directory E:\BACKUP, with user ID *dba*, and password *sql*.

❖ **To complete an incremental backup:**

- Enter the following command line:

```
dbbackup -c "uid=dba;pwd=sql;
dbf=c:\sqlany50\sademo.db" -t e:\backup
```

Daily backups of the transaction log

You should back up the transaction log daily. This is particularly important if you have the transaction log on the same device as the database file. If you get a media failure, you could lose both files. By doing daily backups of the transaction log, you will never lose more than one day of changes.

Daily backups of the transaction log are also recommended when the transaction log tends to grow to an unmanageable size between full backups and you do not want to get a larger storage device or do more frequent full backups. In this case, you can choose to archive and delete the transaction log.

There is a drawback to deleting the transaction log after a daily backup. If you have media failure on the database file, there will be several transaction logs since the last full backup. Each of the transaction logs needs to be applied in sequence to bring the database up to date.

For a description of how to do this, see "Media failure on the database file" on page 416.

Recovery from system failure

After a power failure or other system failure you should run the system disk verification program:

- For OS/2: Run scandisk with the /autofix option.
- For NetWare: Load the Novell VREPAIR NLM to repair any volume that will not mount due to errors.
- For QNX: Use *chkfsys*.
- For DOS or Windows NT, run the DOS or NT console command:

  ```
  chkdsk /f
  ```

  This command fixes simple errors in the file system structure that might have been caused by the system failure. This should be done before running any software, including Windows if applicable.

This fixes up simple errors in the file system structure that might have been caused by the system failure. This should be done before running any other software.

After a system error occurs, SQL Anywhere automatically recovers when you restart the database. The results of each transaction committed before the system error are intact. All changes by transactions that were not committed before the system failure are canceled. It is possible to recover uncommitted changes manually (see "Recovering uncommitted changes" on page 418).

Steps to recover from a system failure

The database engine automatically takes three steps to recover from a system failure:

1. Restore all pages to the most recent checkpoint, using the checkpoint log.
2. Apply any changes made between the checkpoint and the system failure. These changes are in the transaction log.
3. Rollback all uncommitted transactions, using the rollback logs. There is a separate rollback log for every connection.

Frequent checkpoints make recovery from system failure take less time but also create work for the database engine writing out dirty pages.

There are two database options that allow you to control the frequency of checkpoints. CHECKPOINT_TIME will control the maximum desired time between checkpoints and RECOVERY_TIME will control the maximum desired time for recovery in the event of system failure (see "SET OPTION statement" in the chapter "Watcom-SQL Statements"). The RECOVERY_TIME specifies an estimate for steps 1 and 2 only.

Step 3 may take a long time if there are long uncommitted transactions that have already done a great deal of work before the last checkpoint.

The transaction log is optional. When you are running SQL Anywhere with no transaction log, a checkpoint is done whenever any transaction is committed. In the event of system failure, the database engine uses steps 1 and 3 from above to recover a database file. Step 2 is not necessary because there will be no committed transactions since the last checkpoint. This is, however, usually a slower way to run because of the frequent checkpoints.

Runtime engine is an exception
The SQL Anywhere Runtime System does not employ a transaction log.

Quick recovery with a live backup

You can carry out a live backup of the transaction log by using the DBBACKUP command line utility with the -l command-line option.

Live backups provide a redundant copy of the transaction log that is available for restart of your system in case of system failure. A live backup runs continuously, terminating only if the server shuts down. If you suffer a system failure, the backed up transaction log can be used for a rapid restart of the system.

Recovery from media failure

If you have backups, you can always recover all transactions that were committed before the media failure. Recovery from media failure requires you to keep the transaction log on a separate device from the database file. The information in the two files is redundant. Regular backups of the database file and the transaction log reduce the time required to recover from media failures.

The first step in recovering from a media failure is to clean up, reformat, or replace the device that failed.

The steps to take in recovery depend on whether the media failure is on the device holding your database file (see "Media failure on the database file" next) or on the device holding your transaction log (see "Media failure on the transaction log" on page 417).

Media failure on the database file

When your transaction log is still usable, but you have lost your database file, the recovery process depends on whether you keep or delete the transaction log on incremental backup.

If you have a single transaction log

If you have not deleted or restarted the transaction log since the last full backup, the transaction log contains everything since the last backup. Recovery involves four steps:

1. Make a backup of the transaction log immediately. The database file is gone and the only record of the changes is in the transaction log.
2. Restore the most recent full backup (the database file).
3. Use the database engine with the apply transaction log (*-a*) switch, to apply the transaction log and bring the database up to date:

   ```
   DBENG50 sademo.db -a sademo.log
   ```

4. Start the database in the normal way. The database engine will come up normally and any new activity will be appended to the current transaction log.

If you have multiple transaction logs

If you have archived and deleted the transaction log since the last full backup, each transaction log since the full backup needs to be applied in sequence to bring the database up to date.

1. Make a backup of all transaction logs immediately. The database file is gone and the only record of the changes is in the transaction logs.

2 Restore the most recent full backup (the database file).

3 Starting with the first transaction log after the full backup, apply each archived transaction log by starting the database engine with the apply transaction log (-a) switch. For example, if the last full backup was on Sunday and the database file is lost during the day on Thursday.

```
DBENG50W sademo.db -a mon.log
DBENG50W sademo.db -a tue.log
DBENG50W sademo.db -a wed.log
DBENG50W sademo.db -a sample.log
```

4 SQL Anywhere does not allow you to apply the transaction logs in the wrong order or to skip a transaction log in the sequence.

5 Start the database in the normal way. The database engine will come up normally and any new activity will be appended to the current transaction log.

Media failure on the transaction log

When your database file is still usable but you have lost your transaction log, the recovery process is as follows:

1 Make a backup of the database file immediately. The transaction log is gone and the only record of the changes is in the database file.

2 Restart the database with the -f switch.

```
DBENG50 sademo.db -f
```

Without the switch, the database engine will complain about the lack of a transaction log. With the switch, the database engine will restore the database to the most recent checkpoint and then roll back any transactions that were not committed at the time of the checkpoint. A new transaction log will be created.

Consequences of media failure on the transaction log

Media failure on the transaction log can have more serious consequences than media failure on the database file. When you lose the transaction log, all changes since the last checkpoint are lost. This will be a problem when you have a system failure and a media failure at the same time (such as if a power failure causes a head crash that damages the disk). Frequent checkpoints minimize the potential for lost data, but also create work for the database engine writing out dirty pages.

For running high-volume or extremely critical applications, you can protect against media failure on the transaction log by mirroring the transaction log or by using a special-purpose device, such as a storage device that mirrors the transaction log automatically. If you are using the SQL Anywhere Server for NetWare, NetWare allows you to automatically mirror a NetWare volume.

For information on using a transaction log mirror, see "Using a transaction log mirror" on page 406.

Recovering uncommitted changes

The transaction log keeps a record of all changes made to the database. Even uncommitted changes are stored in the transaction log. The DBTRAN utility has a command line option *(-a)* to translate transactions that were not committed. With this option, you can recover changes that were not committed by editing the SQL command file and picking out changes that you want to recover.

Changes not present in the transaction log
The transaction log may contain changes right up to the point where a failure occurred. It will contain any changes that were made before the most recent COMMIT by *any* transaction.

CHAPTER 26

Introduction to SQL Remote Replication

About this chapter

SQL Remote is a message-based replication system for two-way server-to-laptop, server-to-desktop, and server-to-server data replication between SQL Anywhere databases.

This chapter introduces SQL Remote and guides you through setting up a simple replication system.

Contents

Topic	Page
Introduction to data replication	420
SQL Remote concepts	422
SQL Remote features	427
Message systems supported by SQL Remote	429
Tutorial: setting up SQL Remote using Sybase Central	430
Set up the consolidated database in Sybase Central	433
Set up the remote database in Sybase Central	438
Tutorial: setting up SQL Remote using ISQL and DBXTRACT	439
Set up the consolidated database	442
Set up the remote database	445
Start replicating data	447
A sample publication	450
Some sample SQL Remote setups	451

Introduction to data replication

Data replication is the sharing of data among physically distinct databases. Changes made to shared data at any one database are replicated to the other databases in the replication setup. The SQL Remote data replication system enables replication of data among SQL Anywhere databases.

Replication and data availability

One of the key benefits of a data replication system is that data is made available locally, rather than through potentially expensive, less reliable, and slow connections to a single central database. Data is accessible locally even in the absence of any connection to a central server, so that you are not cut off from data in the event of a failure of a long-distance network connection.

Replication and performance

Replication improves response times for data requests for two reasons. Requests are processed on a local server without accessing some wide area network, so that transfer rates are faster. Also, local processing offloads work from a central database server so that competition for processor time is decreased.

Replication and integrity

One of the challenges of any replication system is to ensure that each database retains data integrity at all times. Today's replication systems, such as Sybase Replication Server and SQL Remote, replicate portions of the transaction log in such a way that transactions are replicated atomically: either a whole transaction is replicated, or none of it is replicated. This ensures data integrity at each database in the setup.

Replication and data consistency

Another challenge to replication systems is to maintain data consistency throughout the setup. SQL Anywhere and other replication systems maintain a **loose consistency** in the setup as a whole: that is, all changes are replicated to each site over time in a consistent manner, but because of the time lag different sites may have different copies of data at any instant.

Replication with SQL Anywhere

SQL Anywhere supports two forms of replication. SQL Anywhere databases can take part in Sybase Replication Server setups. Replication Server is a powerful, high-performance, technology for maintaining replicated data at multiple sites on a network. Replication Server permits replication among heterogeneous databases and database management systems. Replication Server can replicate data to SQL Anywhere databases using the Open Server Gateway that is part of this package. To replicate data from SQL Anywhere databases to other databases using Replication Server, you need a SQL Anywhere Replication Agent, which is available as a separate product.

SQL Remote is a system for data replication between SQL Anywhere databases only. SQL Remote is less flexible and configurable than Replication Server; for example, it has strict requirements for object names and attributes to be identical at different databases in the replication setup. Rather, SQL Remote is designed to be easy to administer, suitable for widespread deployment, capable of supporting replication to many replicate sites from one database, and is designed to replicate data even between databases not directly connected to each other. The following sections describe SQL Remote in more detail.

SQL Remote concepts

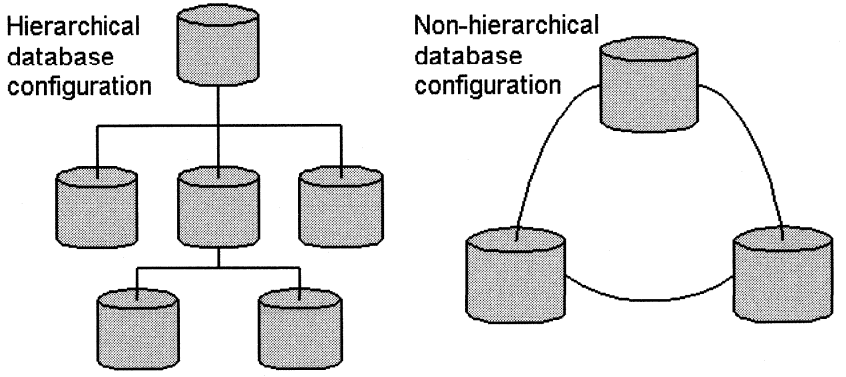
This section introduces concepts and terms used throughout the chapter.

Consolidated and remote databases

SQL Remote provides data replication between a **consolidated database** and a **remote database** or databases. Remote databases are SQL Anywhere servers or standalone database engines that may be running at the same site as the consolidated database or at different sites. Stated another way, SQL Remote supports hierarchical configurations of databases; it does not support peer-to-peer replication or other non-hierarchical configurations.

The consolidated database contains all the data to be replicated. Remote databases may contain all the data or may contain just some of the data. Replication is two-way: changes made at the consolidated database are replicated to remote databases, while changes made at remote databases are replicated to the consolidated database, and thence to other remote databases.

With a hierarchy constructed in this way, each database contains all or a subset of the data replicated by the node above it in the hierarchy.



Remote databases can contain tables that are not present at the consolidated database, as long as they are not involved in replication.

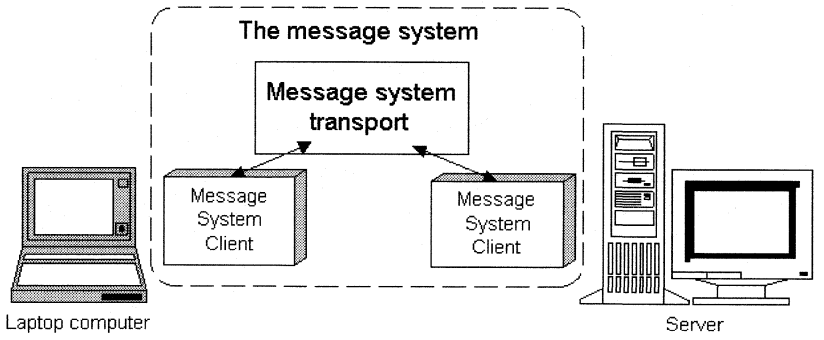
Message-based replication

SQL Remote exchanges data between databases using **messages**. This allows replication between databases that have no direct connection: an occasional message-based connection such as e-mail is sufficient.

In message-based communications, each message carries its destination address and other control information, so that no direct connection is needed between applications exchanging information. E-mail is one example of message-based communications.

Just as session-based client/server applications rely on network communication protocol stacks, such as TCP/IP or Novell NetWare's IPX, so message-based applications rely on message services such as Microsoft's Messaging API (MAPI), Lotus' Vendor Independent Messaging (VIM), Internet Simple Mail Transfer Protocol (SMTP) or a shared file link. Message services use **store-and-forward** methods to get each message to its destination: for example, e-mail systems store messages until the recipient opens their mail folder to read their mail, at which time the e-mail system forwards the message.

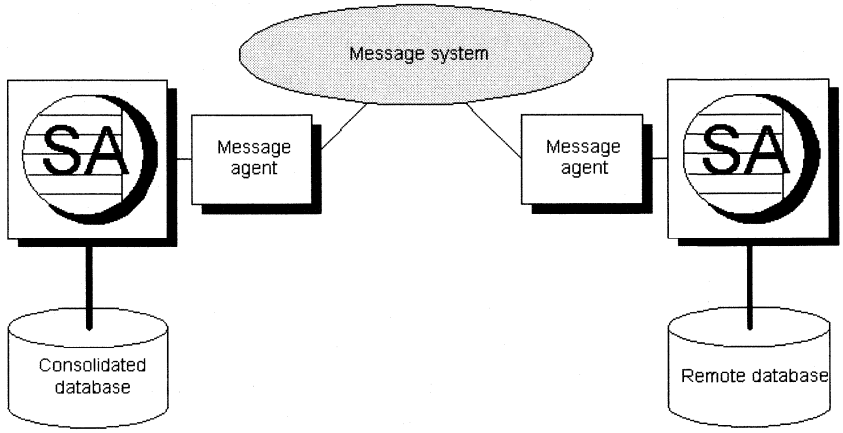
Unlike some session-based communications, many message-based systems do not guarantee that messages reach their destination, or that messages are received in the same order they were sent. SQL Remote incorporates a protocol to guarantee reception of replication updates in the correct order.



Building a replication system on top of a message system means that SQL Remote does not need to implement a store-and-forward system to get messages to their destination. Just as session-based client/server applications do not implement their own protocol stacks to pass information between client and server, so SQL Remote uses existing message systems to pass the messages. Each computer involved in a setup must have a message system client installed, and they exchange information through the message system.

The Message Agent

The SQL Remote technology is contained partly in the SQL Anywhere database engine and partly in the **Message Agent**. The Message Agent is a client application that sends and receives messages from database to database. The Message Agent is a program called DBREMOTE, and it must be installed at both the consolidated and at the remote sites.

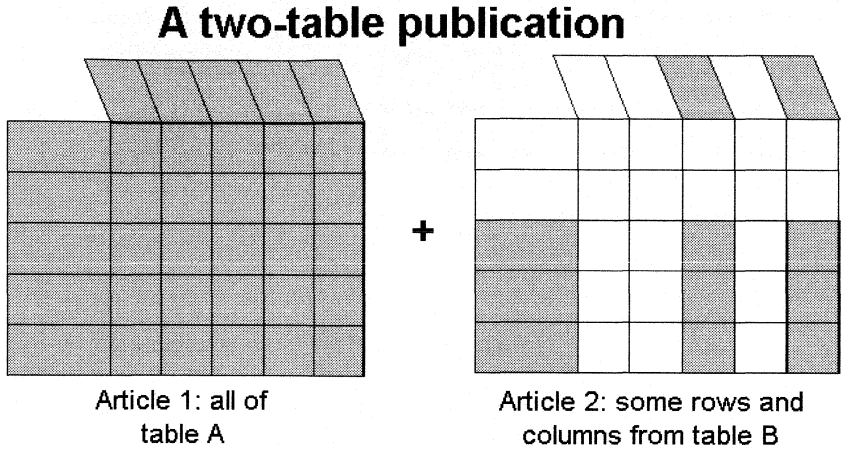


Publications and subscriptions

The **publication** is a database object describing data to be replicated. Remote users of the database who wish to receive a publication do so by **subscribing** to a publication.

Data is organized into publications

A publication may include data from several tables. Each table's contribution to a publication is called an **article**. Each article may consist of a whole table, or a subset of the rows and columns in a table.

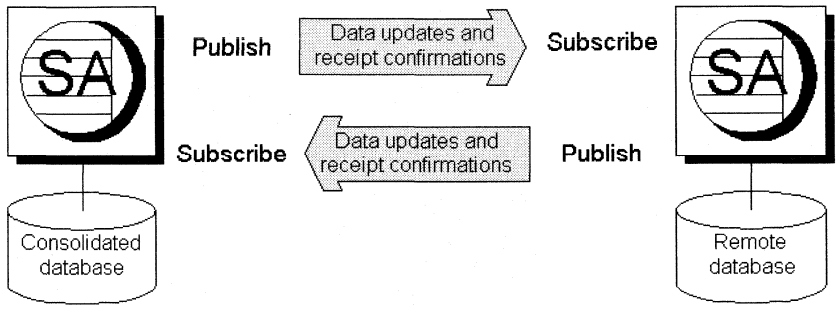


Periodically, the changes made to each publication in a database are replicated to all subscribers to that publication. These replications are called publication **updates**.

Each end of the replication needs a publication and a subscription

Remote databases subscribe to publications on the consolidated database so that they can receive data from the consolidated database. To do this, a **subscription** is created at the consolidated database, identifying the subscriber by name and by the publication they are to receive. SQL Remote always involves messages being sent two ways. The consolidated database sends messages containing publication updates to remote databases, but remote databases also send messages to the consolidated database.

For example, if data in a publication at the remote database is updated, those updates must be sent to the consolidated database. Even if the publication is never updated at the remote database, the remote database sends confirmation messages back to the consolidated database, to keep track of the status of the replication. Messages must be sent both ways, so not only does a remote database subscribe to a publication created at the consolidated database, but the consolidated database must subscribe to an identical publication created at the remote database.



When remote database users modify their own copies of the data, their changes are replicated to the consolidated database. When the replication is made successfully at the consolidated database the changes become part of the consolidated database's publication, and are included in the next round of updates to all remote sites (except the one it came from). In this way, replication from remote site to remote site takes place via the consolidated database.

When a subscription is initially set up, the two databases must be brought to a state where they both have the same set of information, ready to start replication. This process of setting up a remote database to be consistent with the consolidated database is called **synchronization**. Synchronization can be carried out manually, but the database extraction utility automates the process. You can run the extraction utility from Sybase Central or as a command-line utility.

The appropriate publication and subscription are created automatically at remote databases when you use the SQL Remote database extraction utility to create a remote database.

Remote database users

A replication setup includes many copies of the information in a database. While each replicate is a physically separate database on a separate computer, whether it be a sales representative's laptop computer or an office database on a server, all must stay consistent with the consolidated database. The entire replication setup may be considered a single database, with the master copy of all data being kept at the consolidated database.

Each remote site that submits replications to the consolidated database is considered to be a **remote user** of the consolidated database. In the case that a remote site is a multi-user server, the entire site is considered to be a single remote user of the consolidated database.

SQL Remote features

The following features are key to SQL Remote's design.

Transaction log-based replication

SQL Remote replication is based on the transaction log, enabling it to replicate only changes to data, rather than all data, in each update.

The transaction log is the repository of all changes made to a database. SQL Remote replicates changes made to databases as recorded in the transaction log. Periodically, all committed transactions in the consolidated database transaction log belonging to any publication are sent to remote databases. At remote sites, all committed transactions in the transaction log are periodically submitted to the consolidated database.

By replicating only committed transactions, SQL Remote ensures proper transaction atomicity throughout the replication setup and maintains a consistency among the databases involved in the replication, albeit with some time lag while the data is replicated.

Central administration

SQL Remote is designed to be centrally administered, at the consolidated database. This is particularly important for mobile workforce applications, where laptop users should not have to carry out database administration tasks. It is also important in replication involving small offices that have servers but little in the way of administration resources.

Administration tasks include setting up and maintaining publications, remote users, and subscriptions, as well as correcting errors and conflicts if they occur.

For more information

For information on administering remote users, see "Granting and revoking REMOTE and CONSOLIDATE permissions" in the chapter "SQL Remote Administration".

For information on administering publications, see "Setting up publications" in the chapter "SQL Remote Administration".

For information on administering subscriptions, see "Setting up subscriptions" in the chapter "SQL Remote Administration".

For information on replication conflicts and their management, see "Error reporting and conflict resolution in SQL Remote" in the chapter "SQL Remote Administration".

The passthrough mode is designed for direct intervention at remote databases from the consolidated site, and is described in "Using passthrough mode for administration" in the chapter "SQL Remote Administration".

You can carry out many administration tasks from the Sybase Central database management tool.

Support for many subscribers

SQL Remote is designed to support replication with many subscribers to a publication.

This feature is of particular importance for mobile workforce applications, which may require replication to the laptop computers of hundreds of sales representatives from a single office database.

Economical resource requirements

The only software required to run SQL Remote in addition to the SQL Anywhere engine is the Message Agent, which is an executable named DBREMOTE, and a message system. If you use the shared file link, no message system software is required as long as you have access to the directory where the message files are stored.

The Message Agent runs on Windows, DOS, Windows NT, and OS/2 platforms. The DOS Message Agent supports only the shared file link. Any SQL Anywhere server or standalone engine can act as a consolidated or remote database.

Memory and disk space requirements have been kept moderate for all components of the replication system, so that you do not have to invest in extra hardware to run SQL Remote.

Message systems supported by SQL Remote

The message systems supported by SQL Remote are:

- **MAPI** Used in e-mail systems, including Microsoft Mail.
- **FILE** A simple file-sharing protocol.
- **VIM** Used in e-mail systems such as Lotus cc:Mail and groupware products such as Lotus Notes.
- **SMTP** Used in Internet e-mail systems.

A database can exchange messages using one or more of the available message systems.

For information about supported message systems, see "Supported message types" in the chapter "SQL Remote Administration".

Tutorial: setting up SQL Remote using Sybase Central

The following sections are a tutorial describing how to set up a simple SQL Remote replication system using Sybase Central.

You do not need to enter SQL statements if you are using Sybase Central to administer SQL Remote. A tutorial for those who do not have access to Sybase Central is presented in "Tutorial: setting up SQL Remote using ISQL and DBXTRACT" on page 439, and contains the SQL statements executed behind the scenes by Sybase Central.

In this tutorial you will act as the DBA of the consolidated database, and set up a simple replication system. The simple example is a primitive model for an office "news" system, with a single table containing messages together with the database from which they were sent. The table is replicated in a setup with one consolidated database and one remote database. You can install this example on one computer.

The tutorial takes you through the following steps:

1. Ensure that the consolidated database site and all remote sites have properly configured message systems.
2. Add the message type you need to the database.
3. Add a publisher user ID to the database to identify the source of outgoing messages.
4. Add a remote user to the database for each remote database in the setup.
5. Create a publication on the consolidated database.
6. Create subscriptions for remote users to the publication.
7. Ensure that each remote user has an initial copy of the data on their database by synchronizing the data.
8. Configure and run the Message Agent at both consolidated and remote sites.
9. Start replicating data.

The following sections describe how to carry out each of these steps, apart from the message system installation, which is independent of SQL Anywhere.

Preparing for the Sybase Central replication tutorial

The following steps prepare you for the replication tutorial:

1. Create a directory to hold the files you make during this tutorial; for example C:\TUTORIAL.
2. The tutorial uses two databases: a consolidated database named HQ.DB and a remote database named FIELD.DB. At this point, you should create the hq database with the Create Database utility in Sybase Central:
   - Start Sybase Central. You will be creating a new database so you do not have to connect to any particular existing database.
   - Click Database Utilities in the left panel.
   - Double-click Create Database in the right panel. The Create Database wizard is displayed.
   - Create a database with filename C:\TUTORIAL\HQ.DB.
   - You can use the default settings for this database. Make sure you elect to maintain a transaction log. Replication cannot take place without a transaction log.
3. If you wish to set up the example system using a file-sharing link you must create a subdirectory for each of the two user IDs in the replication system. Create these subdirectories using the following statements at a system command line:

   mkdir c:\tutorial\hq

   mkdir c:\tutorial\field
4. If you wish to use a MAPI e-mail system as the message system you must have two e-mail user IDs and addresses available; one for the consolidated database and one for the remote database. In this tutorial these will be MAPI user IDs hq_mapi and field_mapi, with MAPI addresses hq_address and field_address.
5. Add a table to the consolidated database:
   - Connect to the hq database from Sybase Central, as user ID DBA:
     1. Click Connect from the Tools Menu.
     2. Enter the user ID DBA and the password SQL, then click More>>.

3 If the hq database is not running, enter the file name with path in the Database File field. If it is running, enter the name hq in the Database Name field.

4 Click OK to connect.

- Click the Tables folder of the hq database.
- Double-click Add Table, enter the name news, and click OK to create the empty table.
- Double-click the news table icon, then its Columns folder, and then double-click Add Column.
- Enter the column name id on the General tab, and on the Data type tab enter the column type INTEGER. Click Edit, and choose a pre-defined default of AUTOINCREMENT. Uncheck the Column Allows NULL setting so that NULLs are not allowed in this column. Then click OK to create the column.
- Double-click Add Column, and enter column name pub, of type CHAR(30) and with a pre-defined default of CURRENT PUBLISHER, not allowing NULL. Click OK to create the column.
- Double-click Add Column again, and enter column name text, of type CHAR(255), allowing NULL and with no default setting. Click OK to create the column.
- Create a two-column primary key. Select the id and pub columns (in that order), click the right mouse button, and select Add to Primary Key from the popup menu.

The two-column primary key, containing an autoincrementing integer and a user ID column, is a useful form of primary key for many tables involved in replication; inclusion of the CURRENT PUBLISHER column default prevents users at different databases from inserting rows with the same primary key value, which would cause conflicts. The CURRENT PUBLISHER default identifies each row by the database from which it originates. For more information, see "Designing publications" in the chapter "SQL Remote Administration".

You are now ready for the rest of the tutorial.

Set up the consolidated database in Sybase Central

This section of the tutorial describes how to prepare the consolidated database of a simple replication system.

Preparing a consolidated database for replication involves the following steps:

1 Create a message type to use for replication.

2 Grant PUBLISH permissions to a user ID to identify the source of outgoing messages.

3 Grant REMOTE permissions to all user IDs that are to receive messages.

4 Create a publication describing the data to be replicated.

5 Create subscriptions describing who is to receive the publication.

You require DBA authority to carry out these tasks.

Add a SQL Remote message type

All messages sent as part of replication use a message type. A message type description has two parts:

- A message link supported by SQL Remote. Currently, the link must be one of **FILE** or **MAPI**. A VIM link and an SMTP/POP link are expected to be available in the first quarter of 1996.
- An address for this message link, to identify the source of outgoing messages.

SQL Anywhere databases already have MAPI and FILE message types created, but you need to supply an address for the message type you will use. To add an address to a message type:

1 Click the SQL Remote folder on the left panel.

2 Double-click the Message Types folder.

3 Double-click whichever message type (MAPI or FILE) you are going to use for the tutorial.

4 Enter a publisher address to provide a return address for remote users. For the **file** message link, enter C:\TUTORIAL\HQ; the directory you have created to hold messages for the consolidated database. (In a production setup, file addresses are taken relative to the value of the SQLREMOTE environment variable or Directory registry setting, which would have been set to C:\TUTORIAL for this tutorial.) For the MAPI message link, enter the valid MAPI address you are using for the consolidated database.

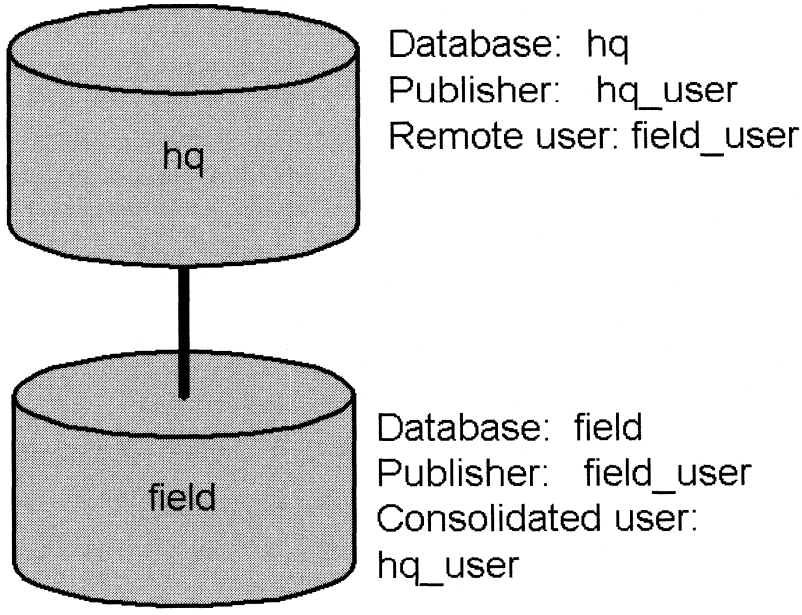
5 Click OK to save the message type.

Add the publisher and remote user to the database

In SQL Remote's hierarchical replication system, each database may have zero or one consolidated database immediately above it and zero or more databases immediately below it (remote databases).

In this tutorial, the current database is the consolidated database of a two-level system. It has no database above it, and only one remote database below it.

The following diagram illustrates the two databases:



For any database in a SQL Remote replication setup, there are three permissions that may be granted to identify databases on the hierarchy:

- PUBLISH permission identifies the current database in all outgoing messages
- REMOTE permission identifies each database receiving messages from the current database that is below it on the hierarchy
- CONSOLIDATE permission identifies a database receiving messages from the current database that is directly above it on the hierarchy.

Permissions can only be granted by a user with DBA authority. To carry out these examples you should connect from Sybase Central to HQ.DB as user ID **DBA**, with password **SQL**.

Add a database publisher user ID

Any database, consolidated or remote, that distributes changes to other databases in the replication system is a publisher database. Each database in the replication system is identified by a single user ID. You set that ID for your database by adding a publisher to the database. This section describes setting permissions for the consolidated hq database.

First create a user ID named hq_user, who will be the publisher user ID.

1. Click the Users & Groups folder on the left panel.
2. Double-click Add User. The New User Wizard is displayed.
3. Enter the name hq_user, with password hq_pwd, and click Next.
4. On the next page, ensure that the user is granted Remote DBA authority; this enables the user ID to run the Message Agent. Then click Next.
5. On the final page, check the box identifying this user ID as the publisher. Then click Finish to create the user.

A database can have only one publisher. You can find out who the publisher is at any time by opening the SQL Remote folder.

Add a remote user

Each remote database is identified in the consolidated database by a user ID with REMOTE permissions. Whether the remote database is a single-user database engine or a database server with many users, it needs a single user ID to represent it to the consolidated database.

In a mobile workgroup setting, remote users may already be users of the consolidated database, and so no new users would need to be added; although they would need to be set as remote users.

When a remote user is added to a database, the message system they use and their address under that message system need to be stored along with their database user ID.

You can add a remote user as follows:

1 Click the SQL Remote folder on the left panel, then click the Remote Users folder on the left panel.

2 Double-click Add Remote User on the right panel. The New Remote User wizard is displayed.

3 Create a remote user with name field_user, password field_pwd, message type file, and address field. For the message type and address, select the type (FILE or MAPI) and the corresponding address you are using for this user.

4 You should ensure that the Send Then Close option is checked. (In many production environments you would not choose Send Then Close, but it is convenient for this tutorial.)

5 You should ensure that the Remote DBA authority is checked, so that the user can run the Message Agent.

6 When you have finished all the entries, click Finish to create the remote user.

Add publications and subscriptions

This section describes how to add a publication to a database, and how to add a subscription for that publication to a user. You can add a publication that replicates all rows of the table news as follows:

1 Click the Publications folder in the SQL Remote folder.

2 Double-click Add Publication. The Publication Wizard is displayed.

3 Name the publication pub_news on page one of the Wizard. On page two, click Add Table and select news from the list. Leave the All Columns button selected, press OK, and complete the Wizard to create the publication. This publication is one of the simplest that can be created. Publications can involve multiple tables, and subsets of the rows and columns in tables. These can all be defined using the Publication Wizard.

Add a subscription

Each user ID that is to receive changes to a publication must have a **subscription** to that publication. Subscriptions can only be created for a valid remote user.

Add a subscription to the pub_news publication for the remote database user field_user:

1 Double-click the Publications folder, which is in the SQL Remote folder, so that the pub_news publication is displayed in the left panel.

2 Click the Remote Users folder so that remote users are displayed in the right panel.

3 Drag the field_user user from the right panel onto the pub_news publication in the left panel. The Create Subscription window is displayed. You should leave both the Subscribe By and With Value text boxes empty; click OK to create the subscription.

Set up the remote database in Sybase Central

The remote database needs to be created and configured in order to send and receive messages and participate in a SQL Remote setup. Like the consolidated database, the remote database needs a publisher (in this case, the field_user user ID) to identify the source of outgoing messages, and it needs to have hq_user identified as a remote user. It needs the pub_news publication to be created and needs a subscription created for the hq_user user ID. The remote database also needs to be **synchronized** with the consolidated database; that is, it needs to have a current copy of the data in order for the replication to start. In this case, there is no data in the table as yet.

The database extraction utility enables you to carry out all the steps needed to create a remote database complete with subscriptions and required user IDs.

To extract a database from the consolidated database for remote user field_user:

1 Click the Remote Users folder, which is in the SQL Remote folder.

2 Right-click the field_user remote user, and select Extract Database from the popup menu. The Extraction Wizard is displayed.

3 On page 1, choose to extract from the running database hq.

4 On the next page, use the dba user ID, with password sql

5 On the next page, choose to Start Subscriptions for user field_user.

6 Create the database as file C:\TUTORIAL\FIELD.DB, using a transaction log in the same directory.

7 Choose to extract all parts of the schema.

8 Leave the other options at their default settings, and create the remote database.

In a proper SQL Remote setup, the remote database field would need to be loaded on to the computer using it, together with a SQL Anywhere engine and any client applications required. For this tutorial, we leave the database where it is and use ISQL to input and replicate data.

You should connect to the field.db database as DBA and confirm that all the database objects are created.

The system is now ready for replication. For the next step, inserting and replicating data, see the section "Start replicating data" on page 447.

Tutorial: setting up SQL Remote using ISQL and DBXTRACT

The following sections are a tutorial describing how to set up a simple SQL Remote replication system for users without Windows 95 or Windows NT 3.51 or later, who cannot run Sybase Central. It may also be useful to users of Sybase Central who want to know what Sybase Central is doing behind the scenes.

This tutorial describes the SQL statements for managing SQL Remote, which can be run from ISQL. It also describes how to run the DBXTRACT command-line utility to extract remote databases from a consolidated database.

In this tutorial you will act as the DBA of the consolidated database, and set up a simple replication system. The simple example is a primitive model for an office "news" system, with a single table containing messages together with the user who entered them. The table is replicated in a two-level setup with one consolidated database and one remote database. You can install this example on one computer.

The tutorial takes you through the following steps:

1 Ensure that the consolidated database site and all remote sites have properly configured message systems.

2 Create a MESSAGE TYPE in the database.

3 Grant PUBLISH permissions to identify the source of outgoing messages.

4 Grant REMOTE permissions for each remote database in the setup.

5 Create publications on the consolidated database.

6 Create subscriptions for remote users to each required publication.

7 Ensure that each remote user has an initial copy of the data on their database by synchronizing the data.

8 Install and run the Message Agent at both consolidated and remote sites.

9 Start replicating data.

The following sections describe how to carry out each of these steps, apart from the message system installation, which is independent of SQL Anywhere.

Preparing for the replication tutorial

The following steps prepare you for the replication tutorial:

1. Create a directory to hold the files you make during this tutorial; for example C:\TUTORIAL.
2. The tutorial uses two databases: a consolidated database named HQ.DB and a remote database named FIELD.DB. Create these databases using the following statements at a command line:

   dbinit c:\tutorial\hq.db

   dbinit c:\tutorial\field.db
3. The database initialization tool for Windows 3.x is DBINITW rather than DBINIT. Under Windows 3.x, you should execute the commands from the Program Manager File>.Run menu or you could make an icon for each command.
4. If you wish to set up the example system using a file-sharing link you must create a subdirectory for each of the two user IDs in the replication system. Create these subdirectories using the following statements at a command line:

   mkdir c:\tutorial\hq

   mkdir c:\tutorial\field
5. If you wish to use a MAPI e-mail system as the message system you need to have two e-mail user IDs and addresses available. In this tutorial these will be user IDs hq_mapi and field_mapi, with MAPI addresses hq_address and field_address.
6. Connect to HQ.DB from ISQL as user ID DBA, and create a table in the consolidated database to replicate. To do this:
   1. Type CONNECT in the ISQL Command window, and click Execute.
   2. Enter the user ID DBA and the password SQL, and click More.
   3. If the hq database is already running, type hq in the Database Name box. If it is not running, enter the Database File C:\TUTORIAL\HQ.DB. Then click OK to connect.
   4. Enter the following CREATE TABLE statement:

```
CREATE TABLE news (
   id INT DEFAULT AUTOINCREMENT,
   pub CHAR(30) DEFAULT CURRENT PUBLISHER,
   text CHAR(255),
   PRIMARY KEY ( id, pub )
```

```
)
```

The two-column primary key, containing an autoincrementing integer and a user ID column, is a useful form of primary key for many tables involved in replication; inclusion of the CURRENT PUBLISHER column default prevents users at different databases from inserting rows with the same primary key value, which would cause conflicts. The CURRENT PUBLISHER default identifies each row by the database from which it originates. For more information, see "Designing publications" in the chapter "SQL Remote Administration".

You are now ready for the rest of the tutorial.

Set up the consolidated database

This section of the tutorial describes how to set up the consolidated database of a simple replication system.

Setting up a consolidated database involves the following steps:

1 Create a message type to use for replication.

2 Grant PUBLISH permissions to a user ID to identify the source of outgoing messages.

3 Grant REMOTE permissions to all user IDs that are to receive messages.

4 Create a publication describing the data to be replicated.

5 Create subscriptions describing who is to receive the publication.

You require DBA authority to carry out these tasks.

Create a SQL Remote message type

All messages sent as part of replication use a message type. A message type description has two parts:

- A message link supported by SQL Remote. Currently, the link must be one of FILE, MAPI, VIM, or SMTP.
- An address for this message link, to identify the source of outgoing messages.

You can create a file message type using the following statement:

```
CREATE REMOTE MESSAGE TYPE file ADDRESS 'hq'
```

You can create a MAPI message type using the following statement:

```
CREATE REMOTE MESSAGE TYPE mapi ADDRESS 'hq_address'
```

Grant PUBLISH and REMOTE at the consolidated database

In the hierarchical replication system supported by SQL Remote, each database may have one consolidated database immediately above it in the hierarchy and many databases immediately below it on the hierarchy (remote databases).

PUBLISH permission identifies the current database for outgoing messages, and the REMOTE permission identifies each database receiving messages from the current database.

Permissions can only be granted by a user with DBA authority. To carry out these examples you should connect using the ISQL utility to HQ.DB as user ID DBA, with password SQL.

GRANT PUBLISH to identify outgoing messages

Each database that distributes its changes to other databases in the replication system is a publisher database. Each database in the replication system that publishes changes to a database is identified by a single user ID. You set that ID for your database using the GRANT PUBLISH statement. This section describes setting permissions for the consolidated database (HQ.DB).

The following statements create a publisher:

```
GRANT CONNECT TO hq_user IDENTIFIED BY hq_pwd ;
GRANT PUBLISH TO hq_user ;
```

You can check the publishing user ID of a database at any time using the CURRENT PUBLISHER special constant:

```
SELECT CURRENT PUBLISHER
```

GRANT REMOTE for each database to which you send messages

Each remote database is identified using the GRANT REMOTE statement. Whether the remote database is a single-user database engine or a database server with many users, it needs a single user ID to represent it to the consolidated database.

In a mobile workgroup setting, remote users may already be users of the consolidated database, and so this would require no extra action on the part of the DBA.

The GRANT REMOTE statement identifies the message system to be used when sending messages to the recipient, as well as the address.

The following statements grant REMOTE permissions to a database with user ID field for a MAPI-based message system:

```
GRANT CONNECT TO field_user IDENTIFIED BY field_pwd
;
GRANT REMOTE TO field_user TYPE mapi ADDRESS
'field_address' ;
```

The address string is the MAPI address of field_address, enclosed in single quotes. The MAPI address is not the same as the mail user ID.

The following statements grant REMOTE permissions to a database with user ID field_user for a file-sharing message system:

```
GRANT CONNECT TO field_user
IDENTIFIED BY field_pwd ;

GRANT REMOTE TO field_user
TYPE file ADDRESS 'field' ;
```

The address string is the directory used to hold messages for field_user, enclosed in single quotes. The directory is a subdirectory of the SQLREMOTE environment variable or Directory registry entry, if set, or of the current working directory if SQLREMOTE is not set.

Create publications and subscriptions

A publication is created using a CREATE PUBLICATION statement. This is a data definition language statement, and requires DBA authority. For the tutorial, you should connect to the hq database as user ID DBA, password SQL, to create a publication.

Set up a publication at the consolidated database

Create a publication that replicates all rows of the table news using the following statement:

```
CREATE PUBLICATION pub_news (
   TABLE news
)
```

This publication is about the simplest that can be created. Publications can involve multiple tables, and subsets of the rows and columns in tables. For more information on publications, see "Designing publications" in the chapter "SQL Remote Administration".

Set up a subscription

Each user ID that is to receive changes to the publication must have a subscription. The subscription can only be created for a user who has REMOTE permissions. The GRANT REMOTE statement contains the address to use when sending the messages.

Create a subscription to the pub_news publication for the remote user field_user:

```
CREATE SUBSCRIPTION TO pub_news FOR field_user ;
```

The full CREATE SUBSCRIPTION statement allows control over the data in subscriptions; allowing users to receive only some of the rows in the publication. For more information, see "CREATE SUBSCRIPTION statement" in the chapter "Watcom-SQL Statements".

The CREATE SUBSCRIPTION statement identifies the subscriber and defines what they receive. However, it does not synchronize data, or start the sending of messages.

Set up the remote database

The remote database needs to be configured in order to send and receive messages and participate in a SQL Remote setup. Like the consolidated database, the remote database needs a CURRENT PUBLISHER to identify the source of outgoing messages, and it needs to have the consolidated database identified as a subscriber. The remote database also needs the publication to be created and needs a subscription created for the consolidated database. The remote database also needs to be **synchronized** with the consolidated database; that is, it needs to have a current copy of the data in order for the replication to start.

The DBXTRACT utility enables you to carry out all the steps needed to create a remote database complete with subscriptions and required user IDs.

Extract the remote database information

Leave the hq database running, and type the following command at the system command line (all on one line) to extract a database for the user remote from the consolidated database:

```
dbxtract -c "dbn=hq;uid=dba;pwd=sql" c:\tutorial field_user
```

In Windows 3.x, you should run the following command either from File>.Run in Program Manager or from an icon:

```
dbxtracw -c "dbn=hq;uid=dba;pwd=sql" c:\tutorial field_user
```

This command assumes the **hq** database is currently running on the default server. If the database is not running, you should enter a database file parameter

```
dbf=c:\tutorial\hq.db
```

instead of the **dbn** database name parameter. For details of the DBXTRACT utility and its options, see "The DBXTRACT command-line utility" in the chapter "SQL Anywhere Components".

The DBXTRACT command creates a SQL command file named RELOAD.SQL in the current directory and a data file in the C:\TUTORIAL directory. It also starts the subscriptions to the remote user.

The next step is to load these files into the remote database.

Load the remote database information

To load the database information, connect to the remote database FIELD.DB from ISQL as user ID **DBA** and password **SQL**.

You can do this by typing CONNECT in the ISQL command window, and then entering DBA, SQL, and the database file C:\TUTORIAL\FIELD.DB in the connection dialog box.

Once connected, run the RELOAD.SQL command file:

```
READ c:\tutorial\reload.sql
```

The RELOAD.SQL command file carries out the following tasks:

- Creates a message type at the remote database.
- Grants PUBLISH and REMOTE permissions to the remote and consolidated database, respectively.
- Creates the table in the database. If the table had contained any data before extraction, the command file would fill the replicated table with a copy of the data.
- Creates a publication to identify the data being replicated.
- Creates the subscription for the consolidated database, and starts the subscription.

While connected to the field database as DBA and confirm that the table is created by typing

```
SELECT * FROM news.
```

The system is now ready to start replicating data.

Start replicating data

You now have a replication system in place. In this section, data is replicated from the consolidated database to the remote database, and from the remote to the consolidated database.

Send data from the consolidated database

First, connect to the consolidated database hq from the ISQL utility as user ID DBA, with password SQL.

Enter and commit a row into the news database:

```
INSERT INTO news (text)
VALUES ('Welcome to our news database.') ;
COMMIT ;
```

To send the new row to the remote database, you must run the Message Agent at the consolidated database. The DBREMOTE program is the Message Agent. The following statement at the command line runs DBREMOTE:

dbremote -c "dbn=hq;uid=dba;pwd=sql"

In Windows 3.x, you should run the following command either from File>.Run in Program Manager or from an icon:

dbremotw -c "dbn=hq;uid=dba;pwd=sql"

These command lines assume that the hq database is currently running on the default server. For information on DBREMOTE command line switches, see "The SQL Remote Message Agent" in the chapter "SQL Anywhere Components".

If you are running under MAPI, you will be prompted for a MAPI user ID and password in order to connect to the mailing system. You should use the user ID and password for the consolidated database user.

Receive data at the remote database

To receive the insert statement at the remote database, you must run the Message Agent, DBREMOTE, at the remote database. The following statement at the command line runs DBREMOTE on the field database running on a server named field.

dbremote -c "dbn=field;uid=dba;pwd=sql"

If you are running under MAPI, you will be prompted for a MAPI user ID and password in order to connect to the mailing system. You should use the user ID and password for the remote database user.

The Message Agent window displays status information while running. This information can be output to a log file for record keeping in a real setup. You will see that the Message Agent first receives a message from hq, and then sends a message. This return message contains confirmation of successful receipt of the replication update; such confirmations are part of the SQL Remote message tracking system that ensures message delivery even in the event of message system errors.

If you now connect to the remote field database using ISQL, and inspect the news table, you will see that the row entered at the consolidated database has been replicated.

Replicate from the remote database to the consolidated database

You should now try entering data at the remote database and sending it to the consolidated database:

1 INSERT a row at the remote database. For example

```
INSERT INTO news (text)
VALUES ('Thanks very much.')
```

2 COMMIT the row.

3 With the FIELD.DB database running, run DBREMOTE to send the message to the consolidated database.

dbremote -c "dbn=field;uid=dba;pwd=sql"
(For Windows 3.x, run the DBREMOTW equivalent.)

4 With the HQ.DB database running, run DBREMOTE to receive the message at the consolidated database:

```
dbremote -c "dbn=hq;uid=dba;pwd=sql"
```

5 Connect to the consolidated database and display news:

❖ **Show the rows of the news table.**

- Type the following statement:

```
SELECT *
FROM news
```

id	pub	text
1	hq	Welcome to our news database.
2	field	Thanks very much.

The pub column identifies the publisher of the database at which the row was entered. The first row was entered at the consolidated database, and has an entry hq, while the second row was entered at the remote database and so has a value field. With this column as part of the primary key, there will be no conflicts arising from different users at different databases inserting the same primary key value.

A sample publication

The command file SALESPUB.SQL contains a set of statements that creates a publication on the sample database. This publication illustrates several of the points of the tutorial, in more detail.

To add the publication to the sample database:

1 Connect to the sample database from ISQL.

2 Type **READ** *path***salespub.sql**, where *path* is your SQL Anywhere installation directory.

The SALESPUB.SQL publication adds columns to some of the tables in the sample database, creates a publication and subscriptions, and also adds triggers to resolve update conflicts that may occur.

Some sample SQL Remote setups

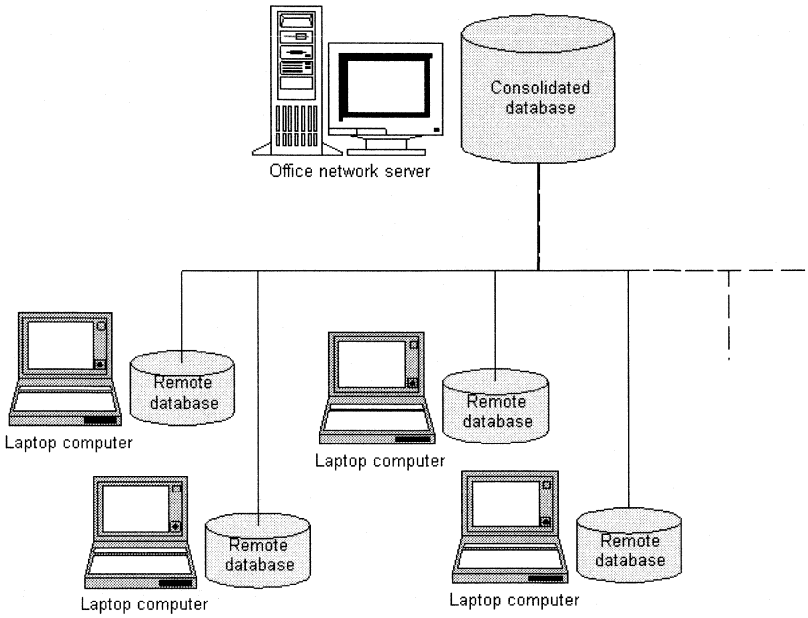
While SQL Remote can provide replication services in many different environments, its features are designed with the following characteristics in mind:

- SQL Remote should be a solution even when no administration load can be assigned to the remote databases, as in mobile workforce applications.
- Data communication among the sites may be occasional and indirect: it need not be permanent and direct.
- Memory and resource requirements at remote sites are assumed to be at a premium.

The following examples show some typical SQL Remote setups.

Server-to-laptop replication for mobile workforces

SQL Remote provides two-way replication between a database on an office network and standalone databases on the laptop computers of sales representatives. Such a setup may use a MAPI e-mail system as a message transport.



The office server may be running a SQL Anywhere network server for Windows NT, Novell NetWare, or OS/2, to manage the company database. The Message Agent at the company database runs as a SQL Anywhere client, either on the same computer as the SQL Anywhere server, or on another computer on the network. For MAPI support, the Message Agent should run on a Windows or Windows NT system.

The laptop computers may be running Windows 95 or Windows for Workgroups, and each sales representative has a SQL Anywhere standalone database engine to manage their own data.

While away from the office, a sales representative can make a single phone call from their laptop to carry out the following functions:

- Collect new e-mail.
- Send any e-mail messages they have written.
- Collect publication updates from the office server.
- Submit any local updates, such as new orders, to the office server.

The updates may include, for example, new specials on the products the sales representative handles, or new pricing and inventory information. These are read by the Message Agent on the laptop and applied to the sales rep's database automatically, without requiring any additional action on the sales representative's part.

The new orders recorded by the sales representative are also automatically submitted to the office without any extra action on the part of the sales representative.

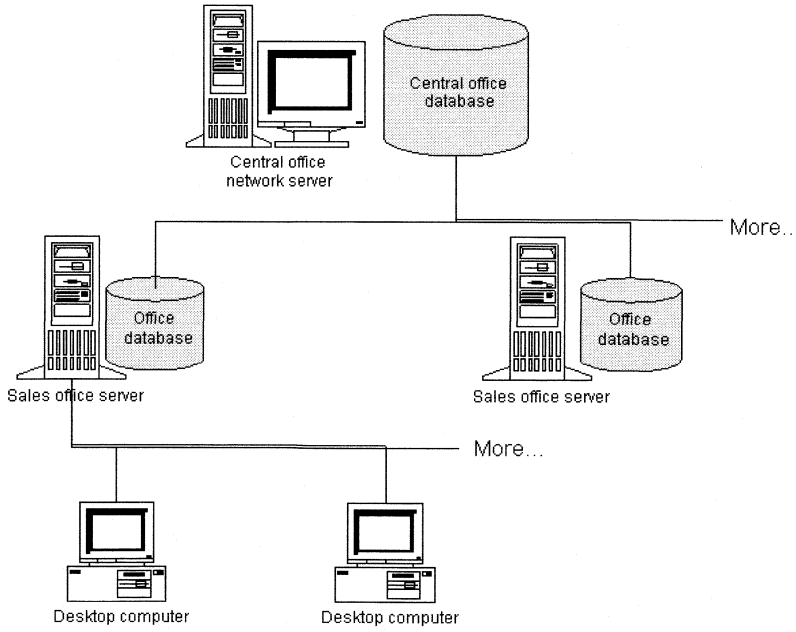
Server-to-server replication among offices

SQL Remote provides two-way replication between database servers at sales offices or outlets and a central company office, without requiring database administration experience at each sales office beyond the initial setup and that required to maintain the server.

Coupled with SQL Anywhere's modest administration requirements, this makes SQL Anywhere ideal for companies with small sales offices and outlets, where the server may be used by only a few people.

SQL Remote is not designed for up-to-the-minute data availability at each site. Instead, it is appropriate where data can be replicated at periods of an hour or so.

Such a setup may use a MAPI e-mail system to carry the replication, if there is already a company-wide e-mail system. Alternatively, an occasional dialup system and file transfer software can be used to implement a FILE message system.



SQL Remote is easy to configure to allow each office to receive their own set of data. Tables that are of office interest only (staff records, perhaps, if the office is a franchise) may be kept private in the same database as the replicated data.

Layers can be added to SQL Remote hierarchies: for example, each sales office server could act as a consolidated database, supporting remote subscribers who work from that office.

CHAPTER 27

SQL Remote Administration

About this chapter

This chapter details design, set-up, and management issues for SQL Remote administrators.

Some of the ground covered in the chapter "Introduction to SQL Remote Replication" is revisited in more detail, and several new topics are covered.

Contents

Topic	Page
SQL Remote administration overview	456
SQL Remote message types	457
Managing SQL Remote permissions	463
Setting up publications	470
Designing publications	475
Setting up subscriptions	483
Synchronizing databases	484
How statements are replicated by SQL Remote	489
Managing a running SQL Remote setup: overview	494
Running the SQL Remote Message Agent	496
The SQL Remote message tracking system	500
Transaction log and backup management for SQL Remote	503
Error reporting and conflict resolution in SQL Remote	507
Using passthrough mode for administration	513

SQL Remote administration overview

There are two distinct phases of SQL Remote administration:

- Designing, implementing, and deploying a SQL Remote replication setup.
- Maintaining an operating SQL Remote setup.

SQL Remote is built to provide robust replication within a well-defined setup. Consequently, while many iterations of schema and publication design and testing can be performed, SQL Remote is not designed to support extensive alterations to database object definitions while the replication system is up and running. Upgrading a SQL Remote setup may well involve resynchronizing all remote users.

This chapter describes topics in the following categories:

- **Design and setup of a SQL Remote installation** This includes creating message types, managing permissions, designing and creating publications, managing subscriptions, as well as extracting databases.
- **Messages and the Message Agent** Using the Message Agent to send and receive messages, and how the message tracking system works.
- **Error reporting and conflict resolution** Major inconsistencies, such as two users inserting rows with the same primary key, must be designed out of your publications. Update conflicts, however, can be resolved using triggers.
- **Direct intervention at remote databases** The passthrough mode enables SQL statements to be applied at remote databases. You may need to make some changes to your database design and applications to make them suitable for replication and updating at multiple sites.

  For more information on such possible changes, see the section "Designing publications" on page 475.

Sybase Central is the recommended tool for SQL Remote administration. It provides a graphical interface to all the elements of a SQL Remote setup: message types, remote users, publishers, publications, subscriptions, and the database extraction utility for synchronizing databases. If you do not have access to a platform on which Sybase Central can be run (Windows 95 or Windows NT 3.51 or later) you can carry out administration tasks using ISQL to enter SQL statements and using the command-line DBXTRACT utility to synchronize databases. This chapter describes both approaches.

SQL Remote message types

You need to define the message types you will use for sending messages before you add remote users to your database.

Supported message types

SQL Remote supports a set of message links. The links currently supported include:

- **FILE** The FILE link: storage of message files in directories on a shared file system for reading by other databases.
- **MAPI** Microsoft's messaging API (MAPI) link, used in Microsoft Mail and other electronic mail systems.
- **VIM** Lotus's Vendor Independent Messaging (VIM), used in Lotus Notes and cc: Mail.
- **SMTP** Internet Simple Mail Transfer Protocol (SMTP/POP), used in Internet e-mail.

The links are implemented as DLLs on Windows 3.x, Windows 95, Windows NT, and OS/2. The Message Agent for NetWare, DOS and QNX supports the FILE link only, as a compiled library.

Supported message types

This section describes the message types supported by SQL Remote.

The MAPI message system

The Message Application Programming Interface (MAPI) is used in several popular e-mail systems, such as Microsoft Mail. SQL Remote supports the MAPI message system under Windows 3.x, Windows 95, and Windows NT.

To use SQL Remote and a MAPI message system, each database participating in the setup requires a MAPI user ID and address. These are distinct identifiers: the MAPI address is the destination of each message, and the MAPI user ID is the name entered by a user when they connect to their mail box.

Although SQL Remote messages may arrive in the same mail box as e-mail intended for reading, they do not show up in your e-mail inbox. SQL Remote sends application-defined messages, which MAPI identifies and hides when the mailbox is opened. In this way, users can use the same e-mail address and same connection to receive their personal e-mail and their database updates, yet the SQL Remote messages do not interfere with the mail intended for reading.

One exception to this is if a message is routed via the Internet, in which case the special message type information is lost. The message then does show up in the recipient's mailbox.

The FILE message system

SQL Remote can be used even if you do not have a message system in place, by using the FILE message link.

The FILE message link is a file-sharing system. A FILE address for a remote user is a subdirectory into which all their messages are written. To retrieve messages from their "inbox", the user reads the messages from the directory containing their files. Return messages are sent to the address (written to the directory) of the consolidated database.

The FILE link addresses are typically subdirectories of a shared directory that is available to all SQL Remote users, whether by modem or on a local area network. Each user should have a registry entry or SQLREMOTE environment variable pointing to the shared directory.

You can also use the FILE link to put the messages in directories on the consolidated and remote machines. A simple file transfer mechanism can then be used to exchange the files periodically to effect replication.

The VIM message system

The Vendor Independent Messaging system (VIM) is used in Lotus cc: Mail and Lotus Notes.

To use SQL Remote and a VIM message system, each database participating in the setup requires a VIM user ID and address. These are distinct identifiers: the VIM address is the destination of each message, and the VIM user ID is the name entered by a user when they connect to their mail box.

VIM messages do appear in your mailbox along with mail intended for reading.

The SMTP message system

The Simple Mail Transfer Protocol (SMTP) is used in Internet e-mail products.

To use SQL Remote and a SMTP message system, each database participating in the setup requires a SMTP user ID and address. These are distinct identifiers: the SMTP address is the destination of each message, and the SMTP user ID is the name entered by a user when they connect to their mail box.

SMTP messages do appear in your mailbox along with mail intended for reading.

Specifying a message type

To specify a message type for a database, you need to specify a message link, and also an address, under that link, for the publisher user ID of the database.

The publisher address at a consolidated database is used by the database extraction utility as a return address when creating remote databases. It is also used by the Message Agent to identify where to look for incoming messages for the FILE link.

The SQL statement to create a message type has the following syntax:

CREATE REMOTE MESSAGE TYPE *link-name*
ADDRESS *address-string*

where *link-name* is one of the message links supported by SQL Remote, and *address-string* is the publisher's address under that message link.

The SQL statement to alter the publisher's address for a message type has the following syntax:

ALTER REMOTE MESSAGE TYPE *link-name*
ADDRESS *new-address-string*

The SQL statement to drop a message type has the following syntax:

DROP REMOTE MESSAGE TYPE *link-name*

You can create and alter message types in Sybase Central. The Message Type folder is inside the SQL Remote folder.

The address supplied with a message type definition is closely tied to the publisher ID of the database. This is discussed further in the following section.

Message link control parameters

Each message link has several parameters that govern aspects of its behavior. This section documents these parameters.

Where the parameters are held

The message link control parameters are stored in the registry under Windows NT and Windows 95, and in SQLANY.INI under Windows 3.x.

FILE message control parameters

The FILE message system uses the following control parameters:

- **Directory** This is set to the of the directory under which the messages are stored. The setting is an alternative to the SQLREMOTE environment variable.
- **Debug** This is set to either YES or NO, with the default being NO. When set to YES, all file system calls made by the FILE link are displayed.

The FILE section of the SQLANY.INI file (Windows 3.x) has the following entries:

```
[FILE]
Directory=path
Debug=yes | no
```

On NetWare, you should create a file named DBREMOTE.INI in the SYS:\SYSTEM directory to hold the directory setting.

MAPI message control parameters

The MAPI message system uses the following control parameters:

- **IPM_Send** This can be set to YES or NO (default NO). If set to YES, the MAPI link sends IPM messages, which are visible in the mailbox. If set to NO, the MAPI link sends IPC messages, which are not visible in the mailbox. This may be useful if your MAPI provider does not support IPC messages.
- **IPM_Receive** This can be set to YES or NO (default NO). If set to YES, the MAPI link receives IPM messages, which are visible in the mailbox. If set to NO, the MAPI link receives IPC messages, which are not visible in the mailbox. This may be useful if your MAPI provider does not support IPC messages. Also, it may be useful when receiving messages over the Internet. In this case, the sender might not be using MAPI or the IPC attributes are have been lost.
- **Force_Download** (default YES) controls if the MAPI_FORCE_DOWNLOAD flag is set when calling MapiLogon. This might be useful when using remote mail software that dials when this flag is set.
- **Debug** When set to YES, displays all MAPI calls and the return codes. This is useful for troubleshooting MAPI support problems. Default is NO.

The MAPI section of the SQLANY.INI file (Windows 3.x) has the following entries:

```
[MAPI]
```

```
IPM_Send=yes | no
IPM_Receive=yes | no
Force_Download=yes | no
Debug=yes | no
```

SMTP message control parameters

The SMTP message system uses the following control parameters:

- **smtp_host** The name of the computer on which the SMTP Server is running.
- **smtp_address** The return address if the mail cannot be delivered. If the address of a subscriber database is wrong the mail may be sent but later returned to this address.
- **local_host** In some environments, the winsock call gethostname cannot return a local host name, yet the local host name is necessary to initiate an SMTP session with an SMTP host. In such environments, you can set **local_host**, and its value is sent to the SMTP host to identify the local host instead of calling gethostname and gethostbyname.
- **pop3_host** The name of the computer on which the POP3 host is running. This is generally the same as the SMTP host.
- **pop3_userid** The user ID for your POP3 account.
- **pop3_password** The password for your POP3 account.
- **Debug** When set to YES, displays all SMTP calls and the return codes. This is useful for troubleshooting SMTP support problems. Default is NO.

If all of the fields are set, or if only the Debug field is not set, the login window is not displayed.

The SMTP section of the SQLANY.INI file (Windows 3.x) has the following entries:

```
[SMTP]
smtp_host=smtp_host
smtp_address=smtp_address
pop3_host=pop3_host
pop3_userid=userid
pop3_password=password
Debug=yes | no
```

VIM message control parameters

The VIM message system uses the following control parameters:

- **Path** This corresponds to the Path field in the cc: Mail login window. It is not applicable to and is ignored under Lotus Notes.
- **Userid** This corresponds to the User ID field in the cc: Mail login window.
- **Password** This corresponds to the Password field in the cc: Mail login window. If all of Path, Userid, and Password are set, the login window is not displayed.
- **Debug** When set to YES, displays all VIM calls and the return codes. This is useful for troubleshooting VIM support problems. Default is NO.
- **Receive_All** When set to YES, the Message Agent checks all messages to see if they are SQL Remote messages. When set to NO (the defaut), the Message Agent looks only for messages of the application-defined type SQLRemoteData. Setting ReceiveAll to YES is useful in setups where the message type is lost, reset, or never set. This includes setups including cc: Mail messages.
- **Send_VIM_Mail** When set to YES, the Message Agent sends messages compatible with SQL Anywhere releases before 5.5.01, and compatible with cc: Mail. If this is set to YES, you should ensure that Receive_All is set to YES also.

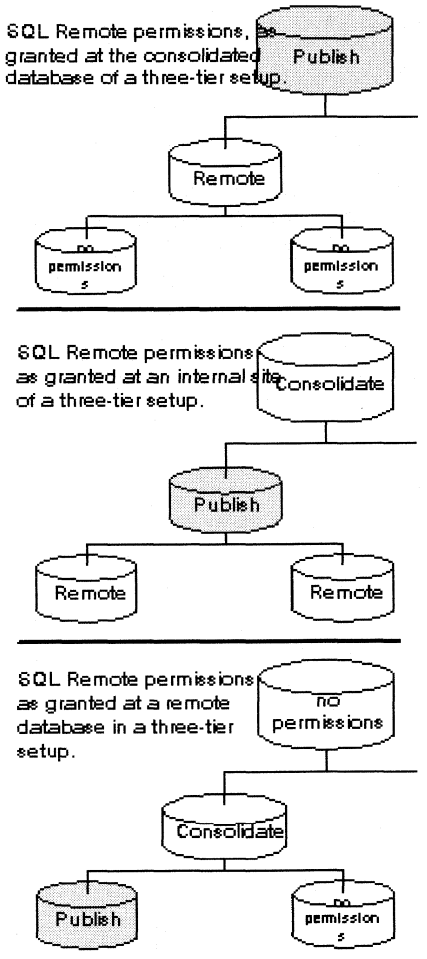
The VIM section of the SQLANY.INI file (Windows 3.x) has the following entries:

```
[VIM]
Path=path
Userid=userid
Password=password
Debug=yes | no
Receive_All = yes | no
Send_VIM_Mail = yes | no
```

Managing SQL Remote permissions

Users of a database involved in SQL Remote replication are identified by one of the following sets of permissions:

- **PUBLISH** A single user ID in a database is identified as the publisher for the database. All outgoing SQL Remote messages, including both publication updates and receipt confirmations, are identified by the publisher user ID. The publisher is the user who has been granted PUBLISH permissions on the database, and can be found by selecting the special constant CURRENT PUBLISHER or by opening the SQL Remote folder in Sybase Central. Every database in a SQL Remote setup must have a single publisher user ID, as every database in a SQL Remote setup sends messages.
- **REMOTE** All recipients of messages from the current database, or senders of messages to the current database, who are immediately lower on the SQL Remote hierarchy than the current database must be granted REMOTE permissions. The GRANT REMOTE statement identifies a user ID, as well as a message system and an address appropriate to that message system to where the messages are sent. It also gives the frequency at which messages are sent.
- **CONSOLIDATE** At most one user ID may be granted CONSOLIDATE permissions in a database. CONSOLIDATE permissions identifies a database immediately above the current database in a SQL Remote setup. Each database can have only one consolidated database directly above it. The permissions in a three-level SQL Remote setup are summarized in the following diagrams. In each diagram one database is gray; the diagram shows the permissions that need to be granted in that database for the user ID representing each of the other databases. The phrase "No permissions" means that the database is not granted any permissions in the gray database.



Granting the appropriate PUBLISH and CONSOLIDATE permissions at remote databases is done automatically by the database extraction utility.

Granting and revoking PUBLISH permissions

When a database sends a message, a user ID representing that database is included with the message to identify its source to the recipient. The GRANT PUBLISH statement identifies a user ID as the publisher user ID of the database.

A publisher is required even for read-only remote databases within a replication system, as even these databases send confirmations to the consolidated database to maintain information about the status of the replication. The GRANT PUBLISH statement for remote databases is carried out automatically by the database extraction utility.

Granting PUBLISH permissions

PUBLISH permissions are granted using the GRANT PUBLISH statement:

```
GRANT PUBLISH TO userid ;
```

The *userid* is a user with CONNECT permissions on the current database.

For example, the following statement grants PUBLISH permissions to user S_Beaulieu:

```
GRANT PUBLISH TO S_Beaulieu
```

You can grant PUBLISH permissions from Sybase Central. You should connect to the database as a user with DBA permissions.

1 Click the SQL Remote folder on the left panel.

2 Double click Set Publisher, and select a user from the list.

3 Click OK to set the selected user as the database publisher.

Revoking PUBLISH permissions

The REVOKE PUBLISH statement revokes the PUBLISH permissions from the current publisher:

```
REVOKE PUBLISH FROM userid
```

You can revoke PUBLISH permissions from the Sybase Central utility:

1 Right-click the current publisher.

2 Click Revoke Publish on the popup menu.

Notes on PUBLISH permissions

- Only one user ID on a database can hold PUBLISH permissions at one time. That user ID is called the publisher for the database.
- To see the publisher user ID in Sybase Central, open the SQL Remote folder; the publisher is shown on the right panel.
- To see the publisher user ID outside Sybase Central, use the CURRENT PUBLISHER special constant. The following statement retrieves the publisher user ID:

  ```
  SELECT CURRENT PUBLISHER
  ```

- If PUBLISH permissions is granted to a user ID with GROUP permissions, it is not inherited by members of the group.
- PUBLISH permissions carry no authority except to identify the publisher in outgoing messages.

- For messages sent from the current database to be received and processed by a recipient, the publisher user ID must have REMOTE or CONSOLIDATE permissions on the receiving database.
- The publisher user ID for a database cannot also have REMOTE or CONSOLIDATE permissions on that database. This would identify them as both the sender of outgoing messages and a recipient of such messages.
- A column with the DEFAULT CURRENT PUBLISHER setting will be filled in with the user ID of the current publisher. This is useful in ensuring unique primary keys throughout a replication system.

  For more information, see the section "Designing publications" on page 475.
- Changing the user ID of a publisher at a remote database will cause serious problems for any subscriptions that database is involved in, including loss of information. You should not change a remote database publisher user ID unless you are prepared to resynchronize the remote user from scratch.
- Changing the user ID of a publisher at a consolidated database while a SQL Remote setup is operating will cause serious problems, including loss of information. You should not change the consolidated database publisher user ID unless you are prepared to close down the SQL Remote setup and resynchronize all remote users.

Granting and revoking REMOTE and CONSOLIDATE permissions

REMOTE and CONSOLIDATE permissions are very similar. Each database receiving messages from the current database must have an associated user ID on the current database that is granted one of REMOTE or CONSOLIDATE permissions. This user ID represents the receiving database in the current database.

Databases directly below the current database on a SQL Remote hierarchy are granted REMOTE permissions, and the at most one database above the current database in the hierarchy is granted CONSOLIDATE permissions. The GRANT REMOTE and GRANT CONSOLIDATE statements identify the message system and address to which replication messages must be sent.

CONSOLIDATE permissions must be granted even at read-only remote databases, for the consolidated database, as receipt confirmations are sent back from the remote databases to the consolidated database. The GRANT CONSOLIDATE statement at remote databases is executed automatically by the database extraction utility.

Granting REMOTE and CONSOLIDATE permissions

Each remote database must be represented by a single user ID in the consolidated database. This user ID must be granted REMOTE permissions to identify their user ID and address as a subscriber to publications.

In the remote database, the publish and subscribe user IDs are inverted. The subscriber (remote user) in the consolidated database becomes the publisher in the remote database. The publisher of the consolidated database becomes a subscriber to publications from the remote database, and is granted CONSOLIDATE permissions.

The GRANT REMOTE statement identifies a user ID as a remote user, specifies a message type to use for exchanging messages with this user ID, and provides an address to where messages are to be sent. For example, the following statement grants remote permissions to user **S_Beaulieu**, using a MAPI e-mail system and with MAPI mail address **Beaulieu, S**:

```
GRANT REMOTE TO S_Beaulieu
TYPE mapi ADDRESS 'Beaulieu, S'
SEND AT '22:00'
```

The SEND AT clause is one of three alternatives that specify when messages should be sent to the remote user. The three alternatives are:

SEND EVERY 'HH: MM' A frequency can be specified in hours and minutes. When any user with SEND EVERY set is sent messages, all users with the same frequency are sent messages also. For example, all remote users who receive updates every twelve hours are sent updates at the same times, rather than being staggered. This reduces the number of times the transaction log has to be processed. You should use as few unique frequencies as possible.

SEND AT 'HH: MM' A time of day, in hours and minutes. Updates are started daily at the specified time. It is more efficient to use as few distinct times as possible than to stagger the sending times. Also, choosing times when the database is not busy minimizes interference with other users.

Default setting (no SEND clause) If any user has no SEND AT or SEND EVERY clause, the Message Agent sends messages every time it is run, and then stops. It runs in batch mode. SQL Remote is not intended for up-to-the-minute replication. Frequencies of less than ten minutes are not recommended.

The following statement grants remote user status to an office in Wakefield, U.K., using a user ID wakefield_office. The statement assumes that both offices are using MAPI e-mail systems as interfaces to internet e-mail, and that the e-mail system has an alias wakefield for the internet e-mail address.

```
GRANT REMOTE TO wakefield_office
TYPE mapi ADDRESS 'wakefield'
SEND EVERY '02:00'
```

The following statement grants REMOTE permissions using a file system to user S_Beaulieu.

```
GRANT REMOTE TO S_Beaulieu
TYPE file ADDRESS 'Beaulieu'
SEND AT '22:00'
```

The messages are placed in a subdirectory Beaulieu of the current working directory for the Message Agent. If the SQLREMOTE environment variable has been set, or if the Directory setting in the FILE message control parameters (held in the registry of INI files) has been set, the messages are stored under the named directory.

You can add a remote user to a database using Sybase Central:

1 Click the Users & Groups folder.

2 Select the user or users you wish to grant REMOTE permissions to, right-click to display a popup menu, and click Set Remote.

3 For each user, select the message type from the list, enter an address, choose the frequency of sending messages, and click OK to make the user a remote user.

At each remote database, the consolidated database must be granted CONSOLIDATE permissions. When you produce a remote database by running the database extraction utility, the GRANT CONSOLIDATE statement is executed automatically at the remote database.

The following statement grants CONSOLIDATE permissions to the hq_user user ID:

```
GRANT CONSOLIDATE TO hq_user
TYPE mapi ADDRESS 'hq_address'
```

There is no SEND clause in this statement. Messages will be sent to the consolidated database every time the Message Agent is run.

Revoking REMOTE and CONSOLIDATE permissions

By analogy with other permissions, REMOTE and CONSOLIDATE permissions can be revoked from a user using the REVOKE statement. The following statement revokes REMOTE permission from user S_Beaulieu.

```
REVOKE REMOTE FROM S_Beaulieu
```

DBA authority is required to revoke REMOTE or CONSOLIDATE access.

Revoking remote user permissions also drops any subscriptions for that user.

Setting up publications

The publication is the key design element of a SQL Remote data replication. A publication consists of a set of **articles**. Each article is a selection of rows and columns from a table in the database. Rows can be selected from a table in two ways; a WHERE clause can be included in an article definition, and/or a SUBSCRIBE BY expression can be supplied (usually a column name). In the latter case, rows are sent to subscribers according to a matching value supplied in their subscription.

This section describes the mechanics of setting up publications using Sybase Central. It also describes options open to you in designing publications, governed by the CREATE PUBLICATION statement that is executed by Sybase Central. Careful publication design is crucial to a properly functioning SQL Remote setup.

For information on designing publications, see "Designing publications" on page 475.

Publishing a set of tables

A simple publication consists of a set of tables, including all columns and rows in each of the tables. The following statement creates such a publication from the sample database:

```
CREATE PUBLICATION sales (
   TABLE customer,
   TABLE sales_order,
   TABLE sales_order_items,
   TABLE product
)
```

Not all the tables in the database have been published. For example, subscribers receive the sales_order table, but not the employee table.

Notes

- In the sample database, the sales_order and employee tables are related by a foreign key (sales_rep in the sales_order table is a foreign key to the emp_id column in the employee table). Although this could lead to referential integrity problems in the remote database, they are easily avoided by using the database extraction utility: for a discussion of this and other publication design issues, see "Designing publications" on page 475.

- A slightly different publication design (including an article that contains enough of the employee table to satisfy the foreign key relationship) would make the publication more robust; for this section we are publishing whole tables only.

In Sybase Central, you can add a publication to a database from within the SQL Remote folder.

1 Click the Publications folder, which is inside the SQL Remote folder.

2 Double-click Add Publication. The Publication Wizard is displayed.

3 Follow the instructions in the Wizard.

For more information, see the Sybase Central online Help.

Publishing a subset of columns

The product table keeps information such as unit price and current inventory for a set of products. One column lists the current inventory for each item. In the following single-table publication, this column is not replicated:

```
CREATE PUBLICATION product (
   TABLE product (   id,
                     name,
                     description,
                     size,
                     color,
                     unit_price
                     )
)
```

In a multi-table publication, any or all of the tables may have a subset of columns published.

In Sybase Central, the Publication Wizard guides you through selecting a subset of columns in an article.

Notes

- All columns in the primary key must be included in the publication.
- All columns used in WHERE clauses or SUBSCRIBE BY expressions must be included in the publication.
- You must not include a subset of columns in an article unless either:
- The remaining columns have default values or allow NULLs.

- No inserts are carried out at remote databases. Updates would not cause problems as long as they do not change primary key values. If you include a subset of columns in an article in situations other than these, INSERT statements at the consolidated database will fail.

Publishing a subset of rows using a WHERE clause

The following is a single-article publication sending relevant order information to Samuel Singer, a sales rep:

```
CREATE PUBLICATION pub_orders_samuel_singer (
   TABLE sales_order WHERE sales_rep = 856
)
```

In Sybase Central, the Publication Wizard guides you through creating a WHERE clause for an article.

The WHERE clause in a CREATE PUBLICATION statement may contain a subquery, but subqueries should only be used in very specific circumstances.

For a description of the circumstances where you can use a subquery in a publication, see "Using subqueries in publications " on page 480.

Publishing a subset of rows using a SUBSCRIBE BY expression

In a typical mobile workforce situation, a sales publication may be wanted where each sales rep subscribes to their own sales orders, enabling them to update their sales orders locally and replicate the sales to the consolidated database. Using the WHERE clause model, a separate publication for each sales rep would be needed: the following publication is for sales rep Samuel Singer: each of the other sales reps would need a similar publication.

```
CREATE PUBLICATION pub_orders_samuel_singer (
   TABLE sales_order WHERE sales_rep = 856
)
```

To address the needs of setups requiring large numbers of different subscriptions, SQL Remote allows an expression to be associated with an article. Subscriptions receive rows depending on the value of a supplied expression.

Publications using a SUBSCRIBE BY expression are more compact, easier to understand, and provide better performance than maintaining several WHERE clause publications. The database engine must add information to the transaction log, and scan the transaction log to send messages, in direct proportion to the number of publications. The SUBSCRIBE BY expression allows many different subscriptions to be associated with a single publication, whereas the WHERE clause does not.

The following statement creates a publication of the sales_order table with a SUBSCRIBE BY expression that is just the column sales_rep.

```
CREATE PUBLICATION pub_orders (
   TABLE sales_order SUBSCRIBE BY sales_rep
)
```

Each subscription to this publication specifies a value, using the syntax described in "Setting up subscriptions" on page 483. The subscriber receives updates of those rows for which the SUBSCRIBE BY expression (here a column) matches the value of the subscription parameter (the sales rep's employee id). For example, the following statement creates a subscription for Samuel Singer to the pub_orders publication:

```
CREATE SUBSCRIPTION
TO pub_orders ( '856' )
FOR SamS
```

Users can subscribe to more than one publication, and can have more than one subscription to a single publication.

For more information

- For information on how to use subqueries in a publication, see "Using subqueries in publications " on page 480.
- For more information on creating subscriptions, see "Setting up subscriptions" on page 483.
- In Sybase Central, the Publication Wizard guides you through creating a SUBSCRIBE BY expression for an article.

Dropping publications

Publications can be dropped using the DROP PUBLICATION statement. The following statement drops the publication named pub_orders.

```
DROP PUBLICATION pub_orders
```

Dropping a publication has the side effect of dropping all subscriptions to that publication.

Notes on publications

- The different publication types described above can be combined. A single publication can publish a subset of columns from a set of tables, and use both a WHERE clause to select a set of rows to be replicated and a SUBSCRIBE BY expression to partition rows by subscription.
- DBA authority is required to create publications.
- Publications can be altered and dropped by the DBA.
- Altering publications in a running SQL Remote setup is likely to cause replication errors, and could lead to loss of data.
- Views cannot be included in publications.
- Stored procedures cannot be included in publications. For a discussion of how SQL Remote replicates procedures and triggers, see "Replication of procedures and triggers" on page 490.
- For other considerations of referential integrity, see the section "Designing publications" on page 475.

Designing publications

SQL Remote allows data modification at any site in the system. This is a powerful and useful feature that some replication systems do not have. It does, however, raise the possibility of replication errors. For example, a user at one remote database could delete a row, and a remote user at another site update the same row. When these operations are replicated, if the DELETE arrives at the consolidated database first, the UPDATE will fail.

Proper design of publications can reduce the overhead required by the system and reduce the number of data modification errors and conflicts in the system.

SQL Remote can detect and resolve replication conflicts as long as the data being replicated conforms to certain good practices. Some important kinds of replication error must be avoided altogether with proper replication design. This section describes how to design publications so that important classes of replication errors are avoided.

A full discussion of conflict handling and error reporting is given in "Error reporting and conflict resolution in SQL Remote" on page 507.

Replication error and conflict overview

SQL Remote is designed to allow databases to be updated at many different sites. With a little planning, this can be made to work with few conflicts, but if some basic rules are not followed, replication errors can be a frequent and damaging occurrence. This section describes the kinds of conflict and error that can occur in a replication setup; subsequent sections describe how you can design your publications to avoid errors and manage conflicts.

There is one class of error that can and must be designed out of your SQL Remote publications to ensure that they do not occur. These are primary key errors. Replication errors and conflicts fall into the following categories:

Failures because of an INSERT Two users INSERT a row using the same primary key values. The second INSERT to reach a given database in the replication system will fail. This is an example of a primary key error, and must be avoided by proper design.

Failures because of a DELETE A user DELETES a row (that is, the row with a given primary key value). A second user UPDATES or DELETES the same row at another site. The second user's operation will not find the row, so the second statement fails.

Updates of primary keys An update of a primary key has similar effects to a DELETE followed by an INSERT, as far as conflicts are concerned. Any DELETE or UPDATE that refers to the old primary key values will not find the row, and any INSERT that refers to the new primary key will fail. This is an example of a primary key conflict, and must be avoided by proper design.

UPDATE conflicts A user updates a row. A second user UPDATES the same row at another site. The second user's operation succeeds, and SQL Remote allows a trigger to be fired to resolve these conflicts in a way that makes sense for the data being changed. These RESOLVE triggers are described in "RESOLVE UPDATE triggers" on page 509. Properly handled UPDATE conflicts are not a problem in SQL Remote.

Referential integrity errors If a column containing a foreign key is included in a publication, but the associated primary key is not included, the extraction utility leaves the foreign key definition out of the remote database so that INSERTS at the remote database will not fail.

Also, referential integrity errors can occur when a primary table has a SUBSCRIBE BY expression and the associated foreign table does not: rows from the foreign table may be replicated, but the rows from the primary table may be excluded from the publication.

For information about designing to avoid such conflicts, see "Designing to avoid referential integrity errors" on page 478.

Not all errors and conflicts can be designed out of a replication system. In particular, UPDATE conflicts will occur in many installations. SQL Remote allows appropriate resolution of UPDATE conflicts as part of the regular operation of a SQL Remote setup, using triggers.

For information about how SQL Remote handles conflicts as they occur, see "Error reporting and conflict resolution in SQL Remote" on page 507.

Primary key errors can and must be designed out of publications. This section describes how to design publications so that primary key errors do not occur.

Designing to avoid primary key errors

Primary key errors can be avoided if primary keys of tables that may be inserted at more than one site are guaranteed to be unique. This can be achieved by including a column identifying the site at which the INSERT is being made.

For example, a column with a default value of CURRENT PUBLISHER can be used. CURRENT PUBLISHER is the user ID that has been granted PUBLISH permissions on the current database. If such a column is included in the primary key, a row cannot be inserted at more than one site and primary key conflicts can be avoided.

Primary key errors are not corrected in SQL Remote, and designing them out of your publications is essential. Further, SQL Remote applications should not update primary key values.

Handling UPDATE conflicts

UPDATE conflicts occur when two users at different databases update the same column, that is not a primary key column, in a row.

Consider a publication that includes the contact table from the sample database. This table includes columns holding information about company contacts, such as the first and last name, the address, and phone and fax numbers.

If one user updates the street column for a particular contact, and another user updates the phone number, there is no conflict, and both updates will be replicated throughout the setup. Should you wish to detect and report such events, you can do so using the VERIFY_ALL_COLUMNS option, described in "RESOLVE UPDATE triggers" on page 509.

If two users update the street column for a particular contact, then a conflict will occur, as SQL Remote checks that the old values in the database where an update originates made match the values in the database where a replication is being applied.

SQL Remote allows you to define triggers to handle UPDATE conflicts so that data can be altered reliably at more than a single site; these conflict reporting and handling procedures are discussed in "Error reporting and conflict resolution in SQL Remote" on page 507. When an UPDATE conflict is detected at a consolidated database, the following sequence of events takes place.

1 Any RESOLVE UPDATE triggers defined for the operation are fired.

2 The UPDATE takes place.

3 Any actions of the trigger, as well as the UPDATE, are replicated to all remote databases, including the sender of the message that triggered the conflict.

4 At remote databases, no RESOLVE UPDATE triggers are fired when a message from a consolidated database contains an UPDATE conflict.

5 The UPDATE is carried out at the remote databases.

At the end of the process, the data is consistent throughout the setup.

UPDATE conflicts cannot happen where data is shared for reading, but each row (as identified by its primary key) is updated at only one site.

Designing to avoid referential integrity errors

The tables in a relational database are related through foreign key references. The referential integrity constraints applied as a consequence of these references ensure that the database remains consistent. If you wish to replicate only a part of a database, there are potential problems with the referential integrity of the replicated database.

By paying attention to referential integrity issues while designing publications you can avoid these problems. This section describes some of the more common integrity problems and suggests ways to avoid them.

Unreplicated referenced table errors

The sales publication described in "Setting up publications" on page 470 includes the sales_order table:

```
CREATE PUBLICATION pub_sales (
   TABLE customer,
   TABLE sales_order,
   TABLE sales_order_items,
   TABLE product
)
```

The sales_order table has a foreign key to the employee table. The id of the sales rep is a foreign key in the sales_order table referencing the primary key of the employee table. However, the employee table is not included in the publication.

If the publication is created in this manner, new sales orders would fail to replicate unless the remote database has the foreign key reference removed from the sales_order table.

If you use the extraction utility to create the remote databases, the foreign key reference is automatically excluded from the remote database, and this problem is avoided. However, there is no constraint in the database to prevent an invalid value from being inserted into the sales_rep_id column of the sales_order table, and if this happens the INSERT will fail at the consolidated database. To avoid this problem, you can include the employee table (or at least its primary key) in the publication.

Unreplicated row integrity errors

The pub_orders publication described in "Publishing a subset of rows using a SUBSCRIBE BY expression" on page 472 includes the rows containing their own sales orders, for each sales rep:

```
CREATE PUBLICATION pub_orders (
   TABLE sales_order SUBSCRIBE BY sales_rep
)
```

The following publication also includes all line items from these orders, but will cause problems in replication.

```
CREATE PUBLICATION pub_orders (
   TABLE sales_order SUBSCRIBE BY sales_rep,
   TABLE sales_order_items
)
```

The primary key in the sales_order table is id, an order ID number. The sales_order_items table includes the sales order ID as a foreign key to the sales_order table. Although, for example, Samuel Singer receives only his own rows from the sales order table (sales_rep = 856), he receives all the rows of the sales_order_items table.

When another sales rep adds an order, the new sales_order_items entries are replicated to Samuel Singer. The replication fails as Samuel Singer's database does not have a primary key corresponding to the foreign key of the new sales_order_items entries.

One way of of avoiding the problem is to include the sales_rep column in the sales_order_items table. In this case, triggers can be used to fill in the proper values of the user ID in the sales_order_items table. An example is given in the SALESPUB.SQL script included in your installation directory.

Under certain restricted conditions only, you could also use a subquery in a SUBSCRIBE BY expression to solve the problem.

For information on using subqueries in SUBSCRIBE BY expressions, see the section "Using subqueries in publications " on page 480.

Designing triggers to avoid errors

Actions performed by triggers are not replicated: triggers that exist at one database in a SQL Remote setup are assumed by the replication procedure to exist at other databases in the setup. When an action that fires a trigger at the consolidated database is replicated at the replicate site, the trigger is automatically fired. By default, the database extraction utility replicates the trigger definitions, so that they are in place at the remote database also.

If a publication includes only a subset of a database, a trigger at the consolidated database may refer to tables or rows that are present at the consolidated database, but not at the remote databases. You can design your triggers to avoid such errors by making actions of the trigger conditional using an IF statement. The following list suggests some ways in which triggers can be designed to work on consolidated and remote databases.

- Have actions of the trigger be conditional on the value of CURRENT PUBLISHER. In this case, the trigger would not execute certain actions at the remote database.
- Have actions of the trigger be conditional on the object_id function not returning NULL. The object_id function takes a table or other object as argument, and returns the ID number of that object or NULL if the object does not exist.
- Have actions of the trigger be conditional on a SELECT statement which determines if rows exist.

The RESOLVE UPDATE trigger is a special trigger type for the resolution of UPDATE conflicts, and is discussed in the section "UPDATE conflict resolution examples" on page 510. The actions of RESOLVE UPDATE triggers are replicated to remote databases, including the database that caused the conflict.

Using subqueries in publications

Subqueries can be used in publications, but require some extra work in order to maintain the integrity of subscriptions.

If no special measures are taken, rows in tables that use subqueries will not be removed from or added to subscriptions properly. This is illustrated by example,and the solution to this problem is then outlined.

Why subqueries are useful

The sample database includes a sales_order table and a sales_order_items table. The sales_order_items table has the order id number as part of its primary key, and also as a foreign key to the sales_order table. The sales_order table contains a region column that is not present in the sales_order_items table.

A publication that replicated the rows in sales_order to subscribers by region may include the following article:

```
CREATE PUBLICATION SalesPub (
   TABLE sales_order SUBSCRIBE BY region
   ...
)
```

If this publication was to replicate the rows in the sales_order_items table also, only those items corresponding to orders for the region should be replicated. The following statement shows a subquery that accomplishes this.

```
CREATE PUBLICATION SalesPub (
   TABLE sales_order SUBSCRIBE BY region
   TABLE sales_order_items SUBSCRIBE BY
      (  SELECT region
         FROM sales_order
         WHERE sales_order.id = sales_order_items.id)
   ...
)
```

Potential problems with subqueries

There is a potential problem with a subquery SUBSCRIBE BY expression. If the region in a row in the sales_order table is updated, so that the row is removed from a remote user's subscription, that UPDATE is replicated to the remote user as a DELETE, to remove the row from the subscriber's table (see "Replication of updates" on page 489).

However, the corresponding rows in sales_order_items, which are not affected by the UPDATE, remain in the tables at the remote database. SQL Remote replicates changes as recorded in the transaction log, and nothing in the sales_order_items has changed, so there are no log entries.

Maintaining subscriptions when subqueries are used

SQL Anywhere provides a special syntax of the UPDATE statement in order to allow subqueries to be used. The syntax is as follows:

UPDATE *table*
...**PUBLICATION** *publication*
...**SUBSCRIBE BY** *expression*
...**WHERE** *search-condition*

An UPDATE statement of this form must be executed in a BEFORE trigger for any operation that has a side effect of changing a related table's SUBSCRIBE BY values. In this statement, expression is typically a value or a subquery.

The following trigger solves the updated region problem in the example above:

```
CREATE TRIGGER MoveItems
BEFORE UPDATE OF region
ON sales_order
REFERENCING NEW AS newrow
FOR EACH ROW
BEGIN
   UPDATE sales_order_items PUBLICATION SalesPub
   SUBSCRIBE BY newrow.region
   WHERE sales_order_items.id = newrow.id;
END
```

The UPDATE statement does not make any alteration to the table itself, but places entries in the transaction log which the Message Agent uses to adjust the rows that are present in remote databases.

Incorrect triggers cause replication errors

If the trigger is not correct, replication errors result. If the incorrect SUBSCRIBE BY value is given, the rows are moved to the wrong remote databases.

An alternative solution is to duplicate the region column in the sales_order_items table, and maintain the value in the column using a trigger. In this way, applications do not have to be aware of the extra column. The sample publication built in the SALESPUB.SQL command file in your SQL Anywhere installation directory uses this approach.

SUBSCRIBE BY subqueries returning multiple values

Subqueries in SUBSCRIBE BY clauses can return more than one value. This is useful because it allows SUBSCRIBE BY publications that are not partitioned into disjoint subsets.

One situation where this feature is useful is where two tables have a many-to-many relationship, and both tables need to have only a subset of rows in the remote databases.

Multiple value SUBSCRIBE BY queries must be used in conjunction with the special UPDATE statement described in the previous section, in order to maintain subscriptions properly.

Setting up subscriptions

To subscribe to a publication, each subscriber must be granted REMOTE permissions (see "Granting and revoking REMOTE and CONSOLIDATE permissions" on page 466) and a subscription must also be created for that user. The following statement creates a subscription for a user ID SamS to the pub_orders_samuel_singer publication, which was created using a WHERE clause:

```
CREATE SUBSCRIPTION
TO pub_orders_samuel_singer
FOR SamS
```

The following statement creates a subscription for Samuel Singer (user ID SamS, employee ID 856) to the pub_orders publication, defined with a SUBSCRIBE BY expression sales_rep, requesting the rows for Samuel Singer's own sales:

```
CREATE SUBSCRIPTION
TO pub_orders ( '856' )
FOR SamS
```

In order to receive and apply updates properly, each subscriber needs to have an initial copy of the data. The synchronization process is discussed in "Synchronizing databases" on page 484.

Synchronizing databases

SQL Remote replication is carried out using the information in the transaction log, but there are two circumstances where 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 site. This process is called **synchronization**.

Synchronization is used under the following circumstances:

- When a subscription is created at a consolidated database a synchronization is carried out, so that the remote database starts off with a database in the same state as the consolidated database.
- If a remote database gets corrupt or gets out of step with the consolidated database, and cannot be repaired using SQL passthrough mode, synchronization forces the remote site database back in step with the consolidated site.

Synchronizing a remote database can be done in the following ways:

- Use the database extraction utility to synchronize the remote database. This is generally the recommended procedure.
- Synchronize the remote database manually by loading from files, using the PowerBuilder pipeline, or some other tool.
- Synchronize the remote database via the message system using the SYNCHRONIZE SUBSCRIPTION statement.

You should not execute SYNCHRONIZE SUBSCRIPTION at a remote database.

Using the extraction utility to synchronize databases

The extraction utility can be accessed from Sybase Central, or as the DBXTRACT command-line utility.

The DBXTRACT command-line utility unloads a database schema and data suitable for building a remote database for a named subscriber. It produces a SQL command file with default name RELOAD.SQL and a set of data files. To create the remote database you must:

1 Create a database using Sybase Central or using the DBINIT utility.

2 Connect to the database from the ISQL utility, and run the RELOAD.SQL command file. The following statement entered in the ISQL command window runs the RELOAD.SQL command file: read *path*\RELOAD.SQL where *path* is the path of the reload command file.

When used from Sybase Central, the extraction utility carries out the database unloading task, in the same way that DBXTRACT does, and then takes the additional step of creating the new database.

The extraction utility does not use a message system. The reload file (DBXTRACT) or database (from Sybase Central) is created in a directory accessible from the current machine. Synchronizing many subscriptions over a message link can produce heavy message traffic and, if the message system is not completely reliable, it may take some time for all the messages to be properly received at the remote sites.

You must complete the following tasks before using the extraction utility at a consolidated database.

- You must have created message types for replication.
- You must have added a publisher user ID to the database.
- You must have added remote users to the database.
- You must have added the publication to the database.
- You must have created a subscription for the remote users.

For a description of how to carry out these steps, see the tutorial in the chapter "Introduction to SQL Remote Replication".

When you use the extraction utility to create a remote database, the user for which you are creating the database receives the same permissions they have in the consolidated database. Further, if the user is a member of any groups on the consolidated database, those group IDs are created in the remote database with the permissions they have in the consolidated database.

Using the extraction utility from Sybase Central

For full information on using the extraction utility from Sybase Central, see the Sybase Central online Help. This section describes one way to extract a database for a remote user from the current consolidated database.

To extract a database for a remote user:

1 Click the Remote Users folder on the left panel, which is in the SQL Remote folder. The right panel displays the remote users.

2 Right-click the remote user for whom you wish to extract a database, and select Extract Database from the popup menu. The Extraction Wizard is displayed.

3 Follow the instructions in the Wizard. The following section describes the options available.

For more information

For information on the DBXTRACT command-line options, see "The DBXTRACT command-line utility" in the chapter "SQL Anywhere Components" and the following section.

For information about the extraction utility options, available as command-line options for DBXTRACT or as choices presented by the Database Extraction Wizard, see "Extraction utility options" in the chapter "SQL Anywhere Components".

Limits to using the extraction utility

While the extraction utility is the recommended way of creating and synchronizing remote databases from a consolidated databases, there are some circumstances where it cannot be used, and you must synchronize remote databases manually. This section describes some of those cases.

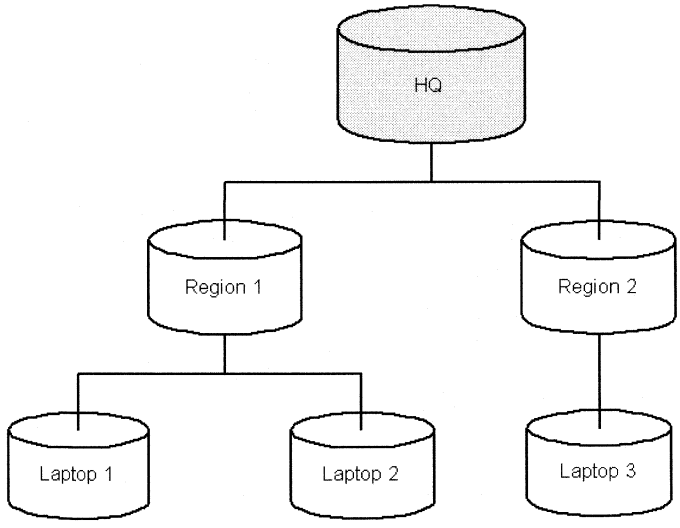
Additional tables at the remote database

Remote databases can have tables not present at their consolidated database as long as these tables do not take part in replication. Of course, the extraction utility cannot extract such tables from a consolidated database.

Using the extraction utility in multi-tiered setups

To understand the role of the extraction utility in multi-tiered arrangements, consider a three-tiered SQL Remote setup.

This setup is illustrated in the following diagram.



From the consolidated database at the top level, you can use the extraction utility to create the second-level databases. You can then add remote users to these second-level databases, and use the extraction utility from each second-level database to create the remote databases. However, if you have to reextract the second-level databases from the top-level consolidated database, you will delete the remote users that were created, along with their subscriptions and permissions, and will have to rebuild those users. The exception is if you resynchronize data only, in which case you can use the extraction utility to replace the data in the database, without replacing the schema.

Synchronizing data over a message system

A subscription is created at a consolidated database using the CREATE SUBSCRIPTION statement:

```
CREATE SUBSCRIPTION
TO [owner].pubname [ ( string ) ]
FOR userlist
```

The CREATE SUBSCRIPTION statement defines the data to be received. It does not synchronize or start a subscription.

The SYNCHRONIZE SUBSCRIPTION statement causes the Message Agent (DBREMOTE) to send a copy of all rows in the subscription to the subscriber. It assumes that an appropriate database schema is in place.

The SYNCHRONIZE SUBSCRIPTION statement has the following syntax:

```
SYNCHRONIZE SUBSCRIPTION
TO [owner].pubname [ ( string ) ]
FOR userlist
```

When synchronization messages are received at a subscriber database, the Message Agent replaces the current contents of the database with the new copy. Any data at the subscriber that is part of the subscription, and which has not been replicated to the consolidated database, is lost. Once synchronization is complete, the subscription is started by the Message Agent using the START SUBSCRIPTION statement. The syntax of the START SUBSCRIPTION statement is as follows:

```
START SUBSCRIPTION
TO [owner].pubname [ ( string ) ]
FOR userlist
```

If a remote database becomes out of step with the consolidated database, and cannot be brought back in step using the SQL passthrough capabilities of SQL Remote, the SYNCHRONIZE SUBSCRIPTION statement forces the remote database into step with the consolidated database by copying the rows of the subscription from the consolidated database over the contents at the remote database.

Data loss on synchronization
Any data in the remote database that is part of the subscription, but which has not been replicated to the consolidated database, is lost when the subscription is synchronized. You may wish to unload or back up the remote database using Sybase Central or the DBUNLOAD utility before synchronizing the database.

Notes on synchronization

Synchronizing large numbers of subscriptions, or synchronizing subscriptions to large, frequently-used tables, can slow down database access for other users. You may wish to synchronize such subscriptions when the database is not in heavy use. This happens automatically if you use a SEND AT clause with a quiet time specified.

Synchronization applies to an entire subscription. There is currently no straightforward way of synchronizing a single table.

How statements are replicated by SQL Remote

When a simple INSERT statement is entered at one database, it is sent to other databases in the SQL Remote setup as an INSERT statement. However, not all statements are replicated exactly as they are entered by the client application. This section describes how SQL Remote replicates SQL statements.

Replication of inserts and deletes

INSERT and DELETE statements are the simplest replication case. SQL Remote takes each INSERT or DELETE operation from the transaction log, and sends it to all sites that subscribe to the row being inserted or deleted.

If only a subset of the columns in the table is subscribed to, the INSERT statements sent to subscribers contains only those columns.

Replication of updates

UPDATE statements are not replicated exactly as the client application enters them. This section describes two ways in which the replicated UPDATE statement may differ from the entered UPDATE statement.

Inclusion of a VERIFY clause for conflict detection

SQL Remote replicates UPDATE statements including a VERIFY clause in the statement.

Consider the following UPDATE statement, entered by an application at a remote database.

```
UPDATE "dba".product
SET quantity = 23
WHERE id = 300
```

When this UPDATE is sent to the consolidated database, SQL Remote adds a VERIFY clause to the statement, as follows:

```
UPDATE "dba".product
SET quantity = 23
VERIFY ( quantity )
VALUES ( 28 )
WHERE id = 300
```

The VERIFY clause is a check to ensure that the values being updated at the consolidated database are the same as those that were updated at the remote database.

In the example, the UPDATE statement succeeds without a conflict at the consolidated database only if the quantity column has a value of 28 in the row being updated, before the update takes place. If the value is not 28, then any RESOLVE UPDATE trigger created for the table is fired before the UPDATE is applied. The VERIFY clause allows UPDATE conflicts to be detected and resolved by RESOLVE UPDATE triggers.

For more information on resolution of UPDATE conflicts, see "UPDATE conflict resolution examples" on page 510.

You can include all columns in the VERIFY clause using the VERIFY_ALL_COLUMNS option, described in "RESOLVE UPDATE triggers" on page 509.

UPDATE statements replicated as INSERTS or DELETES

If an UPDATE statement has the effect of removing a row from a given remote user's subscription, it is sent to that user as a DELETE statement. If an UPDATE statement has the effect of adding a row to a given remote user's subscription, it is sent to that user as an INSERT statement.

For example, the pub_orders publication is defined as follows:

```
CREATE PUBLICATION pub_orders (
   TABLE sales_order SUBSCRIBE BY sales_rep
)
```

Samuel Singer, with user ID SamS, has a subscription to the pub_orders publication as follows:

```
CREATE SUBSCRIPTION
TO pub_orders ( '856' )
FOR SamS
```

Pamela Savarino, with user ID PamS, has a subscription to the pub_orders publication as follows:

```
CREATE SUBSCRIPTION
TO pub_orders ( '949' )
FOR PamS
```

An UPDATE to the sales_order table that changes the sales_rep value of a row from 856 to 949 is replicated to Samuel Singer as a DELETE statement, and to Pamela Savarino as an INSERT statement.

Replication of procedures and triggers

Any replication system is faced with two broad choices when replicating a procedure call. Either the call itself can be replicated (so that a procedure with the same name and parameters is called at the replicate site), or the procedure can be replicated by replicating the individual actions (INSERTs, UPDATEs, DELETEs) of the procedure.

When operations are applied by the Message Agent, the same triggers fired at the publishing database are fired at the subscriber database. If the actions taken by a trigger at the publishing database were included in replication messages, these actions would occur twice.

The current release of SQL Remote does not allow procedures to be included in publication definitions, and so implements the second of the two choices; to replicate the actions of the procedure, not the call. This is described in the following section.

Replication of procedures

SQL Remote replicates procedures by replicating the actions of a procedure. The call itself is not reproduced. Consequently, procedures that modify non-replicated data do not cause problems. If the call to a procedure were replicated, a version of the procedure would have to be written that does not carry out those actions that modify unreplicated data, for the replicate database.

Replication of triggers

SQL Remote does not replicate actions performed by triggers. SQL Remote replicates triggers by assuming the trigger is defined remotely. This avoids permissions issues and the possibility of each action occurring twice. There are some exceptions to this rule, described below.

RESOLVE UPDATE trigger actions

The actions carried out by RESOLVE UPDATE triggers are replicated from a consolidated database to all remote databases, including the one that sent the message causing the conflict.

Replication of BEFORE triggers

Some BEFORE triggers can produce undesirable results when using SQL Remote. For example, a BEFORE UPDATE that bumps a counter column in the row to keep track of the number of times a row is updated will double count, because the BEFORE UPDATE trigger will fire when the UPDATE is replicated. Also, a BEFORE UPDATE that sets a column to the time of the last update will get the time the UPDATE is replicated as well.

An option to replicate trigger actions

The Message Agent has a command-line switch that causes it to replicate all trigger actions. This is the DBREMOTE -t switch.

If you do use this switch, you must ensure that the trigger actions are not carried out twice at remote databases, once by the trigger being fired at the remote site, and once by the explicit application of the replicated actions from the consolidated database.

To ensure that trigger actions are not carried out twice, you can wrap an IF CURRENT REMOTE USER IS NULL ... END IF statement around the body of the triggers.

Replication of data definition statements

Only data manipulation statements are replicated to other databases. Data definition statements (CREATE, ALTER, DROP, and others that modify database objects) are not replicated by SQL Remote unless they are entered while in PASSTHROUGH mode.

For information about PASSTHROUGH mode, see "Using passthrough mode for administration" on page 513.

Replication of blobs

SQL Remote includes special methods for replication of LONG VARCHAR, LONG BINARY, TEXT, and IMAGE data types: values that are longer than 256 characters.

The Message Agent uses a variable in place of the value in the INSERT or UPDATE statement that is being replicated. The value of the variable is built up by a sequence of statements of the form

```
SET vble = vble || 'more_stuff'
```

This makes the size of the SQL statements involving long values smaller, so that they fit within a single message. The SET statements are separate SQL statements, so that the "blob" is effectively split over several SQL Remote messages.

Using the VERIFY_THRESHOLD option to minimize message size

The Verify_threshold database option can prevent long values from being verified (in the VERIFY clause of a replicated UPDATE). The default value for the option is 1000. If the data type of a column is longer than the threshold, old values for the column are not verified when an UPDATE is replicated. This keeps the size of SQL Remote messages down, but has the disadvantage that conflicting updates of long values are not detected.

There is a technique allowing detection of conflicts when verify_threshold is being used to reduce the size of messages. Whenever a "blob" is updated, a last_modified column in the same table should also be updated. Conflicts can then be detected because the old value of the last_modified column is verified.

Using a work table to avoid redundant updates

Repeated updates to a blob should be done in a "work" table, and the final version should be assigned to the replicated table. For example, if a document in progress is updated 20 times throughout the day and the Message Agent is run once at the end of the day, all 20 updates are replicated. If the document is 200Kb in length, this causes 4Mb of messages to be sent. The better solution is to have a *document in progress* table. When the user is done revising a document, the application moves it from the *document in progress* table to the replicated table. The results in a single update (200Kb of messages).

Managing a running SQL Remote setup: overview

When you are in the design and setup phase, you can alter many facets of the SQL Remote setup. Altering publications, message types, writing triggers to resolve update conflicts are all easy to do.

Once you have deployed a SQL Remote application, the situation is different. A SQL Remote setup can be seen as a single dispersed database, spread out over many sites, maintaining a loose form of consistency. The data may never be in exactly the same state in all databases in the setup at once, but all data changes are replicated as complete transactions around the system over time. Consistency is built in to a SQL Remote setup through careful publication design, and through the reconciliation of UPDATE conflicts as they occur.

Upgrading and resynchronization

Once a SQL Remote setup is deployed and is running, it is not easy to tinker with. An upgrade to a SQL Remote installation needs to be carried out with the same care as an initial deployment.

Making changes to a database schema at one database within the system will cause failures because of incompatible database objects. The passthrough mode does allow schema changes to be sent to some or all databases in a SQL Remote setup, but must still be used with care and planning.

The loose consistency in the dispersed database means that updates are always in progress: you cannot stop changes being made to the database, make some changes to the database schema, and restart.

Many changes to a database schema will produce conflicts throughout the installation, and will require all subscriptions to be stopped and resynchronized. Resynchronization involves loading new copies of the data in each remote database, and for more than a few subscribers is a time-consuming process involving work interruptions and possible loss of data.

The following are examples of changes that should not be made to a deployed and running SQL Remote setup:

- Change the publisher for the consolidated database.
- Make restrictive changes to tables, such as dropping a column or altering a column to not allow NULL values. Changes including the column or including NULL entries may already be being sent in messages around the SQL Remote setup, and will fail.

- Alter a publication. Publication definitions must be maintained at both local and remote sites, and changes that rely on the old publication definition may already be being sent in messages around the SQL Remote setup.
- DROP SUBSCRIPTION.
- Unload and reload a database. If a database is participating in replication, it cannot be unloaded and reloaded without re-synchronizing the database. Replication is based on the transaction log, and when a database is unloaded and reloaded, the old transaction log is no longer available. For this reason, good backup practices are especially important when participating in replication.

Running the SQL Remote Message Agent

The SQL Remote Message Agent (DBREMOTE.EXE, or DBREMOTW.EXE under Window 3.x) is a key component in SQL Remote replication. It carries out the following functions:

- It scans the transaction log at each publisher database, and translates the log entries into messages for subscribers.
- It parcels the log entries up into messages no larger than a fixed maximum size (50,000 bytes by default), and sends them to subscribers.
- It processes incoming messages, and applies them in the proper order to the database.
- It maintains the message tracking information in the system tables, and manages the guaranteed transmission mechanism.

The Message Agent can be run in either batch or continuous mode. The options available depend on the sending frequency specified in the GRANT REMOTE statements for remote users.

- If any remote user does not have SEND AT or SEND EVERY set in the GRANT REMOTE statement, the Message Agent can run only in batch mode. In this mode, it processes incoming messages, scans the transaction log once, and stops. In all supported operating systems other than Windows or Windows NT, the Message Agent closes. In Windows or Windows NT, you can run the Message Agent with the -k command-line switch to close the window automatically after scanning the transaction log.
- If all remote users have SEND AT or SEND EVERY set, the Message Agent runs continuously by default. It can be run in batch mode by specifying the -b command-line switch. After the Message Agent processes incoming messages, it determines which remote users should now be sent messages. If any users are due for messages, they are sent. When no users are due for messages, the Message Agent waits. While the Message Agent is waiting, it checks frequently for incoming messages, and applies them when they arrive.

Running the Message Agent continuously can be useful particularly at a consolidated database, where messages may be coming in frequently and many outgoing messages may need to be sent. If you are running SQL Remote on Windows NT, you can run the Message Agent as an NT service.

For information on using NT services, see the chapter "Running Programs as Services".

Running the Message Agent in batch mode is most likely useful at a remote database. For example, an application could run the Message Agent once per day at a remote site. You can choose to process only incoming or outgoing messages using the -r and -s command-line switches, respectively.

For a full description of DBREMOTE command-line switches, see "The SQL Remote Message Agent" in the chapter "SQL Anywhere Components".

The Message Agent and replication security

In the tutorials in the previous chapter, the Message Agent was run using a user ID with DBA permissions. The operations in the messages are carried out from the user ID specified in the Message Agent connection string; by using the user ID DBA, you can be sure that the user has permissions to make all the changes.

In many situations, distributing the DBA user ID and password to all remote database users is an unacceptable practice for security and data privacy reasons. SQL Remote provides a solution that enables the Message Agent to have full access to the database in order to make any changes contained in the messages without creating security problems.

A special permission, REMOTE DBA, has the following properties:

- A user ID granted REMOTE DBA authority has no extra privileges on any connection apart from the Message Agent. Therefore, even if the user ID and password for a REMOTE DBA user is widely distributed, there is no security problem. As long as the user ID has no permissions beyond CONNECT granted on the database, no one can use this user ID to access data in the database.
- When connecting from the Message Agent, a user ID with REMOTE DBA authority has full DBA permissions on the database.

A suggested practice is to grant REMOTE DBA authority at the consolidated database to the publisher and to each remote user. When the remote database is extracted, the remote user becomes the publisher of the remote database, and is granted the same permissions they were granted on the consolidated database, including the REMOTE DBA authority which enables them to use this user ID in the Message Agent connection string. Adopting this procedure means that there are no extra user IDs to administer, and each remote user needs to know only one user ID to connect to the database, whether from the Message Agent (which then has full DBA authority) or from any other client application (in which case the REMOTE DBA authority grants them no extra permissions).

You can grant REMOTE DBA permissions to a user ID named dbremote as follows:

```
GRANT REMOTE DBA
TO dbremote
IDENTIFIED BY dbremote
```

In SQL Anywhere, you can add the REMOTE DBA authority to a remote user by checking the appropriate option in the New Remote User Wizard.

Running the Message Agent when recovery is difficult

In this section, an option for running the Message Agent in situations where database recovery is complicated, or backups are insecure, is described. The -u DBREMOTE command-line option causes the Message Agent to process only those transactions before the latest backup.

As the Message Agent relies on entries in the transaction log, transaction log management is an important part of SQL Remote administration. For a discussion of transaction log management, see "Transaction log and backup management for SQL Remote", on page 503.

When the -u option is used, outgoing transactions and confirmation of incoming transactions are not sent until they are no longer in the live transaction log. This feature is useful in several situations:

- **No transaction log mirror on the consolidated database** It is strongly recommended that a transaction log mirror be used on a SQL Remote consolidated database. In the absence of a mirror, media failure on the transaction log might result in loss of transactions.

  If the -u option is used, recovery in such a situation will not require remote users' databases to be re-extracted, as no transactions have been replicated to remote users until the transactions have been backed up.

- **Offsite backup is required for protection against a total site failure** In this case the transaction log may be backed up across a WAN to another site.

  If the -u option is used, and there is a total site failure, recovery will not require remote users to be re-extracted because no transactions have been replicated to remote users until the transactions have been backed up to another site.

- **Participation in Replication Server replication** In this case, the recovery plan is complicated because it needs to recover to a point where all databases are consistent.

If there is a possibility of an enterprise SQL Server losing committed transactions on recovery (for example,. not using SQL Server transaction log mirroring), then the SQL Anywhere database needs to be restored to a point consistent with the enterprise (throwing away the current transaction log). In this case, it is desirable to prevent remote databases from receiving transactions until they have been backed in a way that they will not be lost.

The SQL Remote message tracking system

SQL Remote has a message tracking system to ensure that all replicated operations are applied in the correct order, no operations are missed, and no operation is applied twice.

Message system failures may lead to replication messages not reaching their destination, or reaching it in a corrupt state. Also, messages may arrive at their destination in a different order from that in which they were sent. This section describes the SQL Remote system for detecting and correcting message system errors, and for ensuring correct application of messages.

If you are using an e-mail message system, you should confirm that e-mail is working properly between the two machines if SQL Remote messages are not being sent and received properly.

The SQL Remote message tracking system is based on status information maintained in the SYSREMOTEUSER system table. The table is maintained by the Message Agent. The Message Agent at a subscriber database sends confirmation to the publisher database to ensure that SYSREMOTEUSER is maintained properly at each end of the subscription.

Status information in the SYSREMOTEUSER system table

The SYSREMOTEUSER system table contains a row for each subscriber, with status information for messages sent to and received by that subscriber. SYSREMOTEUSER at the consolidated database contains a row for each remote user. SYSREMOTEUSER at each remote database contains a single row maintaining information for the consolidated database. (Recall that the consolidated database subscribes to publications from the remote database.)

The SYSREMOTEUSER table at each end of a subscription is maintained by the Message Agent.

Tracking messages by transaction log offsets

The message-tracking status information takes the form of offsets in the transaction logs of the publisher and subscriber databases. Each COMMIT is marked in the transaction log by a well-defined offset. The order of transactions can be determined by comparing their offset values.

When messages are sent, they are ordered by the offset of the last COMMIT of the preceding message. If a transaction spans several messages, there is a serial number within the transaction to order the messages correctly. The default maximum message size is 50,000 bytes, but you can use the DBREMOTE -l switch to change this setting.

The log_sent column holds the local transaction log offset for the latest message sent to the subscriber. When the Message Agent sends a message, it sets the log_sent value to the offset of the last COMMIT in the message. Once the message has been received and applied at the subscribed database, confirmation is sent back to the publisher. When the publisher Message Agent receives the confirmation, it sets the confirm_sent column for that subscriber with the local transaction log offset. Both log_sent and confirm_sent are offsets in the local database transaction log, and confirm_sent cannot be a later offset than log_sent.

When the Message Agent at a subscriber database receives and applies a replication update, it updates the log_received column with the offset of the last COMMIT in the message. The log_received column at any subscriber database therefore contains a transaction log offset in the publisher database's transaction log. After the operations have been received and applied, the Message Agent sends confirmation back to the publisher database and also sets the confirm_received value in the local SYSREMOTEUSER table. The confirm_received column at any subscriber database contains a transaction log offset in the publisher database's transaction log.

SQL Remote subscriptions are two-way operations: each remote database is a subscriber to publications of the consolidated database and the consolidated database subscribes to a matching publication from each remote database. Therefore, the SYSREMOTEUSER tables at the consolidated and remote database hold complementary information.

The Message Agent applies transactions and updates the log_received value atomically. If a message contains several transactions, and a failure occurs while a message is being applied, the log_received value corresponds exactly to what has been applied and committed.

The SYSREMOTEUSER table contains two other columns that handle resending messages. The resend_count and rereceive_count columns are retry counts that are incremented when messages get lost or deleted for some reason.

The log_send column is updated by the Message Agent based on the SEND frequency information (SEND AT or SEND EVERY as specified in the GRANT REMOTE statement). The value of log_send must be greater than that of log_sent for a user in order for messages to be sent to that user.

Handling of lost or corrupt messages

When messages are received at a subscriber database, the Message Agent applies them in the correct order (determined from the log offsets) and sends confirmation to the publisher. If a message is missing, the Message Agent increments the local value of rereceive_count, and requests that it be resent. Other messages present or en route are not applied.

The request from a subscriber to resend a message increments the resend_count value at the publisher database, and also sets the publisher's log_sent value to the value of confirm_sent. This resetting of the log_sent value causes operations to be resent.

Users cannot reset log_sent
The log_sent value cannot be reset by a user, as it is in a system table.

Each message is identified by three values:

- Its resend_count.
- The transaction log offset of the last COMMIT in the previous message.
- A serial number within transactions, for transactions that span messages.

Messages with a resend_count value smaller than rereceive_count are not applied; they are deleted. This ensures that operations are not applied more than once.

Transaction log and backup management for SQL Remote

The importance of good backup practices

Replication depends on access to operations in the transaction log, and access to old transaction logs is sometimes required. This section describes how to set up backup procedures at the consolidated and remote databases to ensure proper access to old transaction logs.

It is crucial to have good backup practices at SQL Remote consolidated database sites. A lost transaction log could easily mean having to resynchronize remote users. At the consolidated database site, a transaction log mirror is recommended.

For information on transaction log mirrors and other backup procedure information, see the chapter "Backup and Data Recovery".

Access to old transactions

In many setups, users of laptop databases may receive updates from the office server every day or so. If some messages get lost or deleted, and have to be resent by the message-tracking system, it is possible that changes made several days ago will be required. If a remote user takes a vacation, and messages have been lost in the meantime, changes weeks old may be required. If the transaction log is backed up daily, the log with the changes will no longer be running on the server.

You can specify a directory on the DBREMOTE command line. This directory is where old transaction logs are kept. This section describes how you can set up a backup procedure to ensure that such a directory is kept in proper shape.

Backup options to use

One of the options to the backup utility is to rename the transaction log on backup and restart. For the DBBACKUP command-line utility, this is the -r command-line switch. It is recommended that you use this option when backing up the consolidated database and remote database transaction logs.

To see how this option works, consider a consolidated database named HQ.DB, in directory C:\HQ, with a transaction log in directory D:\HQLOG\HQ.LOG. Backing up this transaction log to a directory E:\HQBAK using the rename and restart option carries out the following tasks:

1. Backs up the transaction log, creating a backup file E:\HQBACK\HQ.LOG.
2. Renames the existing transaction log to D:\HQLOG\HQ.LNN, where *nn* is the lowest available integer, starting at 00.
3. Starts a new transaction log, as D:\HQLOG\HQ.LOG.

After several backups, the directory D:\HQLOG will contain a set of sequential transaction logs.

You can run the Message Agent with access to these log files using the following command line:

```
dbremote -c "dbn=hq;..." d:\hqlog
```

The log directory should not contain any transaction logs other than the sequence of logs generated by this backup procedure.

You can also run the Message Agent pointing to the directory where backup copies are held. However, the backup utility makes backups to the same file name each time by default, so you will need to ensure that old logs are renamed before subsequent backups.

Using the DELETE_OLD_LOGS option

The DELETE_OLD_LOGS database option is set by default to OFF. If it is set to ON, then the old transaction logs will be deleted automatically by the Message Agent when they are no longer needed. A log is no longer needed when all subscribers have confirmed receiving all changes recorded in that log file. This option can help to manage disk space in replication setups.

You can set the DELETE_OLD_LOGS option either for the PUBLIC group or just for the user contained in the Message Agent connection string.

Backup procedures at remote databases

Backup procedures are not as crucial at remote databases as at the consolidated database. You may choose to rely on replication to the consolidated database as a data backup method. In the event of a media failure, the remote database would have to be re-extracted from the consolidated database, and any operations that have not been replicated would be lost. (You could use the log translation utility to attempt to recover lost operations.)

Even if you do choose to rely on replication to protect remote database data, backups still need to be done periodically at remote databases to prevent the transaction log from growing too large. You should use the same option (rename and restart the log) as at the consolidated database, running the Message Agent so that it has access to the renamed log files. If you set the DELETE_OLD_LOGS option to ON at the remote database, the old log files will be deleted automatically by the Message Agent when they are no longer needed.

Automatic transaction log renaming

You can use the -x Message Agent command-line switch to eliminate the need to rename the transaction log on the remote computer when the database engine is shut down. The -x option renames transaction log after it has been scanned for outgoing messages.

Upgrading consolidated databases

This section describes issues in upgrading a consolidated database in a SQL Remote environment. The same considerations apply to SQL Anywhere databases that are primary sites in a Sybase Replication Server installation.

Installing new software does not always make new features available. In many cases, new features require the Upgrade utility to be run on databases. The Upgrade utility adds any information to the system catalog required for new features to be available. When you run the Upgrade utility, it tells you to archive the transaction log. The reason for this is that a new transaction log is created by the Upgrade utility, with a new file format.

When using SQL Remote or Replication Server, the transaction log must be kept for the Message Agent and the Replication Agent, respectively. After running the Upgrade utility, you should shut down the engine, rename the log, and leave it for the Message Agent to delete. The log should also be archived for backup purposes.

For information on the Upgrade utility, see "The Upgrade utility" in the chapter "SQL Anywhere Components".

The Unload utility and replication

If a database is participating in replication, it cannot be unloaded and reloaded without re-synchronizing the databases in the replication system.

Replication is based on the transaction log, and when a database is unloaded and reloaded, the old transaction log is no longer available. For this reason, good backup practices are especially important when participating in replication.

Error reporting and conflict resolution in SQL Remote

A replication error occurs when a replication cannot be carried out because of some change to the database. If an INSERT statement is received at a database, but a row with the same primary key values has already been inserted, then the INSERT will fail. When a SQL operation fails, it generates an error message from the database. The other operations in the message are still applied.

Primary key errors must be avoided in SQL Remote setups. These errors can be avoided by proper design of publications. Update conflicts can be resolved by SQL Remote using a RESOLVE UPDATE trigger. Properly handled update conflicts are not a problem: they are an integral part of the replication system.

Error reporting and conflict resolution

Errors are reported and conflicts resolved by SQL Remote as follows:

SQL statement failures INSERT errors (two users at different databases insert an identical primary key) and other failures of operations contained in replication messages, are reported in the DBREMOTE output.

UPDATE conflicts SQL Anywhere provides a special form of trigger, the RESOLVE UPDATE trigger, which is triggered by UPDATE conflicts at a consolidated database. In this way, you can design appropriate resolutions based on the situation. If an UPDATE conflict occurs at the consolidated database, any RESOLVE UPDATE triggers are fired, then the UPDATE is applied. Both the RESOLVE UPDATE trigger actions and the UPDATE itself are replicated to other databases. If an update conflict is detected at a remote database, no RESOLVE UPDATE triggers are fired, and the UPDATE is carried out. Examples are provided in the following section.

Error reporting in the Message Agent output

The Message Agent sends log output to a window or a log file recording its operation. By default, log output is sent to the window only; the *-o* command-line option sends output to a log file as well.

The Message Agent log includes the following:

- Listing of messages applied.

- Listing of failed SQL statements.
- Listing of other errors.

In addition, UPDATE conflicts can be inserted into a table by a RESOLVE UPDATE trigger, and reports from that table can be used to track such conflicts.

There may be exceptional cases where you wish to allow an error encountered by the Message Agent when applying SQL statements to go unreported. This may arise when you know the conditions under which the error occurs and are sure that it does not produce inconsistent data and that its consequences can safely be ignored.

To allow errors to go unreported, you can create a BEFORE trigger on the action that will cause the known error. The trigger should signal the REMOTE_STATEMENT_FAILED SQLSTATE or SQLCODE value. For example, if you wish to not report failed INSERT statements on a table because of a missing referenced column, you could create a BEFORE INSERT trigger that signals the REMOTE_STATEMENT_FAILED SQLSTATE when the referenced column does not exist. The INSERT statement fails, but the failure is not reported in the Message Agent log.

Using the REPLICATION_ERROR option

With the REPLICATION_ERROR database option, you can specify a stored procedure to be called by the Message Agent when a SQL error occurs. By default no procedure is called.

The procedure must have a single argument of type CHAR, VARCHAR, or LONG VARCHAR. The procedure is called once with the SQL error message and once with the SQL statement that causes the eror.

While the option allows you to track and monitor SQL errors in replication, you must still design them out of your setup: this option is not intended to resolve such errors.

For example, the procedure could insert the errors into a table with the current time and remote user ID, and this information can then replicate back to the consolidated database. An application at the consolidated database can create a report or send e-mail to an administrator when errors show up.

For information on setting the REPLICATION_ERROR option, see "SET OPTION statement" in the chapter "Watcom-SQL Statements".

RESOLVE UPDATE triggers

An UPDATE conflict is detected by the database engine as a failure of the VERIFY clause values to match the rows in the database.

When an UPDATE conflict is detected, the database engine takes two actions:

1 Any RESOLVE UPDATE triggers are fired.

2 The UPDATE is applied.

UPDATE statements are applied even if the VERIFY clause values do not match, whether or not there is a RESOLVE UPDATE trigger.

This section describes RESOLVE UPDATE triggers. These 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. An UPDATE statement with a VERIFY clause takes the following form:

```
UPDATE table-list
SET column-name = expression, ...
[ FROM table-list ]
[ VERIFY (column-name, ...)
   VALUES ( expression, ...) ]
[ WHERE search-condition ]
```

The VERIFY clause compares the values of specified columns to a set of expected values, which are the values that were present in the publisher database when the UPDATE statement was applied there.

The verify clause is useful only for single-row updates. However, even multi-row update statements entered at a database are replicated as a set of single-row updates by the Message Agent, so this imposes no constraints on client applications.

The syntax for a RESOLVE UPDATE trigger is as follows:

```
CREATE TRIGGER trigger-name
   RESOLVE UPDATE
   OF column-name ON table-name
[ REFERENCING [ OLD AS old_val ]
[ NEW AS new_val ]
[ REMOTE AS remote_val ]
FOR EACH ROW
BEGIN
   ...
END
```

RESOLVE UPDATE triggers fire before each row is updated. The REFERENCING clause allows access to the values in the row of the table to be updated (OLD), to the values the row is to be updated to (NEW) and to the rows that should be present according to the VERIFY clause (REMOTE). Only columns present in the VERIFY clause can be referenced in the REMOTE AS clause; other columns produce a "row not found" error.

Using the VERIFY_ALL_COLUMNS option

The database option VERIFY_ALL_COLUMNS is OFF by default. If it is set to ON, all columns are verified on replicated updates, and a RESOLVE UPDATE trigger is fired whenever any column is different. Setting this option to ON makes messages bigger, because more information is sent for each UPDATE.

If this option is set at the consolidated database before remote databases are extracted, it will be set at the remote databases also.

You can set the VERIFY_ALL_COLUMNS option either for the PUBLIC group or just for the user contained in the Message Agent connection string.

Using the CURRENT REMOTE USER special constant

The CURRENT REMOTE USER special constant holds the user ID of the remote user sending the message. This can be used in RESOLVE UPDATE triggers that place reports of conflicts into a table, to identify the user producing a conflict.

UPDATE conflict resolution examples

This section describes some ways of using RESOLVE UPDATE triggers to handle UPDATE conflicts.

Example: inventory control

The products table of the sample database has a quantity column holding the number of each product left in stock. An update to this column will typically deplete the quantity in stock or, if a new shipment is brought in, add to it.

A sales rep at a remote database enters an order, depleting the stock of small tank top tee shirts (product ID 300) by five, from 28 to 23, and enters this in on her database. Meanwhile, before this update is replicated to the consolidated database, a new shipment of tee shirts comes in, and the warehouse enters the shipment, adding 40 to the quantity column to make it 68.

The warehouse entry gets added to the database: the quantity column now shows there are 68 small tank-top tee shirts in stock. When the update from the sales representative arrives, it causes a conflict-SQL Anywhere detects that the update is from 28 to 23, but that the current value of the column is 68.

A suitable RESOLVE UPDATE trigger for this situation would add the increments from the two updates. For example:

```
CREATE TRIGGER resolve_quantity
RESOLVE UPDATE OF quantity
ON "DBA".product
REFERENCING OLD AS old_name
NEW AS new_name
REMOTE AS remote_name
FOR EACH ROW
BEGIN
   SET new_name.quantity = new_name.quantity
                         + old_name.quantity
                         - remote_name.quantity
END
```

This trigger adds the difference between the old value in the consolidated database (68) and the old value in the remote database when the original UPDATE was executed (28) to the new value being sent, before the UPDATE is implemented. Thus, new_val.quantity becomes 63 (= 23 + 68 - 28), and this value is entered into the quantity column.

Consistency is maintained at the remote database as follows:

1 The original remote UPDATE changed the value from 28 to 23.

2 The warehouse's entry is replicated to the remote database, but fails as the old value is not what was expected.

3 The changes made by the RESOLVE UPDATE trigger are replicated to the remote database.

Example: resolving date conflicts

Suppose the contact table in the sample database had a date column holding the most recent contact with each individual. Update conflicts on this column should be resolved in favor of the most recent date.

One representative talks with the contact on a Friday, but does not upload his changes to the consolidated database until the next Monday. Meanwhile, a second representative meets the contact on the Saturday, and updates her changes that evening.

There is no conflict when the Saturday UPDATE is replicated to the consolidated database, but when the Monday UPDATE arrives it finds the row already changed.

The following RESOLVE UPDATE trigger chooses the most recent of the two new values and enters it in the database.

```
CREATE TRIGGER contact_date RESOLVE UPDATE
ON contact
REFERENCING OLD AS old_name
NEW AS new_name
FOR EACH ROW
```

```
BEGIN
   IF new_name.contact_date <
         old_name.contact_date THEN
      SET new_name.contact_date
            = old_name.contact_date
   END IF
END
```

If the value being updated is later than the value that would replace it, the new value is reset to leave the entry unchanged.

Using passthrough mode for administration

The publisher of the consolidated database can directly intervene at remote sites using a passthrough mode, which enables standard SQL statements to be passed through to a remote site. By default, passthrough mode statements are executed at the local (consolidated) database as well, but an optional keyword prevents the statements from being executed locally.

Passthrough mode is started and stopped using the PASSTHROUGH statement. Any statement entered between the starting PASSTHROUGH statement and the PASSTHROUGH STOP statement which terminates passthrough mode is checked for syntax errors, executed at the current database, and also passed to the identified subscriber and executed at the subscriber database. We can call the statements between a starting and stopping passthrough statement a **passthrough session**.

The following statement starts a passthrough session which passes the statements to a list of two named subscribers, without being executed at the local database:

```
PASSTHROUGH ONLY
FOR userid_1, userid_2;
```

The following statement starts a passthrough session which passes the statements to all subscribers to the specified publication:

```
PASSTHROUGH ONLY
FOR SUBSCRIPTION TO [owner].pubname [ ( string ) ] ;
```

Passthrough mode is additive. In the following example, statement_1 is sent to user_1, and statement_2 is sent to both user_1 and user_2.

```
PASSTHROUGH ONLY FOR user_1 ;
statement_1 ;
PASSTHROUGH ONLY FOR user_2 ;
statement_2 ;
```

The following statement terminates a passthrough session:

```
PASSTHROUGH STOP ;
```

PASSTHROUGH STOP terminates passthrough mode for all remote users.

Notes on using passthrough mode

- You should always qualify object names with the owner name. PASSTHROUGH statements are not executed at remote databases from the same user ID. Consequently, object names without the owner name qualifier may not be resolved correctly.

Uses and limitations of passthrough mode,l0o

Passthrough mode is a powerful tool, and should be used with care. Some statements, especially data definition statements, could cause a running SQL Remote setup to come tumbling down. SQL Remote relies on each database in a setup having the same objects: if a table is altered at some sites but not at others, attempts to replicate data changes will fail.

Also, it is important to remember that in the default setting passthrough mode also executes statements at the local database. To send statements to a remote database without executing them locally you must supply the ONLY keyword. The following set of statements drops a table not only at a remote database, but also at the consolidated database.

```
-- Drop a table at the remote database
-- and at the local database
PASSTHROUGH TO Joe_Remote ;
DROP TABLE CrucialData ;
PASSTHROUGH STOP ;
```

The syntax to drop a table at the remote database only is as follows:

```
-- Drop a table at the remote database only
PASSTHROUGH ONLY TO Joe_Remote ;
DROP TABLE CrucialData ;
PASSTHROUGH STOP ;
```

The following are tasks that can be carried out on a running SQL Remote setup:

- Add new users.
- Resynchronize users.
- Drop users from the setup.
- Change the address, message type, or frequency for a remote user.
- Add a column to a table.

Many other schema changes are likely to cause serious problems if executed on a running SQL Remote setup.

Passthrough works on only one level of a hierarchy

In a multi-tier SQL Remote installation, it becomes important that passthrough statements work on one the level of databases immediately beneath the current level. In a multi-tier installation, passthrough statements must be entered at each consolidated database, for the level beneath it.

Stored procedures and control statements in passthrough mode

There are special considerations for some statements in passthrough mode.

Calling procedures

When a stored procedure is called in passthrough mode using a CALL or EXEC statement, the CALL statement itself is replicated and none of the statements inside the procedure are replicated. It is assumed that the procedure on the replicate side has the correct effect.

Control of flow statements and cursor operations

Control-flow statements such as IF and LOOP, as well as any cursor operations, are not replicated in passthrough mode. Any statements within the loop or control structure *are* replicated.

CHAPTER 28

Running Programs as Services

About this chapter

The material in this chapter applies only to users of SQL Anywhere on Windows NT.

The SQL Anywhere database server and engine, as well as the SQL Anywhere Client, the Open Server Gateway, the SQL Remote Message Agent, and the SQL Anywhere Replication Agent (available as a separate product) can be run as **services** under Windows NT. This chapter describes services, their benefits, and how to run SQL Anywhere executables as services.

The chapter also describes how to use Sybase Central to manage SQL Anywhere services.

Contents

Topic	Page
Introduction to services	518
Managing services	519
Adding and removing SQL Anywhere services	520
Configuring SQL Anywhere services	522
Starting and stopping services	526
Running more than one service	528
Monitoring a SQL Anywhere network server service	531
The Windows NT Control Panel Service Manager	532

Introduction to services

Although SQL Anywhere applications such as the network server and database engine can be run like any other Windows NT executable, there are limitations to running them in this manner, particularly for the SQL Anywhere network server.

When you run a program, it runs under your NT login session: if you then log off the computer, the program terminates. As only one person can be logged on to Windows NT on any one computer at one time, this restricts the use of the computer if you wish to keep a program running much of the time, as is commonly the case with database servers. You must stay logged in to the computer running SQL Anywhere in order for the program to keep running. In addition to the inconvenience, this arrangement can present a security risk, as the NT computer must be left in a logged on state.

Installing an application as an **service** enables it to run even when you log off. For a full description of services and how to use them, see your Windows NT documentation. Several SQL Anywhere executables can be run as Services.

Programs that can be run as a service

The following SQL Anywhere programs can be run as a service:

- The network server (DBSRV50.EXE).
- The standalone engine (DBENG50.EXE).
- The SQL Anywhere Client (DBCLIENT.EXE).
- The Open Server Gateway (DBOS50.EXE).
- The SQL Anywhere Replication Agent (DBLTM.EXE : available separately).
- The SQL Remote Message Agent (DBREMOTE.EXE).
- A sample application.

Not all these applications are supplied in all editions of SQL Anywhere.

The LocalSystem account

When you start a service, it logs on using a special system account called LocalSystem (or using another account you specify). The service is not tied to the user ID of the person starting it, and is not stopped when that person logs off. A service can also be configured to start automatically whenever the NT computer is started, before any user even logs on.

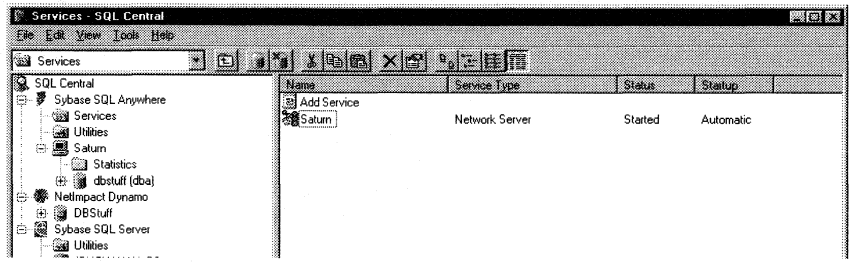
Although you can control most aspects of services from the Services option in the Windows NT Control Panel, Sybase Central provides a more convenient and comprehensive way of managing SQL Anywhere services.

Managing services

You can carry out the following service management tasks from Sybase Central:

- Add, edit, and remove SQL Anywhere services.
- Start, stop, and pause SQL Anywhere services.
- Modify the parameters governing a SQL Anywhere service.
- Add databases to a SQL Anywhere engine or server service, so that you can run several databases at one time.

The service icons in Sybase Central display the current state of each SQL Anywhere service using a traffic light icon (running, paused, or stopped).



Service polling frequency

Sybase Central polls at specified intervals to check the state (started, stopped, paused, removed) of each service, and updates the icons to display the current state.

❖ To alter the Sybase Central polling frequency:

1. In Sybase Central, open the Services folder.
2. Right-click the icon of any service in the right panel of Sybase Central, and select Polling from the popup menu.
3. Set the polling frequency. The frequency applies to all services, not just the one selected.

   The value you set in this window is saved for subsequent sessions, until you change it.

Adding and removing SQL Anywhere services

This section describes how to add and remove SQL Anywhere services.

Adding a SQL Anywhere service

You may wish to set up services for one or more of the SQL Anywhere programs that can be run as a service. This section describes how to set up services using Sybase Central.

❖ **To add a new SQL Anywhere service:**

1. In Sybase Central, open the Services folder.
2. Double-click Add Service. The Service Creation Wizard is displayed..
3. Follow the instructions in the wizard to define the service you are creating:
   - Select the type service you wish to create.
   - Enter a name for the service. Service Names must be unique within the first eight characters.
   - Choose whether you wish the service to start automatically (that is, whenever the computer starts NT), manually (you need to start the service from Sybase Central each time), or if you want the service disabled. You may want to select Disabled if you are setting up a service for future use.
   - Specify the name and complete path of the executable you wish to run as a service.
   - Enter a set of command-line switches for the executable to use in the Parameters box. For example, if you wish a network server to run using the sample database with a cache size of 20Mb, and a name of **myserver**, you would enter the following in the Parameters box:

     ```
     -c 20M
     -n myserver c:\sqlany50.sademo.db
     ```

     Line breaks are optional. For information on valid command-line switches, see the description of each program in the chapter "SQL Anywhere Components".

- Choose the account under which the service will run: the special LocalSystem account or another user ID. For more information about this choice, see "Setting the account options" on page 524.
- If you want the service to be accessible from the Windows NT desktop, check Allow Service to Interact with Desktop. If this option is unchecked, no icon or window appears on the desktop.

For more information on the configuration options, see "Configuring SQL Anywhere services", on page 522.

Removing a SQL Anywhere service

Removing a SQL Anywhere service removes the server name from the list of services. Removing a SQL Anywhere service does not remove any SQL Anywhere software from your hard disk.

If you wish to re-install a SQL Anywhere service that you have removed, you will need to reenter the command-line switches.

❖ To remove a SQL Anywhere service:

1. In Sybase Central, open the Services folder.
2. Right-click the icon of the service you wish to start, and select Delete from the popup menu

Configuring SQL Anywhere services

A SQL Anywhere service runs a SQL Anywhere database server or other application with a set of command-line switches. For a full description of the command-line switches for each of the SQL Anywhere applications, see the chapter "SQL Anywhere Components" and the *SQL Anywhere Network Guide*.

In addition to the command-line switches, SQL Anywhere services accept other parameters that specify the account under which the service will run and the conditions under which it will start.

❖ **To change the parameters for a SQL Anywhere service:**

1. In Sybase Central, open the Services folder.
2. Right-click the icon of the service whose parameters you wish to change in the right panel of Sybase Central, and select Properties from the popup menu.
3. Alter the parameters as needed in the tabs of the Property sheet.
4. When you have altered the parameters as you wish, click OK to accept and save the changes.

Changes to a service configuration do not take effect immediately. They take effect next time the service is started. The Startup option is applied next time Windows NT is started.

Setting the startup option

The following options govern startup behavior for a SQL Anywhere service:

- **Automatic** If you choose the Automatic setting, the SQL Anywhere service starts whenever the NT operating system is started on the computer. This setting is appropriate for database servers and other applications that are run all the time.
- **Manual** If you choose the Manual setting, the SQL Anywhere service starts only when a user with Administrator permissions starts it. For information about Administrator permissions, see your Windows NT documentation.
- **Disable** If you choose the Disable setting, the SQL Anywhere service will not start.

Entering command-line switches

The Configuration tab of the service property sheet provides a text box for entering command-line switches for a service. You should enter command-line switches that govern the behavior of the service in this box. You should not enter the name of the program executable in this box.

Examples

- To start a network server service running two databases, with a cache size of 20 Mb, and with a name of **my_server**, enter the following in the Parameters box:

```
-c 20M
-n my_server
c:\sqlany50\db_1.db
c:\sqlany50\db_2.db
```

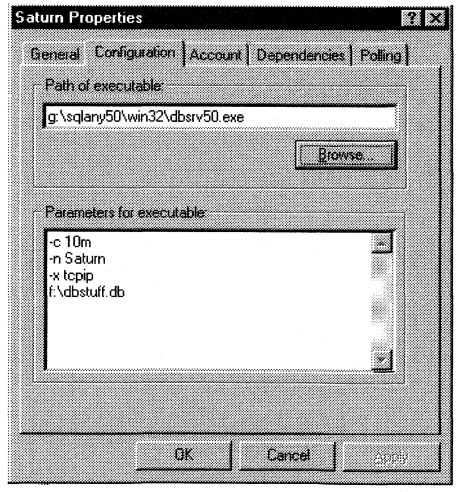
- To start a SQL Remote Message Agent service, connecting to the sample database as user ID DBA, enter the following:

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

- To start a SQL Anywhere Client, connecting to a server named my_server, enter the following:

```
my_server
```

The figure illustrates a sample Property sheet.



The command-line switches for a service are the same as those for the executable. For a full description of the command-line switches for each program, see the chapter "SQL Anywhere Components".

Setting the account options

You can choose under which account the SQL Anywhere service will run. Most services run under the special LocalSystem account, and this is the default option for SQL Anywhere services. You can set the service to log on under another account by clicking Account and entering the account information.

If you choose to run the SQL Anywhere service under an account other than LocalSystem, that account must have the "log on as a service" privilege. This can be granted from the NT User Manager application, under Advanced Privileges.

When an icon appears on the taskbar

Whether or not an icon for the service appears on the taskbar or desktop depends on the account that is selected, and whether Allow Service to Interact with Desktop is checked, as follows:

- If a service is running under LocalSystem, and Allow Service to Interact with Desktop is checked in the Service Configuration window, an icon appears on the desktop of every user logged in to NT on the computer running the service. Consequently any user can open the application window and stop the program that is running as a service.
- If a service is running under LocalSystem, and Allow Service to Interact with Desktop is unchecked in the Service Configuration window, no icon appears on the desktop for any user. Only users with permissions to change the state of services can stop the service.
- If a service is running under another account, no icon appears on the desktop. Only users with permissions to change the state of services can stop the service.

Changing the executable file

To change the program executable file associated with a service, click the Configuration tab on the service property sheet and enter the new path and file name in the Path of Executable box.

If you move an executable file to a new directory, you must modify this entry.

Adding new databases to a server or engine service

Each SQL Anywhere network server or standalone engine can run more than one database. If you wish to run more than one SQL Anywhere database at a time, we recommend that you do so by attaching new databases to your existing SQL Anywhere service, rather than by creating new services.

❖ **To add a new database to a SQL Anywhere service:**

1. Ensure that your SQL Anywhere server is selected in the list of database servers.
2. Double click to change the parameters for the service.
3. Add the filename of the new database to the end of the list of parameters.
4. Click OK to save the changes.

The new database will not be added to the service immediately. It is loaded next time the service is started.

Databases can be started on running servers by client applications, such as ISQL..

For a description of how to start a database on a server from ISQL, see "START DATABASE statement" in the chapter "Watcom-SQL Statements". For a description of how to implement this function in an Embedded SQL application, see the db_start_database function in the chapter "The Embedded SQL Interface". Starting a database from an application does not attach it to the service. If the service is stopped and restarted, the additional database will not be started automatically.

Starting and stopping services

This section describes how you can start, stop, and pause services from Sybase Central.

Starting a SQL Anywhere service

❖ **To start a SQL Anywhere service:**

1. In Sybase Central, open the Services folder.
2. Right-click the icon of the service you wish to start, and select Start from the popup menu

If you close Sybase Central, the SQL Anywhere service will not be stopped. If you log off the computer, the SQL Anywhere service keeps running.

Stopping a SQL Anywhere service

Stopping a SQL Anywhere service for an engine or server closes all connections to the database and unloads the database server. For other applications, the program is closed down.

❖ **To stop a SQL Anywhere service:**

1. In Sybase Central, open the Services folder.
2. Right-click the icon of the service you wish to start, and select Stop from the popup menu

Pausing a SQL Anywhere service

Pausing a service prevents any further action being taken by the application, but does not shut the application down or, in the case of server services, close any client connections to the database. Most users will not need to pause their SQL Anywhere services.

❖ **To pause a SQL Anywhere service:**

1. In Sybase Central, open the Services folder.

2 Right-click the icon of the service you wish to start, and select Pause from the popup menu

Running more than one service

This section describes some topic specific to running more than one service at a time.

Service dependencies

In some circumstances you may wish to run more than one executable as a service, and these executables may depend on each other. For example, you may wish to run a SQL Anywhere Client and also a SQL Remote Message Agent or SQL Anywhere Log Transfer Manager to assist in replication.

In cases such as these, it is important that the services start in the proper order. If a SQL Remote Message Agent service starts up before the SQL Anywhere Client has started, it will fail because it cannot find the server.

You can prevent these problems using **service groups**. You can manage the service groups from Sybase Central.

Service groups overview

You can assign each service on your system to be a member of a service group. By default, each service belongs to a group, as listed in the following table.

Service	Default group
Network server	SQLANYServer
Standalone engine	SQLANYEngine
Client	SQLANYClient
SQL Remote Message Agent	SQLANYRemote
Replication Agent	SQLANYLTM
Open Server Gateway	SQLANYOSG

Before you can configure your services to ensure that they start in the correct order, you must check that your service is a member of an appropriate group. You can check which group a service belongs to, and change this group, from Sybase Central.

❖ **To check which group a service belongs to:**

1. In Sybase Central, open the Services folder.
2. Right-click the icon of the service you wish to start, and select Properties from the popup menu.
3. Click the Dependencies tab. The top text box displays the name of the group to which the service belongs.

❖ **To change which group a service belongs to:**

1. In Sybase Central, open the Services folder.
2. Right-click the icon of the service you wish to start, and select Properties from the popup menu.
3. Click the Dependencies tab. Click Look Up to display a list of available groups on your system.
4. Select one of the groups, or type a name of a new group, and click OK to assign the service to that group.

Managing service dependencies

With Sybase Central you can ensure that at least one member of a each of a list of service groups, and each of a list of services, has started before the current service. The members of these lists are called **dependencies**.

For example, you may wish to ensure that a particular network server has started before a SQL Remote Message Agent that is to run against that server starts.

❖ **To add a service or group to a list of dependencies:**

1. In Sybase Central, open the Services folder.
2. Right-click the icon of the service you wish to start, and select Properties from the popup menu.
3. Click the Dependencies tab. Click Add Service or Add Group to add a service or group to the list of dependencies..
4. Select one of the services or groups from the list, and click OK to add the service or group to the list of dependencies.

Possible problems running more than one server as a service

You may experience problems if you run more than one SQL Anywhere network server as a service at a time. Under some circumstances, it cannot be guaranteed that the correct service will respond to a client that sends a request to the database server as a broadcast. Broadcasts are typically used if you are using the TCP/IP protocol, or the IPX protocol on a Novell network without using the bindery. The broadcast is only used to find the server during startup.

We recommend that you not run more than one SQL Anywhere network server service at a time.

Monitoring a SQL Anywhere network server service

Sybase Central shows whether a service is running, stopped, paused, or not installed. You can find out more information about the state of the server, and control connections to the server, from the following utilities

- **The database server display** When you start a SQL Anywhere network server service, the database server icon appears at the bottom of your screen. Double clicking this icon displays the server display. This display allows you to monitor the database server, as well as close connections from the server.
- **The DBWATCH remote monitoring utility** The DBWATCH display is the same as the database server display. For a description of DBWATCH, see the *SQL Anywhere Network Guide*.

The database server icon is generally displayed on the screen of the machine on which it is running, and server monitoring is generally done from the database server display. There are some cases when services are not allowed to open Windows, and you need an alternative means to monitor the service. These cases are:

- When there is no current logon to Windows NT.
- When the service is running under a normal user account (not LocalSystem), and that account is not logged on.

Also, you may wish to monitor the server from a remote machine. In all these cases, you can use Sybase Central or DBWATCH to monitor and control the server.

The Windows NT Control Panel Service Manager

You can carry out all the service management of SQL Anywhere database servers from Sybase Central. You do not need to use the Windows NT Service Manager in the Control Panel.

If you open the NT Service Manager, a list of services is given. The names of the SQL Anywhere services are formed from the Service Name you provided when installing the service, prefixed by SQL Anywhere. All the SQL Anywhere services you have installed will be found together in the list.

You cannot install or configure parameters for the SQL Anywhere services using the NT Service Manager: you must use Sybase Central.

PART FOUR

Transact-SQL Compatibility

Transact-SQL is the SQL dialect supported by Sybase SQL Server. SQL Anywhere supports a large percentage of Transact-SQL, and this part documents that support.

CHAPTER 29

Using Transact-SQL with SQL Anywhere

About this chapter

This chapter is a guide for creating applications compatible with both SQL Anywhere and SQL Server. It describes SQL Anywhere support for Transact-SQL language elements and statements, and for SQL Server system tables, views, and procedures.

For information about using Transact-SQL-compatible procedures, triggers, and batches in SQL Anywhere, see the chapter "Transact-SQL Procedure Language".

Before you begin

If you are not concerned with SQL Server and Transact-SQL compatibility, you do not need to read this chapter.

Contents

Topic	Page
An overview of SQL Anywhere support for Transact-SQL	536
SQL Server and SQL Anywhere architectures	539
General guidelines for writing portable SQL	543
Configuring SQL Anywhere for Transact-SQL compatibility	544
Using compatible data types	549
Local and global variables	556
Building compatible expressions	561
Using compatible functions	565
Building compatible search conditions	576
Other language elements	580
Transact-SQL statement reference	581
Compatible system catalog information	598
SQL Server system and catalog procedures	601
Implicit data type conversion	603

An overview of SQL Anywhere support for Transact-SQL

The Watcom-SQL dialect supported by SQL Anywhere database engines and servers is based on the ISO/ANSI standard. It conforms to the SQL 89 standard, and has many features defined in IBM's DB2 and SAA specification, as well as in ISO/ANSI SQL/92.

With this release, SQL Anywhere also supports the Sybase Transact-SQL dialect. This chapter describes the compatibility of the two dialects of SQL.

Goals

The goals of SQL Anywhere's Transact-SQL support are:

- To make applications and stored procedures portable. Many applications, stored procedures, and batch files can be written to be used with both SQL Server and SQL Anywhere databases.
- To make data portable. Data can be exchanged and replicated between SQL Anywhere and SQL Server databases with a minimum of effort.

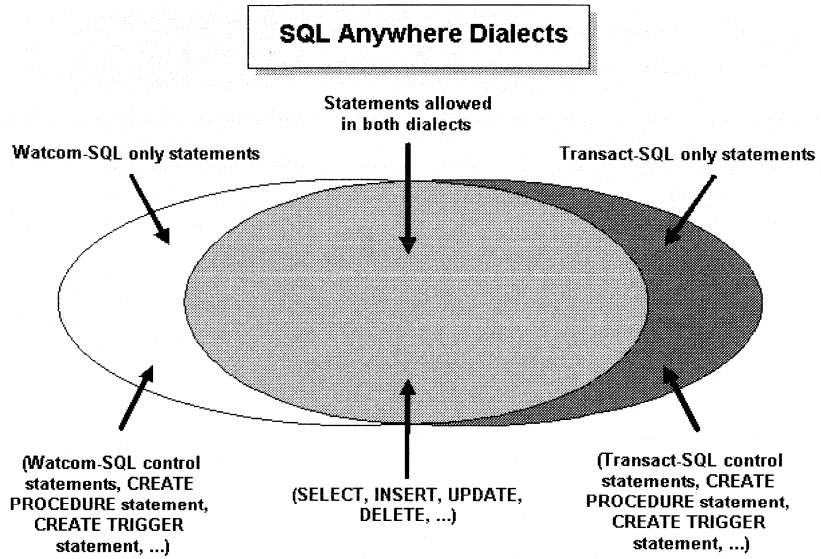
The aim is to make it possible to write applications to work with both SQL Server and SQL Anywhere; existing SQL Server applications will generally require some changes to run on a SQL Anywhere database.

Two dialects in one database

The two dialects of SQL coexist in SQL Anywhere in the following way:

- Many SQL statements can be used in either of the two dialects.
- Where either dialect has an extension that does not conflict with the other dialect, either variant of a statement can be used.
- For groups of statements, as in procedures, triggers, and batches, you must use either the Transact-SQL or Watcom-SQL dialect (including statements common to both) throughout the procedure or batch. For example, each dialect has different flow control statements. You must use control statements from one dialect only throughout the batch or procedure.

The following diagram illustrates how the two dialects overlap.



Similarities and differences

SQL Anywhere supports a very high percentage of Transact-SQL language elements, functions, and statements for working with existing data. For example, all the numeric functions are supported, all but one of the string functions are supported, all aggregate functions are supported, and all date and time functions are supported. As another example, the Transact-SQL outer joins (using =* and *= operators) and extended DELETE and UPDATE statements using joins are supported in SQL Anywhere.

Further, a very high percentage of the Transact-SQL stored procedure language is supported (CREATE PROCEDURE and CREATE TRIGGER syntax, control statements, and so on). The stored procedure language is discussed in the chapter "Transact-SQL Procedure Language".

Many but not all aspects of Transact-SQL data definition language statements are supported.

There are more differences in the architectural and configuration facilities supported by each database. Device management, user management, and maintenance tasks such as backups tend to be system specific, and providing support for such tasks in a consistent manner is less important to most users. Even here, a set of Transact-SQL system tables is provided in SQL Anywhere as views, where the tables that are not meaningful in SQL Anywhere have no rows. Also, a set of system procedures is provided for some of the more common administrative tasks.

This chapter looks first at some system-level issues where differences are most marked, before discussing data manipulation and data definition language aspects of the dialects where compatibility is high.

SQL Server and SQL Anywhere architectures

SQL Server and SQL Anywhere are complementary products, with architectures designed to suit their distinct purposes. SQL Anywhere is designed as a workgroup or departmental server requiring little administration, and as a single-user database. SQL Server is designed as an enterprise-level server for the largest databases.

This section describes architectural differences between SQL Server and SQL Anywhere. It also describes the SQL Server-like tools SQL Anywhere includes for compatible database management.

Servers and databases in SQL Server and SQL Anywhere

The relationship between servers and databases is different in SQL Server and in SQL Anywhere.

In SQL Server, each database exists inside a server, and each server can contain several databases. Users are granted login rights to the server, and can connect to the server. They can then use each database on that server for which they have been granted permissions. There are system-wide system tables held in a master database that contain information common to all databases on the server.

In SQL Anywhere, there is no level corresponding to the SQL Server master database. In SQL Anywhere, each database is an independent entity, containing all its system tables. Users are granted connection rights to a database, not to the server. When a user connects, the connection is to an individual database. There is no system-wide set of system tables maintained at a master database level. Each SQL Anywhere database engine or server can dynamically load and unload multiple databases, and users can maintain independent connections on each, but there is no way of addressing more than one database from within a single connection.

SQL Anywhere provides tools in its Transact-SQL support and in its Open Server Gateway to allow some tasks to be carried out in a SQL Server-like manner. For example, SQL Anywhere provides an implementation of the SQL Server sp_addlogin system procedure that carries out the nearest equivalent action: adding a user to a database.

System tables in SQL Anywhere and SQL Server

In addition to its own system tables, SQL Anywhere provides a set of system views that mimic relevant parts of the SQL Server system tables. These are listed and described individually in "Compatible system catalog information" on page 598, which describes the system catalogs of the two products. This section provides a brief overview of the differences.

The SQL Anywhere system tables are held entirely within each database, while the SQL Server system tables are held partly inside each database and partly at the server level. The SQL Anywhere architecture does not include a level above the database.

In SQL Server, the system tables are owned by the database owner, user ID dbo. In SQL Anywhere, the system tables are owned by the system owner, user ID SYS. The SQL Server-compatible system views provided by SQL Anywhere are owned by a dbo user ID.

Administrative roles in SQL Server and SQL Anywhere

SQL Server has a more elaborate set of administrative roles than SQL Anywhere. In SQL Server the following are distinct roles, although more than one login account on a SQL Server can be granted any role, and one account can possess more than one role.

SQL Server roles

- The System Administrator is responsible for general administrative tasks unrelated to specific applications; can access any database object.
- The System Security Officer is responsible for security-sensitive tasks in SQL Server, but has no special permissions on database objects.
- The Database Owner has full permissions on objects inside the database that he or she owns, can add users to a database and can grant other users the permission to create objects and execute commands within the database.
- Permissions can be granted to users for specific data definition statements, such as CREATE TABLE or CREATE VIEW, enabling the user to use those statements to create database objects.
- Each database object has an owner, who may grant permissions to other users to access the object. The owner of an object automatically has all permissions on the object.

In SQL Anywhere the following database-wide permissions have administrative roles:

- The Database Administrator (DBA permissions) has, like the SQL Server database owner, full permissions on objects inside the database that he or she owns and can grant other users the permission to create objects and execute commands within the database. The default database administrator is user ID DBA. In SQL Anywhere, DBA is a reserved word and must have double quotes around it when used in object name qualifiers.
- The RESOURCE permission allows a user to create any kind of object within a database. This is instead of the SQL Server scheme of granting permissions on individual CREATE statements.
- SQL Anywhere has object owners in the same way that SQL Server does. The owner of an object automatically has all permissions on the object, including the right to grant permissions.

To ease seamless access to data held in both SQL Server and SQL Anywhere, you should create user IDs with appropriate permissions in the database (RESOURCE in SQL Anywhere, or permission on individual CREATE statements in SQL Server) and create objects from that user ID. If the same user ID is used in each environment, object names and qualifiers can be identical in the two databases, helping to ensure compatible access.

Users and groups in SQL Anywhere and SQL Server

There are some differences between the SQL Server and SQL Anywhere models of users and groups. In SQL Server, connections are made to a server and each user requires a login ID and password to the server as well as a user ID for each database they will access on that server. Each user of a database can be a member of at most one group. In SQL Anywhere, where connections are made to a database, there is nothing corresponding to the login ID. Instead, each user is granted a user ID and password on a database in order to use that database. Users can be members of many groups, and group hierarchies are allowed.

SQL Server and SQL Anywhere both allow user groups, to permit granting permissions to many users at one time. However, there are differences in the specifics of groups in the two products (for example, SQL Server allows each user to be a member of only one group, while SQL Anywhere has no such restriction). You should compare the documentation on users and groups in the two products for specific information.

Both SQL Server and SQL Anywhere have a public group, for defining default permissions. Every user is automatically a member of the public group.

SQL Anywhere supports the following SQL Server system procedures for managing users and groups.

For the arguments to each procedure, see "SQL Server system procedures" on page 601:

System procedure	Description
sp_addlogin	In SQL Server, this adds a user to the server. In SQL Anywhere, this adds a user to a database
sp_adduser	In SQL Server and SQL Anywhere, this adds a user to a database. While this is a distinct task from sp_addlogin in SQL Server, in SQL Anywhere they are the same
sp_addgroup	Adds a group to a database
sp_changegroup	Adds a user to a group or moves a user from one group to another
sp_droplogin	In SQL Server, removes a user from the server. In SQL Anywhere, removes a user from the database
sp_dropuser	Removes a user from the database
sp_dropgroup	Removes a group from the database

General guidelines for writing portable SQL

When writing SQL that will be used on more than one database, you should be as explicit as possible in your SQL statements. Even if a given SQL statement is supported by more than one database engine, it may be a mistake to assume that default behavior when an option is not specified as the same on each engine. The following general guidelines apply when writing compatible SQL:

- Spell out all the available options, rather than assuming portable default behavior.
- Make the order of execution within statements explicit using parentheses, rather than assuming identical default order of precedence for operators.
- Use the Transact-SQL convention of an @ sign preceding variable names for SQL Server portability.
- In procedures, triggers, and batches, declare variables and cursors immediately following a BEGIN statement. This is required by SQL Anywhere, although SQL Server allows declarations to be made anywhere in a procedure, trigger, or batch.
- Avoid using keywords from either SQL Server or SQL Anywhere as identifiers in your databases.

Configuring SQL Anywhere for Transact-SQL compatibility

Some differences in behavior between the SQL Anywhere database engine and SQL Server can be eliminated by selecting appropriate options when creating a database or, if you are working on an existing database, when rebuilding the database. Other differences can be controlled by connection level options using the SET OPTION statement in SQL Anywhere or the SET statement in SQL Server.

Creating a Transact-SQL-compatible database

This section describes choices that must be made when a database is created or rebuilt.

Ignore trailing blanks in comparisons

When building a SQL Server-compatible database using SQL Anywhere, you should choose the option to ignore trailing blanks in comparisons.

- If you are using Sybase Central, this option is on the second page of the Create Database wizard.
- If you are using ISQL for OS/2 or Windows 3.x, this option is on the Create Database dialog box that appears when you select Create Database from the Database Tools window and click Create.
- If you are using the DBINIT command line utility, specify the -b command line switch.

With this option chosen, the following two strings are considered equal by both SQL Server and SQL Anywhere:

```
'ignore the trailing blanks   '
'ignore the trailing blanks'
```

If this option is not chosen, the two strings above are considered different by SQL Anywhere.

A side effect of this option is that strings are padded with blanks when fetched by a client application.

Ensure the dbo user ID is set

SQL Anywhere provides system views that mimic relevant parts of the SQL Server system tables. The owner of these views is, by default, the user ID dbo, as in SQL Server databases. If you have a database that already has a user ID named dbo, then you can transfer the ownership of these system views to another user ID. You can set the owner of the Transact-SQL system views as follows:

- If you are using Sybase Central, this option is on the second page of the Create Database wizard.
- If you are using ISQL for OS/2 or Windows 3.x, this option is on the Create Database dialog box which appears when you select Create Database from the Database Tools window and click Create.
- If you are using the DBINIT command line utility, specify the -g command line switch.

Remove historical system views

Older versions of Sybase SQL Anywhere employed two system views whose names conflict with the SQL Server system views provided for compatibility. These views are SYSCOLUMNS and SYSINDEXES. If you are not using Watcom SQL 4.0 and you are using the Open Server Gateway, you should create your database excluding these views. You can do this with the DBINIT -k command-line switch.

If you do not use this option when creating your database, the following two statements return different results:

```
SELECT * FROM SYSCOLUMNS ;

SELECT * FROM dbo.SYSCOLUMNS ;
```

❖ **To drop the system views from an existing database:**

1. Connect to the database as a user with DBA authority.
2. Execute the following statements:

```
DROP VIEW SYS.SYSCOLUMNS ;

DROP VIEW SYS.SYSINDEXES
```

> ***Caution***
> *Ensure that you do not drop the dbo.SYSCOLUMNS or dbo.SYSINDEXES system views.*

Setting options for Transact-SQL compatibility

SQL Anywhere database options are set using the SET OPTION statement. Several database option settings are relevant to Transact-SQL behavior.

Set the allow_nulls_by_default option

By default, SQL Server does not allow NULLs on new columns unless they are explicitly declared to allow NULLs. SQL Anywhere permits NULLs in new columns by default, which is compatible with the SQL/92 ISO standard.

To make SQL Server behave in a SQL/92-compatible manner, use the sp_dboption system procedure to set the allow_nulls_by_default option to true.

To make SQL Anywhere behave in a Transact-SQL-compatible manner, set the allow_nulls_by_default option to OFF. You can do this using the SET OPTION statement as follows:

```
SET OPTION PUBLIC.allow_nulls_by_default = 'OFF'
```

Set the quoted_identifier option

By default, the SQL Server treatment of identifiers and of strings differs from the SQL Anywhere behavior, which matches the SQL/92 ISO standard.

The quoted_identifier option is available in both SQL Server and SQL Anywhere. You should ensure that the option is set to the same value in both databases for identifiers and strings to be treated in a compatible manner.

For SQL/92 behavior, set the quoted_identifier option to ON in both SQL Server and SQL Anywhere.

For Transact-SQL behavior, set the quoted_identifier option to OFF in both SQL Server and SQL Anywhere. If you choose this, you can no longer use double-quoted keywords in identifiers (such as the DBA user ID).

For more information on the quoted_identifier option, see "Compatibility of constants" on page 561.

Set the automatic_timestamp option to ON

Transact-SQL defines a timestamp column with special properties. With the automatic_timestamp option set to ON, the SQL Anywhere treatment of timestamp columns is more similar to SQL Server behavior.

With the automatic_timestamp option set to ON in SQL Anywhere (the default setting is OFF), any new columns with the TIMESTAMP data type that do not have an explicit default value defined are given a default value of timestamp.

For information on timestamp columns, see "The special Transact-SQL timestamp column and data type" on page 552.

Set the string_rtruncation option

Both SQL Server and SQL Anywhere support the string_rtruncation option, which affects whether or not error messages are reported when an INSERT or UPDATE string is truncated. You should ensure that the option is set to the same value in each database.

For more information on the string_rtruncation option, see "SET OPTION statement" in the chapter "Watcom-SQL Statements".

For more information

For more information on database options for Transact-SQL compatibility, see "Transact-SQL SET statement" on page 594.

If you are accessing a SQL Anywhere database using the Open Server Gateway, you can take advantage of actions that the Open Server Gateway performs on connection. These actions are listed in "Open Server Gateway actions on connecting" in the chapter "Using the Open Server Gateway".

Case-sensitivity in SQL Server and SQL Anywhere databases

The case-sensitivity of SQL Anywhere data in comparisons is decided when the database is created. By default, SQL Anywhere databases are case-insensitive in comparisons, although data is always held in the case in which it is entered.

SQL Anywhere does not support case sensitive identifiers.

SQL Server's sensitivity to the case (upper or lower) of identifiers and data depends on the sort order installed on the SQL Server. Case sensitivity can be changed for single-byte character sets by reconfiguring the SQL Server sort order.

In SQL Server, user-defined data type names are case sensitive. In SQL Anywhere, they are case insensitive.

Ensuring compatible object names

Database objects must have a unique name within a certain name space. Outside this name space, duplicate names are allowed. There are some database objects that occupy different name spaces in SQL Server and SQL Anywhere.

In SQL Anywhere, indexes and triggers are owned by the owner of the table on which they are created. Index and trigger names must be unique for a given owner. For example, while the tables t1 owned by user user1 and t2 owned by user user2 may have indexes of the same name, no two tables owned by a single user may have an index of the same name.

SQL Server has a less restrictive name space for index names than SQL Anywhere. Index names must be unique on a given table, but any two tables may have an index of the same name. For compatible SQL, you should stay within the SQL Anywhere restriction of unique index names for a given table owner.

SQL Server has a more restrictive name space on trigger names than SQL Anywhere. Trigger names must be unique in the database. For compatible SQL, you should stay within the SQL Server restriction and make your trigger names unique in the database.

Using compatible data types

Although there are many data types that are common to SQL Server and SQL Anywhere, some data types are unique to each DBMS.

The following tables summarize the compatibility of data types between the two DBMS's. These tables are a guide only, and a marking of both may not mean that the data type performs in an identical manner for all purposes under all circumstances. For detailed descriptions, you should refer to the SQL Server documentation and chapter of this guide "Watcom-SQL Language Reference".

Differences between SQL Anywhere and SQL Server in how data types are stored are not discussed. Such differences do affect performance and resources, but not the syntax or semantics of supported SQL statements.

Integer data types

Data type	Description	Supported by
INT	Signed integer	Both
INTEGER	Synonym for INT	Both
SMALLINT	2-byte integer	Both
TINYINT	<= 255	Both

Notes

- In SQL Anywhere Embedded SQL, TINYINT columns should be fetched into 2-byte or 4-byte integer columns. Also, to send a TINYINT value to a database the C variable should be an integer.
- In SQL Anywhere TINYINT columns should not be fetched into Embedded SQL variables defined as char or unsigned char, since the result is an attempt to convert the value of the column to a string and then assign the first byte to the variable in the program.

Decimal data types

Data type	Description	Supported by
NUMERIC (p,s)	Exact decimal	Both
DECIMAL (p,s)	Synonym for NUMERIC*	Both

Data type	Description	Supported by
REAL	Single precision floating-point number	Both
DOUBLE	Double precision floating-point number	Both
FLOAT [(p)]	Synonym for REAL or DOUBLE	Both

Notes

- Only the NUMERIC data type with s = 0 can be used for the Transact-SQL identity column.
- You should avoid default precision and scale settings for numeric and decimal data types, as these are different in SQL Anywhere and SQL Server. In SQL Anywhere, the default setting is precision = 30 and scale = 6; in SQL Server, the default setting is precision = 18 and scale = 0.
- The FLOAT (*p*) data type is a synonym for REAL or DOUBLE, depending on the value of *p*. For SQL Server, REAL is used for p less than or equal to 15, and DOUBLE for p greater than 15. For SQL Anywhere, the cutoff is platform-dependent, but on all platforms the cutoff value is greater than 15.

Compatibility of character data types

Data type	Description	Supported by
CHAR (n)	Character data of at most n characters	Both
CHARACTER (n)	Synonym for CHAR	SQL Anywhere
NCHAR(n)	As CHAR(n), but for multibyte sets	SQL Server
VARCHAR (n)	Variable length character data	Both
NVARCHAR(n)	As VARCHAR(n), but for multibyte sets	SQL Server
LONG VARCHAR	Arbitrary length character data	SQL Anywhere
TEXT	Arbitrary length character data	Both

Notes

- TEXT is implemented in SQL Anywhere as a user-defined data type, as LONG VARCHAR allowing NULL.

- CHAR(n) fields in SQL Anywhere allow multi-byte strings.

Compatibility of binary data types

Data type	Description	Supported by
BINARY (n)	Binary data of maximum length n	both
LONG BINARY	Arbitrary length binary data.	SQL Anywhere
VARBINARY (n)	Binary data	SQL Server
IMAGE	Binary data	both

Notes

- IMAGE is implemented in SQL Anywhere as a user-defined data type, as LONG BINARY allowing NULL.

Money data types

Data type	Description	Supported by
MONEY	Currency data	Both
SMALLMONEY	Currency data	Both

Notes

- The MONEY data type is implemented in SQL Anywhere as a user-defined data type, as NUMERIC(20,4), allowing NULL.
- The SMALLMONEY data type is implemented in SQL Anywhere as a user-defined data type, as NUMERIC(10,4), allowing NULL.

Bit data type

Data type	Description	Supported by
BIT	0 or 1	Both

Notes

- BIT is implemented in SQL Anywhere as a user-defined data type, as a TINYINT allowing NULL. Consequently, values other than 0 or 1 can be stored in BIT columns in SQL Anywhere. You should, however, avoid storing values other than 0 or 1 in BIT columns.

Compatibility of date and time data types

Data type	Description	Supported by
DATETIME	Date and time information	both
TIMESTAMP	Date and time information	SQL Anywhere: similar to DATETIME
SMALLDATETIME	Dates and times, accurate to a minute	both
TIME	Time of day	SQL Anywhere
DATE	Year, month and day	SQL Anywhere

Notes

- The SQL Anywhere data type TIMESTAMP is not the same as the special TIMESTAMP column in Transact-SQL.
- The DATETIME and SMALLDATETIME data types are implemented in SQL Anywhere as user-defined data type. They are defined as TIMESTAMP data types.

The special Transact-SQL timestamp column and data type

SQL Anywhere supports the Transact-SQL special timestamp column. The timestamp column is used together with the tsequal system function to check whether a row has been updated.

Two meanings of timestamp
SQL Anywhere has a TIMESTAMP data type, which holds accurate date and time information. This is distinct from the special Transact-SQL TIMESTAMP column and data type.

Creating a Transact-SQL timestamp column in SQL Anywhere

To create a Transact-SQL timestamp column, create a column that has the (SQL Anywhere) data type TIMESTAMP and has a default setting of timestamp. The column can have any name, although the name timestamp is commonly used.

For example, the following CREATE TABLE statement includes a Transact-SQL timestamp column:

```
CREATE TABLE table_name (
   column_1 INTEGER ,
   column_2 TIMESTAMP default timestamp
```

```
)
```

The following ALTER TABLE statement adds a Transact-SQL timestamp column to the sales_order table:

```
ALTER TABLE sales_order
ADD timestamp TIMESTAMP default timestamp
```

In SQL Server a column with the name timestamp and no data type specified is automatically given a TIMESTAMP data type. In SQL Anywhere you must explicitly assign the data type yourself.

If you have the AUTOMATIC_TIMESTAMP database option set to ON, then you do not need to set the default value: any new column created with TIMESTAMP data type and with no explicit default is given a default value of timestamp. The following statement sets AUTOMATIC_TIMESTAMP to ON:

```
SET OPTION PUBLIC.AUTOMATIC_TIMESTAMP='ON'
```

The data type of a timestamp column

SQL Server treats a timestamp column as a user-defined data type that is VARBINARY(8), allowing NULL, while SQL Anywhere treats a timestamp column as the TIMESTAMP data type, which consists of the date and time, with fractions of a second held to six decimal places.

When fetching from the table for later updates, the variable into which the timestamp value is fetched should correspond to the way the column is described.

Timestamping an existing table

If you add a special timestamp column to an existing table, all existing rows have a NULL value in the timestamp column. To enter a timestamp value for existing rows, update all rows in the table such that the data does not change. For example, the following statement updates all rows in the sales_order table, without changing the values in any of the rows:

```
UPDATE sales_order
SET region = region
```

In ISQL, you may need to set the TIMESTAMP_FORMAT option to see the differences in values for the rows. The following statement sets the TIMESTAMP_FORMAT option to display all six digits in the fractions of a second:

```
SET OPTION TIMESTAMP_FORMAT='YYYY-MM-DD
HH:NN:ss.SSSSSS'
```

If all six digits are not shown, some timestamp column values may appear to be equal: they are not.

Using tsequal for updates

With the tsequal system function you can tell whether a timestamp column has been updated or not.

For example, an application may SELECT a timestamp column into a variable, and when an UPDATE of one of the selected rows is submitted, it can use the tsequal function to check that the row has not been changed. The tsequal function compares the timestamp value in the table with the timestamp value obtained in the SELECT. If they are the same, the row has not been changed; if they differ, the row has been changed since the SELECT was carried out.

The following is a typical UPDATE statement using the tsequal function:

```
UPDATE publishers
SET city = 'Springfield'
WHERE pub_id = '0736'
AND TSEQUAL(timestamp, '1995/10/25 11:08:34.173226')
```

The first argument to the tsequal function is the name of the special timestamp column; the second argument is the timestamp retrieved in the SELECT statement. In Embedded SQL, the second argument is likely to be a host variable containing a TIMESTAMP value from a recent FETCH on the column.

The special identity column

To create an identity column, use the following CREATE TABLE syntax:

```
CREATE TABLE table-name (
    ...
    column-name numeric(n,0) IDENTITY NOT NULL,
    ...
)
```

where *n* is large enough to hold the maximum number of rows that may be inserted into the table.

The identity column is used to store sequential numbers, such as invoice numbers or employee numbers, that are generated automatically by SQL Anywhere. The value of the identity column uniquely identifies each row in a table.

In SQL Server, each table in a database can have one identity column. The data type must be numeric with scale zero, and the identity column should not allow nulls.

In SQL Anywhere, the identity column is implemented as a column default setting. Values that are not part of the sequence may be explicitly inserted into the column with an INSERT statement. SQL Server does not allow INSERTs into identity columns unless the identity_insert option is set to *on*. In SQL Anywhere, you need to set the NOT NULL property yourself and ensure that no more than one column is an identity column. SQL Anywhere allows any numeric data type to be used as an identity column.

The first time you insert a row into the table, SQL Anywhere assigns a value of *1* to an identity column. On each subsequent insert, the value of the column is incremented by one. The value most recently inserted into an identity column is available in the @@identity global variable.

In SQL Anywhere the identity column is identical to the AUTOINCREMENT default setting for a column.

Compatibility of user-defined data types

SQL Anywhere supports user-defined data types. For a general description, see "User-defined data types" in the chapter "SQL Anywhere Data Types" and for syntax of the SQL statement, see "CREATE DATATYPE statement" in the chapter "Watcom-SQL Statements".

In SQL Anywhere, user-defined data types are created with a base data type, and optionally a NULL or NOT NULL condition, a default value, and a CHECK condition. Named constraints and named defaults are not supported.

You can use the sp_addtype system procedure to add a user-defined data type, or you can use the CREATE DATATYPE statement.

Local and global variables

Local variables are declared by the user and can be used in procedures or in batches of SQL statements to hold information. Global variables are system-supplied variables that provide system-supplied values. All global variables have names beginning with two @ signs. For example, the global variable @@version has a value that is the current version number of the database engine. Users cannot define global variables.

Local variable support

SQL Server and SQL Anywhere both support local variables. In Transact-SQL all variables must be prefixed with an @ sign. In SQL Anywhere the @ prefix is optional. To write compatible SQL, ensure all your variables have the @ prefix.

Scope of local variables

The scope of local variables is different in SQL Anywhere and SQL Server.

SQL Anywhere provides two kinds of local variables:

- You can use the DECLARE statement at the beginning of a compound statement to declare a variable. The variable exists within the compound statement only.
- You can use the CREATE VARIABLE statement to create a variable that exists for the connection.

SQL Server local variables are declared using the DECLARE statement, and exist for the duration of the batch in which they are declared.

SQL Anywhere supports the DECLARE statement to declare local variables within a batch. However, if the DECLARE is executed within a compound statement, it does not exist outside the compound statement.

For more information on batches and local variable scope, see "Variable and cursor declarations" in the chapter "Transact-SQL Procedure Language".

Global variable support

The following list includes all SQL Server global variables supported in SQL Anywhere. SQL Server global variables not supported by SQL Anywhere are not included in the list. All the listed global variables return a value, but in some cases that value is fixed at NULL, 1, -1, or 0, and may not be meaningful. This is indicated in the list.

Some global variables, such as @@identity, hold connection-specific information, and so have connection-specific values. Other variables, such as @@connections, have values that are common to all connections.

Global variable	Returns
@@char_convert	Returns 0
@@client_csname	In SQL Server, the client's character set name. Set to NULL if client character set has never been initialized; otherwise, it contains the name of the most recently used character set. Returns NULL in SQL Anywhere
@@client_csid	In SQL Server, the client's character set ID. Set to -1 if client character set has never been initialized; otherwise, it contains the most recently used client character set ID from syscharsets. Returns -1 in SQL Anywhere
@@connections	The number of logins since the server was last started
@@cpu_busy	In SQL Server, the amount of time, in ticks, that the CPU has spent doing SQL Server work since the last time SQL Server was started. In SQL Anywhere, returns 0
@@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 statement resets @@error, including PRINT statements or IF tests, 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

Global variable	Returns
	that contained it. @@identity retains the last value inserted into an IDENTITY column, even if the statement that inserted it fails to commit
@@idle	In SQL Server, the amount of time, in ticks, that SQL Server has been idle since it was last started. In SQL Anywhere returns 0
@@io_busy	In SQL Server, amount of time, in ticks, that SQL Server has spent doing input and output operations since it was last started. In SQL Anywhere, returns 0
@@isolation	Current isolation level of the connection. In SQL Server, @@isolation takes the value of the active level
@@langid	In SQL Server, defines the local language id of the language currently in use. In SQL Anywhere, returns 0
@@language	In SQL Server, defines the name of the language currently in use. In SQL Anywhere, returns "English"
@@maxcharlen	In SQL Server, maximum length, in bytes, of a character in SQL Server's default character set. In SQL Anywhere, returns 1
@@max_ connections	For the standalone engine, the maximum number of simultaneous connections that can be made to the engine, which is 10. For the network server, the maximum number of active clients (not of database connections, as each client can support multiple connections). For SQL Server, the maximum number of connections to the server.
@@ncharsize	In SQL Server, average length, in bytes, of a national character. In SQL Anywhere, returns 1
@@nestlevel	In SQL Server, nesting level of current execution (initially 0). Each time a stored procedure or trigger calls another stored procedure or trigger, the nesting level is incremented. In SQL Anywhere, returns -1
@@pack_received	In SQL Server, number of input packets read by SQL Server since it was last started. In SQL Anywhere, returns 0
@@pack_sent	In SQL Server, number of output packets written by SQL Server since it was last started. In SQL Anywhere, returns 0
@@packet_errors	In SQL Server, number of errors that have occurred while SQL Server was sending and receiving packets. In SQL

Global variable	Returns
	Anywhere, returns 0.
@@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 local SQL Server or SQL Anywhere server
@@spid	In SQL Server, server process ID number of the current process. In SQL Anywhere, the connection handle for the current connection. This is the same value as that displayed by the sa_conn_info procedure.
@@sqlstatus	Contains status information resulting from the last fetch statement. @@sqlstatus may contain the following values: 0 The fetch statement completed successfully 1 The fetch statement resulted in an error 2 There is no more data in the result set
@@textsize	Current value of the set textsize option, which specifies the maximum length, in bytes, of text or image data to be returned with a select statement
@@thresh_hysteresis	In SQL Server, change in free space required to activate a threshold. In SQL Anywhere, returns 0
@@timeticks	In SQL Server, number of microseconds per tick. The amount of time per tick is machine dependent. In SQL Anywhere, returns 0
@@total_errors	In SQL Server, number of errors that have occurred while SQL Server was reading or writing. In SQL Anywhere, returns 0.
@@total_read	In SQL Server, number of disk reads by SQL Server since it was last started. In SQL Anywhere, returns 0
@@total_write	In SQL Server, number of disk writes by SQL Server since it was last started. In SQL Anywhere, returns 0
@@tranchained	Current transaction mode of the Transact-SQL program. @@tranchained returns a 0 for unchained or a 1 for chained.
@@trancount	Nesting level of transactions. Each begin transaction in a batch increments the transaction count. For details, see "Transact-SQL BEGIN TRANSACTION

Global variable	Returns
	statement" on page 582
@@transtate	In SQL Server, current state of a transaction after a statement executes. In SQL Anywhere, returns -1
@@version	Information on the current version of SQL Server or SQL Anywhere

Obtaining the values of variables

You can use a SELECT statement with an empty table list to display the values of variables. For example, the following statement returns the value of the global variable @@version:

```
SELECT @@version
```

For more information on the SELECT statement, see "Transact-SQL SELECT statement" on page 592.

Building compatible expressions

The following table describes the compatibility of expressions between SQL Server and SQL Anywhere. These tables are a guide only, and a marking of both may not mean that the expression performs in an identical manner for all purposes under all circumstances. For detailed descriptions, you should refer to the SQL Server documentation and the SQL Anywhere User's Guide.

In the table, expr is short for expression, and op is short for operator. Further description of each kind of expression is given in subsequent subsections.

Expression	Supported by
constant	Both
column name	Both
variable name	Both
function (expr)	Both
- expr	Both
expr op expr	Both
(expr)	Both
(subquery)	Both
if-expression	SQL Anywhere

For more information on compatibility of constants, see "Compatibility of constants", next. For more information on compatibility of operators, see "Compatibility of operators" on page 563.

Compatibility of constants

Constant	Supported by
integer	Both
number	Both
'string'	Both
special-constant	Both
host-variable	SQL Anywhere

Default interpretation of delimited strings

By default, SQL Server and SQL Anywhere give different meanings to delimited strings: that is, strings enclosed in apostrophes (single quotes) and in quotation marks (double quotes).

SQL Anywhere employs the SQL/92 convention, that strings enclosed in apostrophes are constant expressions, and strings enclosed in quotation marks (double quotes) are delimited identifiers (names for database objects). SQL Server employs the convention that strings enclosed in quotation marks are constants, while delimited identifiers are not allowed by default and are treated as strings.

The quoted_identifier option

Both SQL Server and SQL Anywhere provide a quoted_identifier option that allows the interpretation of delimited strings to be changed. By default, the quoted_identifier option is set to OFF in SQL Server, and to ON in SQL Anywhere.

While the Transact-SQL SET statement is not supported for most SQL Server connection options, it is supported for the quoted_identifier option.

The following statement in either SQL Anywhere or SQL Server changes the setting of the quoted_identifier option to ON:

```
SET quoted_identifier ON
```

With the quoted_identifier option set to ON, SQL Server allows table, view, and column names to be delimited by quotes. Other object names cannot be delimited in SQL Server.

The following statement in SQL Anywhere or SQL Server changes the setting of the quoted_identifier option to OFF:

```
SET quoted_identifier OFF
```

Compatible interpretation of delimited strings

You can choose to use either the SQL/92 or the default Transact-SQL convention in both SQL Server and SQL Anywhere as long as the quoted_identifier option is set to the same value in each DBMS.

Examples

If you choose to operate with the quoted_identifier option on (the default SQL Anywhere setting), then the following statements involving the SQL keyword user are valid for both DBMS's.

```
CREATE TABLE "user" (
   col1 char(5)
) ;
INSERT "user" ( col1 )
VALUES ( 'abcde' ) ;
```

If you choose to operate with the quoted_identifier option off (the default SQL Server setting), then the following statements are valid for both DBMS's.

```
SELECT *
FROM employee
WHERE emp_lname = "Chin"
```

You cannot use SQL keywords as identifiers if the quoted_identifier option is off.

Compatibility of operators

The following tables describe the compatibility of operators between SQL Server and SQL Anywhere. These tables are a guide only, and a marking of both may not mean that the expression performs in an identical manner for all purposes under all circumstances. For detailed descriptions, you should refer to the SQL Server documentation and the SQL Anywhere User's Guide.

Arithmetic operators

Operator	Description	Supported by
+	addition	Both
-	subtraction	Both
*	multiplication	Both
/	division	Both
%	modulo operator	SQL Server

Notes

- While SQL Anywhere does not provide the % operator, it does provide a MOD function. MOD(21,11) in SQL Anywhere is the same as 21 % 11 in Transact-SQL.
- SQL Anywhere treats % as a comment delimiter.

String concatenation operator

Operator	Supported by
+	Both
||	SQL Anywhere

Notes

- When using the + concatenation operator in SQL Anywhere, you should ensure the operands are explicitly set to strings rather than relying on implicit data conversion. For example, the following query returns the integer value 579: SELECT 123 + 456 whereas the following query returns the character string 123456: SELECT '123' + '456' You can use the CAST or CONVERT function to explicitly convert data types.

Bitwise operators

Operator	Description	Supported by
&	and	Both
|	or	Both
^	exclusive or	Both
~	not	Both

Join operators

The Transact-SQL outer join operators *= and =* are supported in SQL Anywhere, in addition to the SQL/92 join syntax using a table expression in the FROM clause.

Operator precedence

When using more than one operator in an expression, it is recommended that you make the order of operation explicit using parentheses rather than relying on an identical operator precedence between the two database engines.

Using compatible functions

The following tables describe the compatibility of functions between SQL Server and SQL Anywhere. These tables are a guide only, and a marking of both may not mean that the function performs in an identical manner for all purposes under all circumstances. For detailed descriptions, you should refer to the SQL Server documentation and the section "Watcom-SQL Functions".

Compatibility of aggregate functions

Function	Supported by
COUNT(*)	Both
COUNT	Both
LIST	SQL Anywhere
AVG	Both
MAX	Both
MIN	Both
SUM	Both

Compatibility of numeric functions

In the following table, a and b are numeric expressions, and n is an integer expression.

Function	Description	Supported by
ABS (a)	Absolute value of a	Both
ACOS (a)	Arc cosine of a, in radians	Both
ASIN (a)	Arc sine of a, in radians	Both
ATAN (a)	Arc tangent of a, in radians	Both
ATN2 (a,b)	Arc tangent of a/b, in radians	Both
CEILING (a)	Smallest integer not less than a	Both
COS (a)	Cosine of a, with a in radians	Both
COT (a)	Cotangent of a, with a in radians	Both

Function	Description	Supported by
DEGREES (a)	Converts radians to degrees	Both
EXP (a)	Exponential function of a	Both
FLOOR (a)	Largest integer not greater than a	Both
LOG (a)	Natural logarithm of a	Both
LOG10 (a)	Logarithm base 10 of a	Both
MOD (a, b)	Remainder when a is divided by b	SQL Anywhere
PI()	Numeric value Pi	Both
POWER (a,b)	a to the power b	Both
RADIANS (a)	Converts degrees to radians	Both
RAND ([n])	Random number between 0 and 1	Both
ROUND (a,n)	Rounds a to n places	Both
REMAINDER(a,b)	Same as MOD	SQL Anywhere
SIGN (a)	Sign of a	Both
SIN (a)	Sine of a, with a in radians	Both
SQRT (a)	Square root of a	Both
TAN (a)	Tangent of a, with a in radians	Both
TRUNCATE (a, n)	Truncate a to n places	SQL Anywhere

Notes

- In addition to PI(), SQL Anywhere also uses PI(*).
- The value of n in ROUND and TRUNCATE is relative to the decimal point. Negative numbers are to the left of the decimal point.

Compatibility of string functions

In the following, s is a string expression, and n is an integer expression.

Function	Description	Supported by
ASCII(s)	Integer ASCII value of first character in s	Both
CHAR(n)	Character with ASCII value n	Both
CHARINDEX (s1,s2)	Offset of s1 in s2	Both

Function	Description	Supported by
DIFFERENCE (s1,s2)	Difference between two soundex values	Both
INSERTSTR(n,s1,s2)	Inserts s2 into s1 at n	SQL Anywhere
LCASE(s)	Converts s to lower case	SQL Anywhere
LOWER(s)	Converts s to lower case	Both
LEFT(s,n)	Leftmost n characters in s	Both
LENGTH(s)	Number of characters in s	Both
LOCATE(s1, s2[,n])	Offset of s2 in s1, starting at n	SQL Anywhere
LTRIM(s)	Removes leading blanks	Both
PATINDEX (...)	Integer representing starting position of first occurrence of a pattern in s	Both
PLAN(s)	Optimization strategy of the SELECT statement s	SQL Anywhere
REPLICATE (S,n)	Concatenates strings	Both
RIGHT(s,n)	Rightmost n characters in s	Both
RTRIM(s)	Removes trailing blanks	Both
SIMILAR(s1, s2)	An integer representing similarity between s1 and s2	SQL Anywhere
SOUNDEX(s)	A number representing the sound of s	Both
SPACE (n)	Returns n spaces	Both
STR(n)	String representation of n	SQL Server
STRING(s1,...)	Concatenates and converts to strings	SQL Anywhere
STUFF (...)	Deletes and inserts characters	SQL Server
SUBSTR(s, n1, n2)	Substring of s starting at n1, of length n2	SQL Anywhere
SUBSTRING(s,n1,n2)	Substring of s starting at n1, of length n2	Both
TRACEBACK(*)	Traceback of procedures and triggers executing when the most recent error occurred.	SQL Anywhere
TRIM(*)	Removes leading and trailing blanks	SQL Anywhere
UCASE(s)	Converts s to upper case	SQL Anywhere

Function	Description	Supported by
UPPER(s)	Converts s to upper case	Both

Notes

- The SQL Server SOUNDEX and DIFFERENCE return a four-character code, while SQL Anywhere returns an integer.

Compatibility of date and time functions

The arguments are not listed in the date and time function tables. The Transact-SQL date and time functions are now supported as alternatives to SQL Anywhere's own functions. For full descriptions, see the SQL Server and SQL Anywhere documentation.

Date and time functions

Function	Description	Supported by
getdate	Current system date and time	both
datename	Name of part of the datetime value	both
datepart	Value of part of the datetime value	both
datediff	Datetime difference	both
dateadd	Adds datetime parts to the date	both

These date and time functions are used as follows:

- **getdate ()** Returns the current date and time. The following aexample displays the system date and time.

```
SELECT getdate()
```

- **datename (datepart, date)** Returns the name of the specified part (such as the month "June") of a DATETIME value, as a character string. If the result is numeric, such as *23* for the day, it is still returned as a character string. For example, the following statement displays the value May.

```
SELECT datename( month , '1987/05/02' )
```

- **datepart(datepart, date)** Returns an integer value for the specified part of a DATETIME value. For example, the following statement displays the value 5.

```
SELECT datepart( month , '1987/05/02' )
```

- **datediff(datepart, date1, date2)** Returns date2 - date1, measured in the specified date part. For example, the following statement displays the value 102.

```
SELECT datediff( month, '1987/05/02',
'1995/11/15')
```

- **dateadd(datepart, numeric_expression, date)** Returns the date produced by adding the specified number of the specified date parts to the date. The *numeric_expression* can be any numeric type; the value is truncated to an integer. For example, the following statement displays 1995-11-02 00:00:00.000.

  ```
  SELECT dateadd( month, 102, '1987/05/02' )
  ```

  The following table displays allowed values of datepart.

Date Part	Abbreviation	Values
year	yy	1753 - 9999
quarter	qq	1 - 4
month	mm	1 - 12
week	wk	1 - 54
day	dd	1 - 31
dayofyear	dy	1 - 366
weekday	dw	1 - 7 (Sun.-Sat.)
hour	hh	0 - 23
minute	mi	0 - 59
second	ss	0 - 59
millisecond	ms	0 - 999

SQL Anywhere date and time functions

Function	Description	Supported by
YEAR[S] form 1	Year of the given date	SQL Anywhere
YEARS form 2	Date difference, as number of whole years	SQL Anywhere
YEARS form 3	Adds dates, as number of whole years	SQL Anywhere
MONTH[S] form 1	Month of the given date	SQL Anywhere
MONTHS form 2	Date difference, as number of whole months	SQL Anywhere
MONTHS form 3	Adds dates, as number of whole months	SQL Anywhere
WEEK[S] form 1	Week of the given date	SQL Anywhere
WEEKS form 2	Date difference, as number of whole weeks	SQL Anywhere

Function	Description	Supported by
WEEKS form 3	Adds dates, as number of whole weeks	SQL Anywhere
DAY[S] form 1	Day of the given date	SQL Anywhere
DAYS form 2	Date difference, as number of whole days	SQL Anywhere
DAYS form 3	Adds dates, as number of whole days	SQL Anywhere
DOW	Day of the week	SQL Anywhere
HOUR[S] form 1	Hour of the given date/time	SQL Anywhere
HOURS form 2	Date/time difference, as number of whole hours	SQL Anywhere
HOURS form 3	Adds date/times, as number of whole hours	SQL Anywhere
MINUTE[S] form 1	Minute of the given date/time	SQL Anywhere
MINUTES form 2	Date/time difference, as number of whole minutes	SQL Anywhere
MINUTES form 3	Adds date/times, as number of whole minutes	SQL Anywhere
SECOND[S] form 1	Second of the given date/time	SQL Anywhere
SECONDS form 2	Date/time difference, as number of whole seconds	SQL Anywhere
SECONDS form 3	Adds date/times, as number of whole seconds	SQL Anywhere
NOW(*)	Current date and time	SQL Anywhere
TODAY(*)	Today's date	SQL Anywhere

Compatibility of data type conversion functions

Both SQL Anywhere and SQL Server convert many data types implicitly, without the need to call a function to carry out the conversion. Both also provide explicit functions to carry out conversions.

Function	Description	Supported by
convert(datatype, expr [,style])	Converts expr to datatype	Both
hextoint (s)	Returns the integer equivalent of a hexadecimal string.	Both

Function	Description	Supported by
inttohex (n)	Returns the hexadecimal equivalent of an integer	Both

The CONVERT function takes two or three arguments:

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

The value *expression* is converted into data type *datatype*. The optional third parameter, *format-style* is the display format to use for the converted data (if converting to a date or time) or for the input data (if converting from a date or time). This is used for money data types (not supported for CONVERT in SQL Anywhere) and for datetime data types converted to strings to distinguish between, for example, May 2, 1987 and 1987/05/02. The *format-style* argument is a style code in the range 0 through 12 or 100 through 112, describd in the following table. A value greater than 100 produces a four-character year (yyyy) while values less than 100 yield a two-character year (yy). In the table, the two-character year is shown.

(yy)	(yyyy)	Output
0	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	mmm dd yyyy hh:mi:ss:mmmAM (or PM)
10	110	mm-dd-yy
11	111	yy/mm/dd
12	112	yymmdd

The following statements illustrate the use of these format styles:

```
SELECT CONVERT( CHAR( 20 ), order_date, 104 )
FROM sales_order
```

order_date
16.03.1993
20.03.1993
23.03.1993
25.03.1993

```
SELECT CONVERT( CHAR( 20 ), order_date, 7 )
FROM sales_order
```

order_date
mar 16, 93
mar 20, 93
mar 23, 93
mar 25, 93

The additional data type conversion functions supported by SQL Anywhere are as follows:

Function	Description
DATE (*expr*)	Converts the expression to a date
DATETIME (*expr*)	Converts the expression to a timestamp
DATEFORMAT (*date, expr*)	Converts a date to a string using the format specified by expr
YMD(*ny, nm, nd*)	Returns a date value corresponding to the supplied year, month, and day values
CAST(*expr* **AS** - *data*type)	Converts expr to data-type
STRING (*expr*)	Converts expr to a string

Compatibility of miscellaneous functions

SQL Anywhere supports a set of functions that do not fit neatly into any of the above categories.

Function	Description
ARGN (*n, expr-list*)	Returns the n'th expression in expr-list
COALESCE (*expr-list*)	The value of the first expression that is not NULL.
IFNULL (*a, b, c*)	If a is null, returns b. If a is not null, then c.
ISNULL (*expr-list*)	Same as COALESCE.
NUMBER(*)	Generates numbers starting at 1.

Notes

- These miscellaneous functions are not supported by SQL Server.

Compatibility of text and image functions

The SQL Server textptr function is supported by SQL Anywhere. The syntax is:

textptr (*text_columname*)

The function returns the text pointer value, a 16-byte binary value.

The SQL Server textvalid function is not currently supported by SQL Anywhere.

Compatibility of system functions

The following table shows the SQL Server system functions and their status in SQL Anywhere:

Function	Status
col_length	Implemented
col_name	Implemented
db_id	Implemented
db_name	Implemented
index_col	Implemented
object_id	Implemented
object_name	Implemented
proc_role	Always returns 0
show_role	Always returns NULL

Function	Status
tsequal	Implemented
user_id	Implemented
user_name	Implemented
suser_id	Implemented
suser_name	Implemented
datalength	Implemented
curunreserved pgs	Not implemented
data_pgs	Not implemented
host_id	Not implemented
host_name	Not implemented
lct_admin	Not implemented
reserved_pgs	Not implemented
rowcnt	Not implemented
used_pgs	Not implemented
valid_name	Not implemented
valid_user	Not implemented

Notes

- Some of the system functions are implemented in SQL Anywhere as stored procedures.
- The db_id, db_name, and datalength functions are implemented as built-in functions.

The implemented system functions are described in the following table.

System function	Description
col_length(*table-name, column-name*)	Returns the defined length of column
col_name(*table-id, column-id* [, *database-id*])	Returns the column name
db_id([*database-name*])	Returns the database ID number
db_name([*database-id*])	Returns the database name
index_col (*table-name, index-id, key_#* [, *userid*])	Returns the name of the indexed column
object_id (*object-name*)	Returns the object ID

System function	Description
object_name (*object-id* [, *database-id*])	Returns the object name
tsequal(*timestamp, timestamp2*)	Compares timestamp values to prevent update on a row that has been modified since it was selected
user_id([*user-name*])	Returns an integer user identification number. This does not return the SQL Anywhere user ID
user_name([*user-id*])	Returns the user ID (user name in SQL Server)
suser_id([*user-name*])	Returns an integer user identification number
suser_name([*user-id*])	Returns the user ID (server user name in SQL Server)
datalength(*expression*)	Returns the length of expression in bytes

Building compatible search conditions

Search conditions are used in WHERE clauses, ON clauses (in SQL Anywhere), and in HAVING clauses to qualify or disqualify rows or groups from appearing in the result set. A search condition is a combination of expressions that has a value of TRUE, FALSE, or UNKNOWN.

Compatibility of expressions is the subject of "Building compatible expressions" on page 561. For a search condition to be compatible, the expression or expressions of which it is composed must be compatible. This section assumes the expressions are compatible.

There are several kinds of conditions, which are discussed in the following sections.

- Comparison conditions
- BETWEEN and LIKE conditions
- IN conditions
- ALL or ANY conditions
- EXISTS conditions
- IS NULL conditions
- Conditions with logical operators, including AND and OR

Compatibility of comparison conditions

Comparison conditions take the form

```
expression comparison-operator expression
```

The comparison operators are generally compatible between SQL Server and SQL Anywhere.

Compatibility of comparison operators

Operator	Description	Supported by
=	Equal to	Both
>	Greater than	Both
<	Less than	Both

Operator	Description	Supported by
>=	Greater than or equal to	Both
<=	Less than or equal to	Both
<>	Not equal to	Both
!=	Not equal to	Both
!>	Not greater than	Both
!<	Not less than	Both

Trailing blanks in character data are ignored for comparison purposes by SQL Server. The behavior of SQL Anywhere when comparing strings is controlled by an option when creating the database.

For more information, see "Configuring SQL Anywhere for Transact-SQL compatibility" on page 544.

Compatibility of BETWEEN and LIKE conditions

The BETWEEN condition is of the form

```
expr [NOT] BETWEEN start-expr AND end-expr
```

and is compatible with both SQL Server and SQL Anywhere.

The LIKE condition is of the form

```
expression [NOT] LIKE pattern [ESCAPE escape-expr]
```

The ESCAPE clause can be used to cause characters that otherwise take on a special meaning in a search pattern to be treated literally in the comparison. The ESCAPE clause is supported by SQL Anywhere only.

The *pattern* consists of characters and special wildcard characters that match one or more characters in a string. The pattern may be made up from variables, constants, and expressions.

SQL Anywhere supports the same pattern matching that is supported by SQL Server, using the following wild cards:

- **underscore (_)** Matches any character
- **percent (%)** Matches any string of zero or more characters
- **[ ]** Matches sets or ranges of characters

For more information on these conditions, see the section "LIKE conditions" in the chapter "Watcom-SQL Language Reference".

Compatibility of IN conditions

The IN condition tests if an expression equals any of a set of listed values. The IN condition is of the form:

```
expression [NOT] IN ( value-expr1 [, value-expr2 ]
... )
```

or

```
expression [NOT] IN ( subquery )
```

IN conditions are compatible between SQL Server and SQL Anywhere.

Compatibility of ALL and ANY conditions

The ALL and ANY conditions test an expression against the results of a subquery. The ALL and ANY conditions take the form:

```
expression comparison-operator { ANY | ALL }
(subquery)
```

ANY and ALL subqueries are compatible between SQL Server and SQL Anywhere. SQL Anywhere supports SOME as a synonym for ANY.

Compatibility of EXISTS conditions

The EXISTS condition returns TRUE if a subquery result contains at least one row, and FALSE if the subquery result contains no rows. The EXISTS condition is of the form:

```
EXISTS (subquery)
```

The EXISTS condition is implemented in both SQL Server and SQL Anywhere.

Compatibility of IS NULL conditions

Expressions can be tested to see if they are NULL using the IS [NOT] NULL condition. The IS [NOT] NULL condition is implemented in both SQL Server and SQL Anywhere.

The NOT NULL condition is of the form

```
expression IS [NOT] NULL
```

Providing estimates for conditions

SQL Anywhere supports estimates of the percentage of rows that satisfy a condition, to assist in improving query performance. An estimate on a search condition is supplied as follows:

```
( condition, estimate )
```

SQL Server does not support explicit estimates.

Conditions using logical operators

Logical operators include NOT: a compatible unary operator which takes the following form:

```
NOT condition
```

The NOT operator is compatible between SQL Server and SQL Anywhere.

Logical operators also include operators which test explicitly for the value of a condition:

```
condition IS [NOT] { TRUE | FALSE | UNKNOWN }
```

These operators are supported by SQL Anywhere only.

Logical operators also include the AND and OR operators, which allow search conditions to be combined. These combined conditions take the following form:

```
condition { AND | OR } condition
```

The AND and OR operators are supported by both SQL Server and SQL Anywhere. When more than one logical operator is used in a statement, AND operators are evaluated before OR operators by default. You can change the order of precedence with parentheses.

Other language elements

This section describes language elements other than those described in the preceding sections.

Compatible comment indicators

In SQL Anywhere, you can use one of several identifiers to mark off comments in SQL statements.

SQL Anywhere now supports the following comment indicators:

- **-- (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. Of these comment markers, the double-hyphen style and the slash-asterisk style are compatible with Transact-SQL.

Transact-SQL statement reference

This section describes the supported syntax of Transact-SQL statements.

☞ Many SQL statements are common to Watcom-SQL and Transact-SQL, and for information on these, you should look in the chapter "Watcom-SQL Language Reference".

This section contains the following, in alphabetical order:

- Syntax description of statements implemented primarily for Transact-SQL compatibility, and for which there are alternatives in the Watcom-SQL dialect. For example, the READ TEXT statement is a Transact-SQL statement for which the SQL Anywhere GET DATA statement is an alternative.
- Syntax description for statements implemented primarily for Transact-SQL compatibility, and which are different in approach to the ISO/ANSI standard on which Watcom-SQL is based. For example, the BEGIN TRANSACTION statement introduces some functionality that Watcom-SQL does not have, but results in not all COMMIT statements making changes to the database permanent, which is contrary to ISO/ANSI behavior.
- Comments on statements documented in the chapter "Watcom-SQL Language Reference", for which the Transact-SQL syntax is different.
- Comments on some unsupported Transact-SQL statements, with pointers to the ways in which SQL Anywhere implements similar functionality. For example, the CREATE DEFAULT and CREATE RULE statements available in SQL Server are not supported in SQL Anywhere, but similar functionality is provided in the Watcom-SQL CREATE DATATYPE statement.

This section is a guide to the SQL supported by SQL Server and SQL Anywhere; it is not an exhaustive comparison. For those creating applications that must work with both SQL Server and SQL Anywhere, you should directly compare the descriptions of supported syntax in "Watcom-SQL Statements" and in this section against the SQL Server documentation.

Even though they are documented separately, you can use statements documented in this section together with Watcom-SQL statements.

Statements that are part of the Transact-SQL procedure language are documented separately. This includes control statements (IF, WHILE, and so on), CREATE PROCEDURE and CREATE TRIGGER statements, and other statements used primarily in procedures and triggers, such as RAISERROR. For information on these statements, see the chapter "Transact-SQL Procedure Language".

Transact-SQL BEGIN TRANSACTION statement

Syntax

BEGIN TRAN[SACTION] [*transaction-name*]

Purpose

To begin a user-defined transaction.

Usage

Anywhere.

Authorization

None.

Side effects

None.

See also

"Transact-SQL COMMIT statement" on page 584.

Description

The optional parameter *transaction-name* is the name assigned to this transaction. It must be a valid identifier. Use transaction names only on the outermost pair of nested BEGIN/COMMIT or BEGIN/ROLLBACK statements.

When executed inside a transaction, the BEGIN TRANSACTION statement increases the nesting level of transactions by one. The nesting level is decreased by a COMMIT statement. When transactions are nested, only the outermost COMMIT makes the changes to the database permanent.

Chained and unchained modes

SQL Server and SQL Anywhere both have two transaction modes. The default SQL Server transaction mode, called unchained mode, commits each statement individually, unless an explicit BEGIN TRANSACTION statement is executed to start a transaction.

SQL Anywhere supports both chained and unchained modes. You can control the mode by setting the CHAIN database option. The default setting in SQL Anywhere is ON, in which case SQL Anywhere runs in the ISO SQL/92 compatible chained mode. The default for SQL Server is that CHAINED be set to OFF. In unchained mode, a transaction is implicitly started before any data retrieval or modification statement. These statements include: DELETE, INSERT, OPEN, FETCH, SELECT, and UPDATE. You must still explicitly end the transaction with a COMMIT or ROLLBACK statement.

You cannot alter the CHAINED option within a transaction.

> ***Caution***
> *When calling a stored procedure, you should ensure that it operates correctly under the required transaction mode.*

For more information about the CHAINED option and the chained mode, see "SET OPTION statement" in the chapter "Watcom-SQL Statements".

The current nesting level is held in the global variable @@trancount. The @@trancount variable has a value of zero before a BEGIN TRANSACTION statement is executed, and only a COMMIT executed when @@trancount is equal to one makes changes to the database permanent.

A ROLLBACK statement, without a transaction or savepoint name, always rolls back statements to the outermost BEGIN TRANSACTION (explicit or implicit) statement, and cancels the entire transaction.

Example

The following batch reports successive values of @@trancount as 0, 1, 2, 1, 0.

```
PRINT @@trancount
BEGIN TRANSACTION
PRINT @@trancount
BEGIN TRANSACTION
PRINT @@trancount
COMMIT TRANSACTION
PRINT @@trancount
COMMIT TRANSACTION
PRINT @@trancount
```

@@trancount values in SQL Server and SQL Anywhere

You should not rely on the value of@@trancount for more than keeping track of the number of explicit BEGIN TRANSACTION statements that have been issued.

When SQL Server starts a transaction implicitly, the @@trancount variable is set to 1. SQL Anywhere does not set the @@trancount value to 1 when a transaction is started implicitly. Consequently, the SQL Anywhere @@trancount variable has a value of zero before any BEGIN TRANSACTION statement, (even though there is a current transaction), while in SQL Server (in chained mode) it has a value of 1.

For transactions starting with a BEGIN TRANSACTION statement, @@trancount has a value of 1 in both SQL Anywhere and SQL Server after the BEGIN TRANSACTION statement. If a transaction is implicitly started with a different statement, and a BEGIN TRANSACTION statement is then executed, @@trancount has a value of 1 in SQL Anywhere, and a value of 2 in SQL Server after the BEGIN TRANSACTION statement.

Transact-SQL COMMIT statement

Syntax

COMMIT TRAN[SACTION] [*transaction-name*]

Purpose

To terminate a user-defined transaction or make changes to the database permanent.

Usage

Anywhere.

Authorization

None.

Side effects

None.

See also

"Transact-SQL BEGIN TRANSACTION statement" on page 582.

Description

The optional parameter *transaction-name* is the name assigned to this transaction. It must be a valid identifier. Use transaction names only on the outermost pair of nested BEGIN/COMMIT or BEGIN/ROLLBACK statements.

When executed inside a transaction, the COMMIT statement decreases the nesting level of transactions by one. When transactions are nested, only the outermost COMMIT makes the changes to the database permanent.

For a discussion of transaction nesting in SQL Server and SQL Anywhere, see "Transact-SQL BEGIN TRANSACTION statement" on page 582.

Savepoints and the ROLLBACK statement are discussed in the chapter "Watcom-SQL Language Reference".

Example

The following batch reports successive values of @@trancount as 0, 1, 2, 1, 0.

```
PRINT @@trancount
BEGIN TRANSACTION
PRINT @@trancount
BEGIN TRANSACTION
PRINT @@trancount
COMMIT TRANSACTION
PRINT @@trancount
```

```
COMMIT TRANSACTION
PRINT @@trancount
```

Transact-SQL CREATE DEFAULT and CREATE RULE statements

SQL Anywhere does not support the Transact-SQL CREATE DEFAULT statement or CREATE RULE statement. The Watcom-SQL CREATE DATATYPE statement allows a default and a rule (called a CHECK condition) to be incorporated into the definition of a user-defined data type, and so provides similar functionality to the Transact-SQL CREATE DEFAULT and CREATE RULE statements.

In SQL Server, the CREATE DEFAULT statement creates a named default which can be used as a default value for columns by binding the default to a particular column or which can be used as a default value for all columns of a user-defined data type by binding the default to the data type. A default is bound to a data type or column using the sp_bindefault system procedure.

The CREATE RULE statement creates a named rule which can be used to define the domain for columns by binding the rule to a particular column or which can be used as a rule for all columns of a user-defined data type by binding the rule to the data type. A rule is bound to a data type or column using the sp__bindrule system procedure.

In SQL Anywhere, a user-defined data type can have a default value and a CHECK condition associated with it, which are applied to all columns defined on that data type. The user-defined data type is created using the CREATE DATATYPE statement.

Default values and rules, or CHECK conditions, can be defined for individual columns using the CREATE TABLE statement or the ALTER TABLE statement. For a description of the SQL Anywhere syntax for these statements, see "Watcom-SQL Statements".

Transact-SQL CREATE INDEX statement

The supported CREATE INDEX statement syntax is described in "CREATE INDEX statement" in the chapter "Watcom-SQL Statements".

SQL Server indexes can be either **clustered** or **nonclustered**. A clustered index almost always retrieves data faster than a nonclustered index. Only one clustered index is permitted per table.

SQL Anywhere does not support clustered indexes. The CLUSTERED and NONCLUSTERED keywords are allowed by SQL Anywhere, but no action is taken.

SQL Anywhere also allows, by ignoring, the following keywords:

- FILLFACTOR
- IGNORE_DUP_KEY
- SORTED_DATA
- IGNORE_DUP_ROW
- ALLOW_DUP_ROW

Physical placement of an index is carried out differently in SQL Server and in SQL Anywhere. The *ON segment-name* clause is supported in SQL Anywhere, but *segment-name* refers to a SQL Anywhere dbspace.

Index names must be unique on a given table for both SQL Anywhere and SQL Server.

Transact-SQL CREATE MESSAGE statement

Syntax

CREATE MESSAGE *message-num*
... **AS** *'message-text"*

Purpose

To add a user-defined message to the sysusermessages system table for use by print and raiserror calls.

Usage

Anywhere.

Authorization

Resource authority

Side effects

Automatic commit.

See also

"Transact-SQL PRINT statement" in the chapter "Transact-SQL Procedure Language"
"Transact-SQL RAISERROR statement" in the chapter "Transact-SQL Procedure Language"

Description

The CREATE MESSAGE statement does not exist in SQL Server. It is an provided in SQL Anywhere as an alternative to the sp_addmessage system procedure.

The replaceable text in the syntax is as follows:

- **message_num** The message number of the message to add. The message number for a user-defined message must be 20000 or greater.

- **message_text** The text of the message to add. The maximum length is 255 bytes. Print and raiserror recognize placeholders in the message text to print out. A single message can contain up to 20 unique placeholders in any order. These placeholders are replaced with the formatted contents of any arguments that follow the message when the text of the message is sent to the client.

  The placeholders are numbered to allow reordering of the arguments when translating a message to a language with a different grammatical structure. A placeholder for an argument appears as "%nn!", a percent sign (%), followed by an integer from 1 to 20, followed by an exclamation mark (!). The integer represents the argument number in the string in the argument list. "%1!" is the first argument in the original version, "%2!" is the second argument, and so on.

The language argument for sp_addmessage is not currently supported.

Transact-SQL CREATE SCHEMA statement

SQL Anywhere does not support the use of REVOKE statements within the CREATE SCHEMA statement, and does not allow its use within Transact-SQL batches or procedures.

☞ The CREATE SCHEMA statement supported in SQL Anywhere is described in "CREATE SCHEMA statement" in the chapter "Watcom-SQL Statements".

Transact-SQL CREATE TABLE statement

SQL Anywhere does not support named constraints or named defaults, but does support user-defined data types which allow constraint and default definitions to be encapsulated in the data type definition. It also supports explicit defaults and CHECK conditions in the CREATE TABLE statement.

You can create a temporary table by preceding the table name in a CREATE TABLE statement with a pound sign (#). These temporary tables are SQL Anywhere declared temporary tables, and are available only in the current connection. For information about declared temporary tables in SQL Anywhere, see "DECLARE TEMPORARY TABLE statement" in the chapter "Watcom-SQL Statements".

By default, columns in SQL Server default to NOT NULL, whereas in SQL Anywhere the default setting is NULL, to allow NULL values. This setting can be controlled using the allow_nulls_by_default option. For information on this option, see "Setting options for Transact-SQL compatibility" on page 546. You should explicitly specify NULL or NOT NULL to make your data definition statements transferable.

Physical placement of a table is carried out differently in SQL Server and in SQL Anywhere. The *ON segment-name* clause is supported in SQL Anywhere, but *segment-name* refers to a SQL Anywhere dbspace.

For information about the CREATE TABLE statement supported in SQL Anywhere is described in "CREATE TABLE statement" in the chapter "Watcom-SQL Statements".

Transact-SQL DATABASE statements

The Transact-SQL statements CREATE DATABASE, DROP DATABASE, DUMP DATABASE, and LOAD DATABASE are not supported in SQL Anywhere.

SQL Server and SQL Anywhere have different models of creating and dropping databases, reflecting the different uses for the two products.

In SQL Server, users connect to a single server, and can access the different databases running on that server. From the server, the CREATE DATABASE statement is used to create a new database, and DROP DATABASE is used to delete a database; DUMP DATABASE and DUMP TRANSACTION are used to back up a database (with its transaction log) and a transaction log only. The LOAD DATABASE and LOAD TRANSACTION statements load a backup copy of a database or transaction log.

In SQL Anywhere, users connect to a single database running on an engine or server. The SQL Anywhere engine or server can load and unload databases dynamically.

- To create a SQL Anywhere database, you use the initialization utility. For information on this utility, see "The Initialization utility" in the chapter "SQL Anywhere Components".
- To erase a SQL Anywhere database, you use the erase utility. For information on this utility, see "The Erase utility" in the chapter "SQL Anywhere Components".

- To back up a SQL Anywhere database, you use the backup utility. For information on this utility, see "The Backup utility" in the chapter "SQL Anywhere Components". Transaction logs are applied to the database using database engine command-line switches.

Transact-SQL DELETE statement

Subject to the expressions being compatible, the syntax of the DELETE statement is compatible between SQL Server and SQL Anywhere.

Transact-SQL DISK statements

SQL Anywhere and SQL Server use different models for managing devices and disk space, reflecting the different uses for the two products. While SQL Server sets out a comprehensive resource management scheme using a variety of Transact-SQL statements, SQL Anywhere is designed to be able to manage its own resources automatically.

For information on SQL Anywhere disk management, see "Working with databases" in the chapter "Working with Database Objects", as well as the following statements in "Watcom-SQL Statements":

- ALTER DBSPACE statement
- ALTER TABLE statement
- CREATE DBSPACE statement
- CREATE INDEX statement
- CREATE TABLE statement

SQL Anywhere does not support Transact-SQL DISK statements, such as DISK INIT, DISK MIRROR, DISK REFIT, DISK REINIT, DISK REMIRROR, and DISK UNMIRROR.

Transact-SQL GRANT and REVOKE statements

In SQL Server, users are created on a server-wide basis. In SQL Anywhere, users are created for individual databases.]

For an overview of users and groups in SQL Server and SQL Anywhere, see "Users and groups in SQL Anywhere and SQL Server" on page 541. SQL Anywhere provides system procedures that allow SQL Server-like management of users and groups, which are listed in that section and in "SQL Server system procedures" on page 601

Database object permissions

The SQL Server and SQL Anywhere GRANT and REVOKE statements for granting permissions on individual database objects are very similar. Both DBMS's allow SELECT, INSERT, DELETE, UPDATE, and REFERENCES permissions on database tables and views, and UPDATE permissions on selected columns of database tables. Both DBMS's allow EXECUTE permissions to be granted on stored procedures.

For example, the following statement is valid in both SQL Server and SQL Anywhere:

```
GRANT INSERT, DELETE
ON TITLES
TO MARY, SALES
```

This statement grants permission to use the INSERT and DELETE statements on the titles table to user Mary and to the sales group.

The WITH GRANT OPTION clause, allowing the recipient of permission to grant them in turn, is supported in both SQL Anywhere and SQL Server, although SQL Anywhere does not permit WITH GRANT OPTION to be used on a GRANT EXECUTE statement.

Database-wide permissions

SQL Server and SQL Anywhere use different models for database-wide user permissions. These are discussed in "Users and groups in SQL Anywhere and SQL Server" on page 541. SQL Anywhere employs DBA permissions to allow a user full authority within a database. This permission is enjoyed by the System Administrator in SQL Server, for all databases on a server. However, DBA authority on a SQL Anywhere database is different from the permissions of a SQL Server Database Owner, who must use the SQL Server setuser statement to gain permissions on objects owned by other users.

SQL Anywhere employs RESOURCE permissions to allow a user the right to create objects in a database. A closely corresponding SQL Server permission is GRANT ALL used by a Database Owner.

Transact-SQL INSERT statement

Subject to the expressions being compatible, the syntax of the INSERT statement is compatible between SQL Server and SQL Anywhere. You should ensure that expressions being inserted or updated are compatible.

Transact-SQL joins

In Transact-SQL, joins are specified in the WHERE clause, using the following syntax:

Start of select, update, insert, delete, or subquery
FROM {*table-list* | *view-list*} **WHERE** [**NOT**]
... [*table-name.* | - *view name.*]*column-name*
join-operator [*table-name.*| *view-name.*]*column_name*
... [{ **AND** | **OR** } [**NOT**]
[*table-name.*| *view-name.*]*column_name*
join-operator [*table-name.*| *view-name.*]*column-name*]...
End of select, update, insert, delete, or subquery

The *join_operator* in the WHERE clause may be any of the comparison operators listed in the section "Compatibility of operators" on page 563, or may be either of the two outer-join operators.

The Transact-SQL outer-join operators are supported in SQL Anywhere as an alternative to the native SQL/92 syntax.

For joins other than outer joins, SQL Anywhere provides a SQL/92 syntax also, in which the joins are placed in the FROM clause rather than the WHERE clause. For information about joins in SQL Anywhere and in SQL/92, see "FROM clause" in the chapter "Watcom-SQL Statements". Specifying the join in the WHERE clause is also accepted. All joins other than outer joins can be specified in the WHERE clause so that they are compatible with both SQL Anywhere and SQL Server.

There is some ambiguity in the meaning of the outer join operators in some situations where more than two tables are queried. For multi-table outer joins, you should confirm that the results of the query are identical in SQL Server and SQL Anywhere.

Transact-SQL READTEXT statement

Syntax

READTEXT *table-name.column-name*
... *text-pointer offset size* [**HOLDLOCK**]

Purpose

Reads text and image values, starting from a specified offset and reading a specified number of bytes.

Usage

Anywhere.

Authorization

None.

Side effects

None.

Description

SQL Server supports the following clause, which is not supported by SQL Anywhere:

```
USING { BYTES | CHARS | CHARACTERS }
```

These options are identical for all single-byte character sets. SQL Anywhere uses bytes only, which is the SQL Server default setting.

Transact-SQL ROLLBACK statement

The Transact-SQL ROLLBACK TRANSACTION syntax is supported.

For a discussion of Transact-SQL transaction management in SQL Anywhere, see "Transact-SQL BEGIN TRANSACTION statement" on page 582.

Transact-SQL SELECT statement

There are two criteria for writing a query that runs on both SQL Anywhere and SQL Server databases:

- Ensure that the data types, expressions, and search conditions in the query are compatible.
- Ensure that the syntax of the SELECT statement itself is compatible.

This section is concerned with compatible SELECT statement syntax, and assumes compatible data types, expressions, and search conditions. The examples assume a quoted_identifier setting of OFF: the default SQL Server setting, but not the default SQL Anywhere setting.

The following subset of the Transact-SQL SELECT statement is supported in SQL Anywhere.

Syntax

SELECT [**ALL** | **DISTINCT**] *select-list*
...[**INTO** *#temporary-table-name*]
...[**FROM** *table-spec* [**HOLDLOCK** | **NOHOLDLOCK**],
... *table-spec* [**HOLDLOCK** | **NOHOLDLOCK**], ...]
...[**WHERE** *search-condition*]
...[**GROUP BY** *column-name*, ...]
...[**HAVING** *search-condition*]
...[[**ORDER BY** *expression* [**ASC** | **DESC**], ...] |
| [**ORDER BY** *integer* [**ASC** | **DESC**], ...] |

Parameters

select-list:
{ *table-name* | *alias-name* = *expression* }...

table-spec:

[*owner* .]*table-name*
... [[**AS**] *correlation-name*]
... [(**INDEX** *index_name* [**PREFETCH** *size*][**LRU** | **MRU**])]

alias-name:
identifier | '*string*' | "*string*"

The following keywords and clauses of the Transact-SQL SELECT statement syntax are not supported by SQL Anywhere:

- The SHARED keyword.
- The COMPUTE clause.
- The FOR BROWSE clause.
- The GROUP BY ALL clause.

Notes

- The INTO table_name clause, which creates a new table based on the SELECT statement result set, is supported only for declared temporary tables where the table name starts with a #. Declared temporary tables exist for a single connection only.
- SQL Anywhere does not support the Transact-SQL extension to the GROUP BY clause allowing references to columns and expressions not used for creating groups. In SQL Server, this extension produces summary reports.
- The FOR READ ONLY clause and the FOR UPDATE clause are parsed, but have no effect.
- The performance parameters part of the table specification is parsed, but has no effect.
- The HOLDLOCK keyword is supported by SQL Anywhere. It makes a shared lock on a specified table or view more restrictive by holding it until the completion of a transaction (instead of releasing the shared lock as soon as the required data page is no longer needed, whether or not the transaction has been completed). For the purposes of the table for which the HOLDLOCK is specified, the query is carried out at isolation level 3.
- The HOLDLOCK option applies only to the table or view for which it is specified, and only for the duration of the transaction defined by the statement in which it is used. Setting transaction isolation level 3 option of the set command implicitly applies a holdlock for each select within a transaction. You cannot specify both a HOLDLOCK and NOHOLDLOCK option in a query.
- The NOHOLDLOCK keyword is recognized by SQL Anywhere, but has no effect.

- Transact-SQL uses the SELECT statement to assign values to local variables:

  ```
  SELECT @localvar = 42
  ```

  The corresponding statement in the Watcom-SQL dialect is the SET statement:

  ```
  SET localvar = 42
  ```

  However, the Transact-SQL SELECT to assign values to variables is supported inside batches.

- The following clauses of the Watcom-SQL SELECT statement syntax are not supported by SQL Server:
  - INTO host-variable-list
  - INTO variable-list.
  - Parenthesized queries.
- A major difference between Watcom-SQL and Transact-SQL SELECT statements is in the way that joins are specified for multi-table queries.

  For information on SQL Anywhere support for Transact-SQL joins, see "Transact-SQL joins" on page 591.

Transact-SQL SET statement

In SQL Server, the SET statement sets database options for the duration of a user's connection. In SQL Anywhere this task is carried out by the SET TEMPORARY OPTION statement, while SET is used to assign values to variables. The following statements show the difference in the syntaxes for setting options:

- Set the isolation level to 3, in SQL Server:

  ```
  SET TRANSACTION ISOLATION LEVEL 3
  ```

- Set the isolation level to 3, in SQL Anywhere:

  ```
  SET OPTION PUBLIC.ISOLATION_LEVEL = 3
  ```

The following two statements show the difference in the syntaxes for setting the values of variables:

- Set a variable named @var_name to a value of 3, in SQL Server:

  ```
  SELECT @var_name = 3
  ```

- Set a variable named var_name to a value of 3, in SQL Anywhere:

```
SET var_name = 3
```

SQL Anywhere provides support for the Transact-SQL SET statement for a set of options that are particularly useful for compatibility. The following options can be set using the Transact-SQL SET statement in SQL Anywhere as well as in SQL Server:

- **SET ANSINULL { ON | OFF }** The default behavior for comparing values to NULL in SQL Anywhere and SQL Server is different. Setting ANSINULL to OFF provides Transact-SQL compatible comparisons with NULL
- **SET ANSI_PERMISSIONS { ON | OFF }** The default behavior in SQL Anywhere and SQL Server regarding permissions required to carry out an UPDATE or DELETE containing a column reference is different. Setting ANSI_PERMISSIONS to OFF provides Transact-SQL compatible permissions on UPDATE and DELETE
- **SET CLOSE_ON_ENDTRANS { ON | OFF }** The default behavior in SQL Anywhere and SQL Server for closing cursors at the end of a transaction is different. Setting CLOSE_ON_ENDTRANS to OFF provides Transact-SQL compatible behavior.
- **SET QUOTED_IDENTIFIER { ON | OFF }** Controls whether strings enclosed in double quotes are interpreted as identifiers (ON) or as literal strings (OFF). For information about this option, see "Setting options for Transact-SQL compatibility" on page 546.
- **SET ROWCOUNT** *integer* The Transact-SQL ROWCOUNT option limits the number of rows fetched for any cursor to the specified integer. This includes rows fetched by re-positioning the cursor. Any fetches beyond this maximum return a warning. The option setting is considered when returning the estimate of the number of rows for a cursor on an OPEN request.

  SET ROWCOUNT also limits the number of rows affected by a searched UPDATE or DELETE statement to *integer*. This might be used, for example, to allow COMMIT statements to be performed at regular intervals to limit the size of the rollback log and lock table. The application (or procedure) would need to provide a loop to cause the update/delete to be re-issued for those rows not affected by the first operation. A simple example is given below:

```
begin
   declare @count integer

   set rowcount 20

   while(1=1) begin
```

```
            update employee set emp_lname='new_name'
            where emp_lname <> 'old_name'

            /* Stop when no rows changed */
            select @count = @@rowcount
            if @count = 0 break
            print string('Updated ',
                    @count,' rows; repeating...')
            commit
        end

        set rowcount 0
    end
```

In SQL Anywhere, if the ROWCOUNT setting is greater than the number of rows that ISQL can display, ISQL may do some extra fetches to reposition the cursor. Thus the number of rows actually displayed may be less than the number requested. Also, if any rows are re-fetched due to truncation warnings, the count may be inaccurate.

- **SET SELF_RECURSION { ON | OFF }** The self_recursion option is used within triggers to enable (ON) or prevent (OFF) operations on the table associated with the trigger from firing other triggers.
- **SET STRING_RTRUNCATION { ON | OFF }** The default behavior in SQL Anywhere and SQL Server when non-space characters are truncated on assigning SQL string data is different. Setting STRING_RTRUNCATION to ON provides Transact-SQL compatible string comparisons.
- **SET TRANSACTION ISOLATION LEVEL { 0 | 1 | 2 | 3 }** Sets the locking isolation level for the current connection, as described in "Isolation levels and consistency" in the chapter "Using Transactions and Locks". For SQL Server, only 1 and 3 are valid options. For SQL Anywhere, any of 0, 1, 2, or 3 is a valid option.

For more information on these and other database options, see "SET OPTION statement" in the chapter "Watcom-SQL Statements".

In addition, the following SET statement is allowed by SQL Anywhere for compatibility, but has no effect:

- **SET PREFETCH {ON | OFF}**

Transact-SQL UPDATE statement

Subject to the expressions being compatible, the syntax of the UPDATE statement is compatible between SQL Server and SQL Anywhere. You should ensure that expressions being inserted or updated are compatible.

Transact-SQL WRITETEXT statement

Syntax

WRITETEXT *table-name.column-name*
... *text_pointer* [**WITH LOG**] *data*

Purpose

Permits non-logged, interactive updating of an existing text or image column.

Usage

Anywhere.

Authorization

None.

Side effects

WRITETEXT does not fire triggers, and by default WRITETEXT operations are not recorded in the transaction log.

Description

Updates an existing text or image value. The update is not recorded in the transaction log unless the WITH LOG option is supplied.

Example

The following code fragment illustrates the use of the WRITETEXT statement. The SELECT statement in this example returns a single row. The example replaces the contents of the column_name column on the specified row with the value newdata.

```
EXEC SQL create variable textpointer binary(16);
EXEC SQL set textpointer =
   (  select textptr(column_name)
      from table_name where ... ) ;
EXEC SQL writetext table_name.column_name
   textpointer 'newdata' ;
```

Compatible system catalog information

SQL Server and SQL Anywhere have different system catalogs, reflecting the different uses for the two products.

In SQL Server there is a single master database containing a set of system tables holding information that applies to all databases on the server. Many databases may exist within the master database, and each has additional system tables associated with it.

In SQL Anywhere, each database exists independently, and contains its own system tables. There is no master database that contains system information on a collection of databases. Each server may run several databases at a time, dynamically loading and unloading each database as needed.

The SQL Server and SQL Anywhere system catalogs are different. The SQL Server system tables and views are owned by the special user dbo, and exist partly in the master database, partly in the sybsecurity database, and partly in each individual database; the SQL Anywhere system tables and views are owned by the special user SYS and exist separately in each database.

To assist in preparing compatible applications, SQL Anywhere provides a set of views owned by the special user dbo, which correspond to the SQL Server system tables and views. Where architectural differences make the contents of a particular SQL Server table or view meaningless in a SQL Anywhere context, the view is empty, containing just the column names and data types.

The following tables list the SQL Server system tables and their implementation in the SQL Anywhere system catalog. The owner of all tables is dbo in each DBMS.

Tables existing in each SQL Server database

Table name	Description	Data?
sysalternates	One row for each user mapped to a database user	No
syscolumns	One row for each column in a table or view, and for each parameter in a procedure	Yes
syscomments	One or more rows for each view, rule, default, trigger, and procedure, giving SQL definition statement	Yes
sysconstraints	One row for each referential and check constraint associated with a table or column	No
sysdepends	One row for each procedure, view, or table that is referenced by a procedure, view, or trigger	No

Table name	Description	Data?
sysindexes	One row for each clustered or nonclustered index, and one row for each table with no indexes, and an additional row for each table containing text or image data.	Yes
syskeys	One row for each primary, foreign, or common key; set by user (not maintained by SQL Server)	No
syslogs	Transaction log	No
sysobjects	One row for each table, view, procedure, rule, trigger default, log, and (in tempdb only) temporary object	Contains compatible data only
sysprocedures	One row for each view, rule, default, trigger, and procedure, giving internal definition	No
sysprotects	User permissions information	No
sysreferences	One row for each referential integrity constraint declared on a table or column	No
sysroles	Maps server-wide roles to local database groups No	
syssegments	One row for each segment (named collection of disk pieces)	No
systhresholds	One row for each threshold defined for the database	No
systypes	One row for each system-supplied and user-defined datatype	Yes
sysusermessages	One row for each user-defined message	Yes (this is a SQL Anywhere system table)
sysusers	One row for each user allowed in the database	Yes

Tables existing in the SQL Server master database

Table name	Description	Data?
syscharsets	One row for each character set or sort order	No
sysconfigures	One row for each user-settable configuration parameter	No
syscurconfigs	Information about configuration parameters currently being used by the server	No
sysdatabases	One row for each database on the server	No

Table name	Description	Data?
sysdevices	One row for each tape dump device, disk dump device, disk for databases, and disk partition for databases	No
sysengines	One row for each engine currently online	No
syslanguages	One row for each language (except U.S. English) known to the server	No
syslocks	Information about active locks	No
sysloginroles	One row for each server login that possesses a system- defined role	No
syslogins	One row for each valid user account	Yes
sysmessages	One row for each system error or warning	No
sysprocesses	Information about server processes	No
sysremotelogins	One row for each remote user	No
syssrvroles	One row for each server-wide role	No
sysservers	One row for each remote server	No
sysusages	One row for each disk piece allocated to a database	No

Tables existing in the SQL Server sybsecurity database

Table name	Description	Data?
sysaudits	One row for each audit record	No
sysauditoptions	One row for each global audit option	No

SQL Server system and catalog procedures

SQL Server provides system and catalog procedures to carry out many administrative functions and to obtain system information. SQL Anywhere has implemented support for some of these procedures.

System procedures are built-in stored procedures used for getting reports from and updating system tables. Catalog stored procedures retrieve information from the system tables in tabular form.

SQL Server system procedures

The following list describes the SQL Server system procedures provided in SQL Anywhere.

System procedure	Description
sp_addgroup *group-name*	Adds a group to a database
sp_addlogin *userid, password[, defdb [, deflanguage [, fullname]]]*	Adds a new user account to a database
sp_addmessage *message-num, message_text [, language]*	Adds user-defined messages to SYSUSERMESSAGES for use by stored procedure PRINT and RAISERROR calls
sp_addtype *typename, data-type, [, "identity" | nulltype]*	Creates a user-defined data type
sp_adduser *login_name [, name_in_db [, grpname]]*	Adds a new user to a database
sp_changegroup *new-group-name, userid*	Changes a user's group or adds a user to a group
sp_dboption *[dbname, optname, {true | false}]*	Displays or changes database options
sp_dropgroup *group-name*	Drops a group from a database
sp_droplogin *userid*	Drops a user from a database
sp_dropmessage *message-number [, language]*	Drops user-defined messages
sp_droptype *typename*	Drops a user-defined data type
sp_dropuser *userid*	Drops a user from a database
sp_getmessage *message-num, @msg-var output [, language]*	Retrieves stored message strings from SYSMESSAGES and SYSUSERMESSAGES for PRINT and RAISERROR statements.

System procedure	Description
sp_helptext *object-name*	Displays the text of a system procedure, trigger, or view
sp_password *caller_passwd, new_passwd [, userid]*	Adds or changes a password for a user ID

SQL Server catalog procedures

SQL Anywhere implements all catalog procedures with the exception of the sp_column_privileges procedure. The implemented catalog procedures are described in the following table.

The following list describes the SQL Server catalog procedures provided in SQL Anywhere.

Catalog procedure	Description
sp_column_privileges	Unsupported
sp_columns *table-name [, table-owner] [, table-qualifier] [, column-name]*	Returns the data types of the specified column
sp_fkeys *pktable_name [, pktable-owner][, pktable-qualifier] [, fktable-name] [, fktable_owner] [, fktable-qualifier]*	Returns foreign key information about the specified table
sp_pkeys *table-name [, table_owner] [, table_qualifier]*	Returns primary key information for a single table
sp_special_columns *table_name [, table-owner] [, table-qualifier] [, col-type]*	Returns the optimal set of columns that uniquely identify a row in a table
sp_sproc_columns *proc-name [, proc_owner] [, proc-qualifier] [, column-name]*	Returns information about a stored procedure's input and return parameters
sp_stored_procedures [*sp-name] [, sp-owner] [, sp-qualifier]*	Returns information about one or more stored procedures
sp_tables *table-name [, table-owner] [, table-qualifier] [, table-type]*	Returns a list of objects that can appear in a FROM clause

Implicit data type conversion

Both SQL Anywhere and SQL Server carry out implicit data type conversions among some of the data types. Implicit data type conversions are conversions carried out as required by the context, and without any explicit data type conversion function being called. For example, if you compare a CHAR expression and a DATETIME expression, or a SMALLINT expression and an INT expression, or CHAR expressions of different lengths, both SQL Anywhere and SQL Server automatically convert one data type to another in order to carry out the comparison.

This section documents the differences between SQL Anywhere and SQL Server implicit data type conversions.

Differences in string to datetime conversions

There are differences in behavior between SQL Anywhere and SQL Server when converting strings to date and time data types.

If a string containing only a time value (no date) is converted to a date/time data type, SQL Server uses a default date of January 1, 1900, SQL Anywhere uses the current date.

If the milliseconds portion of a time is less than 3 digits, SQL Server interprets the value differently depending on whether it was preceded by a period or a colon. If preceded by a colon, the value means thousandths of a second. If preceded by a period, one digit means tenths, two digits mean hundredths, and three digits mean thousandths. SQL Anywhere interprets the value the same way, regardless of the separator.

For example, SQL Server would convert the values below as shown.

```
12:34:56.7 to 12:34:56.700
12.34.56.78 to 12:34:56.780
12:34:56.789 to 12:34:56.789
12:34:56:7 to 12:34:56.007
12.34.56:78 to 12:34:56.078
12:34:56:789 to 12:34:56.789
```

SQL Anywhere converts the milliseconds value in the manner that SQL Server does for values preceded by a period, in both cases:

```
12:34:56.7 to 12:34:56.700
12.34.56.78 to 12:34:56.780
```

```
12:34:56.789 to 12:34:56.789
12:34:56:7 to 12:34:56.700
12.34.56:78 to 12:34:56.780
12:34:56:789 to 12:34:56.789
```

CHAPTER 30

Transact-SQL Procedure Language

About this chapter

The chapter describes the SQL Anywhere support for writing procedures, triggers, and batches that are portable between SQL Server and SQL Anywhere. Procedures, triggers, and batches all include sets of SQL statements.

The audience for this chapter includes those moving applications from SQL Server to SQL Anywhere, and those developing applications on SQL Anywhere that will also be used on SQL Server.

Before you begin

If you are developing applications for SQL Anywhere only, see the information in the chapter "Using Procedures, Triggers, and Batches" and in the Reference section: you do not need to read this chapter.

Contents

Topic	Page
Transact-SQL procedure language overview	606
Automatic translation of SQL statements	608
Transact-SQL stored procedure overview	610
Transact-SQL trigger overview	611
Transact-SQL batch overview	613
Supported Transact-SQL procedure language statements	614
Returning result sets from Transact-SQL procedures	624
Variable and cursor declarations	625
Error handling in Transact-SQL procedures	627

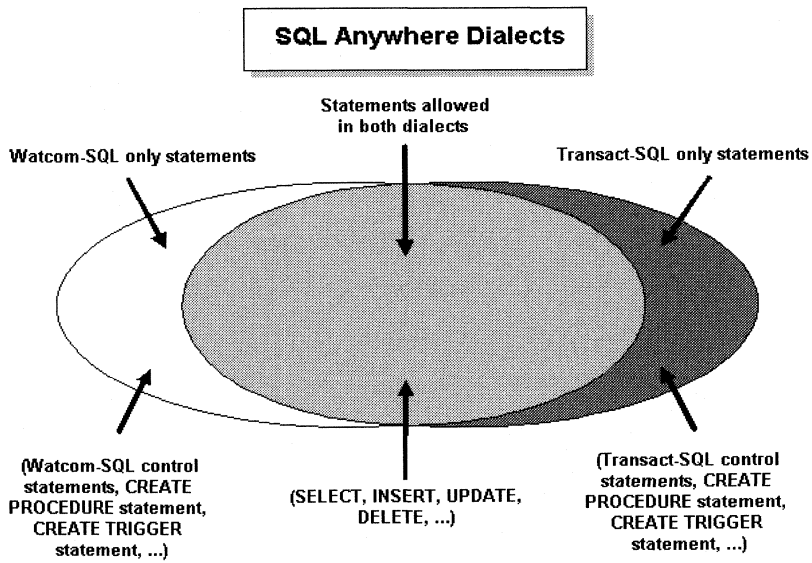
Transact-SQL procedure language overview

The previous chapter described SQL Anywhere support for individual SQL statements. This chapter describes the SQL Anywhere support for sets of SQL statements, whether they appear as stored procedures, triggers, or batches. Also discussed in this chapter are individual statements that are typically found within stored procedures, such as variable assignment statements, as well as control-of-flow statements such as IF and WHILE, and compound statements (bracketed by BEGIN and END). Collectively, these aspects of SQL are called the procedure language.

This chapter describes how the two dialects of SQL coexist in SQL Anywhere, and describes the SQL Anywhere functions for automatic translation of Transact-SQL procedures. The chapter also provides reference for the individual statements that make up the Transact-SQL procedure language.

Mixing Transact-SQL and Watcom-SQL dialects

SQL Anywhere supports two SQL dialects: the native Watcom-SQL and Sybase Transact-SQL. There is a large overlap between the two dialects: *SELECT * FROM employee* is a SQL statement that is part of both Watcom-SQL and Transact-SQL.



Transact-SQL only

There are SQL statements supported by SQL Anywhere that are part of one dialect, but not the other. For example: the following are part of the Transact-SQL dialect only:

- The Transact-SQL control statements IF and WHILE.
- The Transact-SQL EXECUTE statement.
- The Transact-SQL CREATE PROCEDURE and CREATE TRIGGER statements.
- The Transact-SQL BEGIN TRANSACTION statement.
- SQL Statements not separated by semicolons are part of a Transact-SQL procedure or batch.

Watcom-SQL only

The following statements are part of the Watcom-SQL dialect only:

- The Watcom-SQL control statements CASE, IF, LOOP, FOR, and WHILE.
- The Watcom-SQL CALL statement.
- The Watcom-SQL CREATE PROCEDURE, CREATE FUNCTION, and CREATE TRIGGER statements.
- SQL Statements separated by semicolons are part of a Watcom-SQL procedure or batch.

Notes

The two dialects cannot be mixed within a procedure, trigger, or batch. That is:

- You can include Transact-SQL-only statements together with statements that are part of both dialects in a batch, procedure, or trigger.
- You can include Watcom-SQL-only statements together with statements that are part of both dialects in a batch, procedure, or trigger.
- You cannot include Transact-SQL-only statements together with Watcom-SQL-only statements in a batch, procedure, or trigger: a syntax error results.

Automatic translation of SQL statements

In addition to supporting Transact-SQL alternative syntax, SQL Anywhere provides aids for translating statements between the Watcom-SQL and Transact-SQL dialects. The following functions return information about SQL statements and enable automatic translation of SQL statements:

- SQLDialect(statement) returns Watcom-SQL or Transact-SQL.
- WatcomSQL(statement) returns the Watcom-SQL syntax for the statement.
- TransactSQL(statement) returns the Transact-SQL syntax for the statement.

These are functions, and so can be accessed, for example, using a select statement from ISQL. For example, the following statement:

```
select SqlDialect('select * from employee')
```

returns the value WatcomSQL.

Using Sybase Central to translate stored procedures

The Sybase Central database management tool has facilities for creating, viewing, and altering procedures and triggers.

For information on connecting to a database using Sybase Central, and displaying database objects, see the chapter "Managing Databases with Sybase Central" or the Sybase Central online help.

❖ To translate a stored procedure using Sybase Central:

1. Connect to a database using Sybase Central, either as owner of the procedure you wish to change, or as a DBA user.
2. Double-click the Procedures folder for the database to list the stored procedures in the database.
3. Using the right mouse button, click the procedure you wish to translate, and choose the dialect you wish to translate it to from the popup menu: either Watcom-SQL or Transact-SQL.

   The procedure is displayed in the selected dialect. If the selected dialect is not the one in which the procedure is stored, it is translated to that dialect by the database engine. Any untranslated lines are displayed as comments.

4 Rewrite any untranslated lines as needed, and click the Execute Script button to save the translated version to the database. You can also export the text to a file for editing outside Sybase Central.

Transact-SQL stored procedure overview

The Watcom-SQL stored procedure language is based on the ISO/ANSI draft standard, which differs from the Transact-SQL dialect in many ways. Many of the concepts and features are similar, but the syntax is different. SQL Anywhere support for Transact-SQL takes advantage of the similar concepts by providing automatic translation between the Watcom-SQL and Transact-SQL dialects. However, a procedure must be written in one of the two dialects exclusively, not in a mixture.

SQL Anywhere support for Transact-SQL stored procedures

SQL Anywhere support for the following Transact-SQL stored procedure elements is discussed in this chapter:

- Passing parameters
- Returning result sets
- Returning status information
- Providing default values for parameters
- Control statements
- Error handling

Transact-SQL trigger overview

Trigger compatibility requires compatibility of trigger features and of trigger syntax. This section provides an overview of the feature compatibility of Transact-SQL and SQL Anywhere triggers.

SQL Server triggers are executed after the triggering statement has completed: they are statement level, after triggers. SQL Anywhere supports both row level triggers (which execute before or after each row has been modified) and statement level triggers (which execute after the entire statement has been executed).

Row-level triggers are not discussed here, as they are not part of the Transact-SQL compatibility features. For information on row-level triggers in SQL Anywhere, see the chapter "Using Procedures, Triggers, and Batches".

Description of unsupported or different Transact-SQL triggers

The following list describes some features of Transact-SQL triggers that are either not supported or are different in SQL Anywhere:

- **Triggers firing other triggers** Suppose a trigger carries out an action that would, if carried out directly by a user, fire another trigger. SQL Anywhere and SQL Server have slightly different behavior for this case. The default SQL Server behavior is for triggers to fire other triggers up to a configurable nesting level, which has the default value of 16. The nesting level can be controlled by the SQL Server option nested triggers. In SQL Anywhere, triggers fire other triggers without limit unless memory is exhausted.
- **Triggers firing themselves** Suppose a trigger carries out an action that would, if carried out directly by a user, fire the same trigger. SQL Anywhere and SQL Server have different behavior for this case. The default SQL Server behavior is that a trigger does not call itself recursively, but you can turn on the self_recursion option to allow triggers to call themselves recursively. The default SQL Anywhere behavior for Watcom-SQL dialect triggers is for triggers to fire themselves recursively. The default SQL Anywhere behavior for Transact-SQL dialect triggers is for triggers not to fire themselves recursively. This behavior is configurable using the Transact-SQL *SET SELF_RECURSION { ON | OFF}* statement; see "Transact-SQL SET statement" in the chapter "Using Transact-SQL with SQL Anywhere".

- **ROLLBACK statement in triggers** SQL Server permits the ROLLBACK TRANSACTION statement within triggers, which rolls back the entire transaction of which the trigger is a part. SQL Anywhere does not permit ROLLBACK (or ROLLBACK TRANSACTION) statements in triggers. A triggering action and its trigger together form an atomic statement, and SQL Anywhere does not permit ROLLBACKs within atomic statements.
- **ROLLBACK TRIGGER statement** The ROLLBACK TRIGGER statement is not supported by SQL Anywhere.

Transact-SQL batch overview

In Transact-SQL, a batch is a set of SQL statements submitted together and executed as a group, one after the other. A SQL Anywhere command file serves a similar function to a batch file, and the ISQL utility in SQL Anywhere and in SQL Server provide similar capabilities for executing batches interactively.

The control statements used in procedures can also be used in batches. SQL Anywhere supports the use of control statements in batches and the Transact-SQL-like use of non-delimited groups of statements terminated with a GO statement to signify the end of a batch.

Parameters in command files are supported in SQL Anywhere, but not by SQL Server.

Supported Transact-SQL procedure language statements

This section lists the supported Transact-SQL procedure language statements:

- Transact-SQL BEGIN ... END statement (compound statement)
- Transact-SQL CREATE PROCEDURE statement
- Transact-SQL CREATE TRIGGER statement
- Transact-SQL EXECUTE statement
- Transact-SQL GOTO statement
- Transact-SQL IF statement
- Transact-SQL PRINT statement
- Transact-SQL RAISERROR statement
- Transact-SQL RETURN statement
- Transact-SQL WHILE statement

Transact-SQL BEGIN ... END statement

BEGIN and END are used in Transact-SQL to group a set of statements into a single compound statement, so that control statements such as IF ... ELSE , which only affect the performance of a single SQL statement, can affect the performance of the whole group.

In Watcom-SQL, the BEGIN and END keywords are not needed for control statements, as the Watcom-SQL control statements affect blocks of statements, rather than a single statement. The BEGIN and END keywords are required in Watcom-SQL to enclose the body of a stored procedure or trigger. The Watcom-SQL compound statement supports the ATOMIC keyword, that is not part of the Transact-SQL dialect.

In Watcom-SQL, DECLARE statements must always immediately follow a BEGIN keyword, and the cursor or variable declared exists for the duration of the compound statement. Variable declarations in Transact-SQL have different behavior: for more information, see "Variable and cursor declarations" on page 625.

Transact-SQL CREATE PROCEDURE statement

The following subset of the Transact-SQL CREATE PROCEDURE statement is supported by SQL Anywhere.

Syntax

CREATE PROCEDURE [*owner.*]*procedure_name*
... [[( ] *@parameter_name data-type*
... [= *default*] [*output*], ... [ )]]
... [**WITH RECOMPILE**]
... **AS**
... *compound statement*

Notes

- If the Transact-SQL WITH RECOMPILE optional clause is supplied, it is ignored. SQL Anywhere always recompiles procedures the first time they are executed after a database is started, and stores the compiled procedure until the database is stopped.
- Groups of procedures are not supported.

Comparison of Transact-SQL and Watcom-SQL CREATE PROCEDURE statements

The following differences between Transact-SQL and Watcom-SQL statements are listed to help those writing in both dialects.

- **Variable names prefixed by @** The "@" sign denotes a Transact-SQL variable name, while Watcom-SQL variables can be any valid identifier, and the @ prefix is optional.
- **Input and output parameters** Watcom-SQL procedure parameters are specified as IN, OUT, or INOUT, while Transact-SQL procedure parameters are INPUT parameters by default or can be specified as OUTPUT. Those parameters that would be declared as INOUT or as OUT in SQL Anywhere should be declared with OUTPUT in Transact-SQL.
- **Parameter default values** Watcom-SQL procedure parameters are given a default value using the keyword DEFAULT, while Transact-SQL uses an equality sign (=) to provide the default value.
- **Returning result sets** Watcom-SQL uses a RESULT clause to specify returned result sets. In Transact-SQL procedures, the column names or alias names of the first query are returned to the calling environment.

  The following Transact-SQL procedure illustrates how result sets are returned from Transact-SQL stored procedures:

```
CREATE PROCEDURE showdept @deptname varchar(30)
AS
```

```
SELECT employee.emp_lname, employee.emp_fname
FROM department, employee
WHERE department.dept_name = @deptname
AND department.dept_id = employee.dept_id
```

The following is the corresponding Watcom-SQL procedure:

```
CREATE PROCEDURE showdept(in deptname
        varchar(30) )
RESULT ( lastname char(20), firstname char(20))
ON EXCEPTION RESUME
BEGIN
   SELECT employee.emp_lname, employee.emp_fname
   FROM department, employee
   WHERE department.dept_name = deptname
   AND department.dept_id = employee.dept_id
END
```

For more information, see "Returning result sets from Transact-SQL procedures" on page 624.

- **Procedure body** The body of a Transact-SQL procedure is a list of Transact-SQL statements prefixed by the AS keyword. The body of a Watcom-SQL procedure is a compound statement, bracketed by BEGIN and END keywords.

Transact-SQL CREATE TRIGGER statement

The following Transact-SQL CREATE TRIGGER syntax is supported in SQL Anywhere.

Syntax

1. **CREATE TRIGGER** [*owner.*]*trigger_name*
   ... **ON** [*owner.*]*table_name*
   ... **FOR** { **INSERT** , **UPDATE** , **DELETE** }
   ... **AS**
   ... *statement-list*

2. **CREATE TRIGGER** [*owner.*]*trigger_name*
   ... **ON** [*owner.*]*table_name*
   ... **FOR** {**INSERT** , **UPDATE**}
   ... **AS**
   ... [**IF UPDATE** (*column_name*)
   ... [{ **AND** | **OR**} **UPDATE** (*column_name*)] ...]
   ... *statement-list*
   ... [**IF UPDATE** (*column_name*)
   ... [{ **AND** | **OR**} **UPDATE** (*column_name*)] ...]
   ... *statement-list*

Notes

- The rows deleted or inserted are held in two declared temporary tables given the default names **deleted**, and **inserted**, as in SQL Server.

- In SQL Server, trigger names must be unique in the database. In SQL Anywhere, trigger names must be unique for a given owner. For compatible databases, you should make trigger names unique in the database.
- Transact-SQL triggers are executed after the triggering statement.

Transact-SQL EXECUTE statement

The Transact-SQL EXECUTE statement is supported to execute stored procedures, as an alternative to the Watcom-SQL CALL statement.

The EXECUTE syntax is as follows

Syntax

EXECUTE [*@return_status* =] [*creator.*]*procedure_name*
... | [*@parameter-name* =] *expression*, |
| [*@parameter-name* =] *@variable* [*output*] |, ...

Purpose

To invoke a procedure, as an alternative to the Watcom-SQL CALL statement.

Usage

Anywhere

Authorization

Must be the owner of the procedure, have EXECUTE permission for the procedure, or have DBA authority.

Side effects

None.

Description

The EXECUTE statement executes a stored procedure, optionally supplying procedure parameters and retrieving output values and return status information.

The EXECUTE statement is implemented for Transact-SQL compatibility, but can be used in either Transact-SQL or Watcom-SQL batches and procedures.

Example

The following demonstration procedure is used to illustrate the EXECUTE statement.

```
CREATE PROCEDURE p1( @var INTEGER = 54 )
AS
PRINT 'on input @var = %1! ', @var
DECLARE @internal_var integer
SELECT @intvar=123
SELECT @var=@intvar
PRINT 'on exit @var = %1!', @var
```

- The following statement executes the procedure, supplying the input value of 23 for the parameter.

```
EXECUTE p1 23
```

- The following is an alternative way of executing the procedure, which is useful if there are several parameters.

```
EXECUTE p1 @var = 23
```

- The following statement executes the procedure, using the default value for the parameter

```
EXECUTE p1
```

- The following statement executes the procedure, and stores the return value in a variable for checking return status.

```
EXECUTE @status = p1 23
```

Transact-SQL GOTO statement

Syntax

label:

GOTO *label*

Purpose

To branch to a labeled statement.

Usage

Procedures, triggers, and batches only.

Authorization

None.

Side effects

None.

Description

Any statement in a Transact-SQL procedure, trigger, or batch can be labeled. The label name is a valid identifier followed by a colon. In the GOTO statement the colon is not used.

Example

The following Transact-SQL batch prints the message "yes" on the engine window four times:

```
declare @count smallint
select @count = 1
restart:
   print 'yes'
   select @count = @count + 1
   while @count <=4
    goto restart
```

Transact-SQL IF statement

Syntax

IF *expression*
... *statement*
... [**ELSE**
... [**IF** *expression*]

... *statement*]

Purpose

To provide conditional execution of a SQL statement, as an alternative to the Watcom-SQL IF statement.

Usage

Procedures, triggers, and batches only.

Authorization

None.

Side effects

None.

Description

The Transact-SQL IF conditional and the ELSE conditional each control the performance of only a single SQL statement or compound statement (between the keywords BEGIN and END).

Example

```
IF (SELECT max(id) FROM sysobjects) < 100
   RETURN
ELSE
   BEGIN
      PRINT "These are the user-created objects"
      SELECT name, type, id
      FROM sysobjects
      WHERE id < 100
END
```

Notes

- There is no THEN in the Transact-SQL IF statement.
- Transact-SQL has no ELSEIF or END IF keywords.

The following two statement blocks illustrate Transact-SQL and Watcom-SQL compatibility:

```
/* Transact-SQL IF statement */
IF @v1 = 0
   PRINT '0'
ELSE IF @v1 = 1
   PRINT '1'
ELSE
   PRINT 'other'
/* Watcom-SQL IF statement */
IF v1 = 0 THEN
   PRINT '0'
ELSEIF v1 = 1 THEN
   PRINT '1'
ELSE
   PRINT 'other'
END IF
```

Transact-SQL PRINT statement

Syntax

PRINT *format-string* [, *arg-list*]

Purpose To display a message on the message window of the database engine or server.

Usage Procedures, triggers, and batches.

Authorization Must be connected to the database.

Side effects None.

See also MESSAGE Statement

Description The PRINT statement displays an expression on the message window of the database engine or server. In SQL Server the message is displayed on the client window.

The format string can contain placeholders for the arguments in the optional argument list. These placeholders are of the form *%nn!*, where *nn* is an integer between 1 and 20.

Examples

- The following procedure displays a message on the engine message window:

```
CREATE PROCEDURE print_test
AS
PRINT 'Procedure called successfully'
```

  The statement

```
EXECUTE print_test
```

  displays the string Procedure called successfully on the database engine message window.

- The following statement illustrates the use of placeholders in the PRINT statement:

```
DECLARE @var1 INT, @var2 INT
SELECT @var1 = 3, @var2 = 5
PRINT 'Variable 1 = %1!, Variable 2 = %2!',
@var1, @var2
```

Transact-SQL RAISERROR statement

Syntax **RAISERROR** *error-number* [*format-string*] [, *arg-list*]

Purpose To signal an error, and display a message on the message window of the database engine or server.

Usage Anywhere.

Authorization Must be connected to the database.

Side effects None.

See also CREATE TRIGGER Statement

Description The RAISERROR statement allows user-defined errors to be signaled, and displays a message on the engine or server message window. In SQL Server, the message is displayed on the client window.

The *error-number* is a five-digit integer greater than 17000. If *format-string* is not supplied or is empty, the error number is used to locate an error message in the system tables. SQL Server obtains messages 17000-19999 from the SYSMESSAGES table. In SQL Anywhere this table is an empty view, so errors in this range should provide a format string. Messages for error numbers of 20000 or greater are obtained from the SYS.SYSUSERMESSAGES table. The error number is stored in the global variable @@error.

The extended values supported by the SQL Server RAISERROR statement are not supported in SQL Anywhere.

The format string can contain placeholders for the arguments in the optional argument list. These placeholders are of the form *%nn!*, where *nn* is an integer between 1 and 20.

Examples The following statement raises error 99999, which is in the range for user-defined errors, and displays a message.

```
RAISERROR 99999 'Invalid entry for this column:
%1!', @val
```

There is no comma between the *error-number* and the *format-string* parameters. The first item following a comma is interpreted as the first item in the argument list.

The Transact-SQL RETURN statement

Syntax **RETURN** [*integer-expression*]

Purpose To exit a procedure, returning an integer status value to the calling environment.

Usage Procedures and triggers only.

Authorization None.

Side effects None.

Description

The RETURN statement exits a procedure. Statements following the RETURN statement are not executed. If an integer expression is supplied, that is returned to the calling statement as the return value of the procedure. If no integer expression is supplied, a built-in integer status value between 0 and -15 is returned to the calling environment, describing the success of the procedure.

The return value can be assigned to a local variable as in the following batch:

```
DECLARE @status INT
EXECUTE @status = sample_proc
PRINT 'status = %1!', @status
```

For a description of procedure return values, see "Error handling in Transact-SQL procedures" on page 627.

Transact-SQL WHILE statement

Syntax

WHILE *expression*
... *statement*

Purpose

To provide repeated execution of a statement or compound statement.

Usage

Procedures and triggers only.

Authorization

None.

Side effects

None.

Description

The WHILE conditional affects the performance of only a single SQL statement, unless statements are grouped into a compound statement between the keywords BEGIN and END.

The BREAK statement and CONTINUE statement can be used to control execution of the statements in the compound statement. The BREAK statement terminates the loop, and execution resumes after the END keyword marking the end of the loop. The CONTINUE statement causes the WHILE loop to restart, skipping any statements after the CONTINUE.

Example

```
WHILE (SELECT AVG(unit_price) FROM product) < $30
BEGIN
   UPDATE product
   SET unit_price = unit_price + 2
   IF ( SELECT MAX(unit_price) FROM product ) > $50
      BREAK
END
```

The BREAK statement breaks the WHILE loop if the most expensive product has a price above $50. Otherwise the loop continues until the average price is greater than $30.

Returning result sets from Transact-SQL procedures

Watcom-SQL uses a RESULT clause to specify returned result sets. In Transact-SQL procedures, the column names or alias names of the first query are returned to the calling environment.

Example of Transact-SQL procedure

The following Transact-SQL procedure illustrates how result sets are returned from Transact-SQL stored procedures:

```
CREATE PROCEDURE showdept @deptname varchar(30)
AS
   SELECT employee.emp_lname, employee.emp_fname
   FROM department, employee
   WHERE department.dept_name = @deptname
   AND department.dept_id = employee.dept_id
```

Example of Watcom-SQL procedure

The following is the corresponding Watcom-SQL procedure:

```
CREATE PROCEDURE showdept(in deptname varchar(30))
RESULT ( lastname char(20), firstname char(20))
ON EXCEPTION RESUME
BEGIN
   SELECT employee.emp_lname, employee.emp_fname
   FROM department, employee
   WHERE department.dept_name = deptname
   AND department.dept_id = employee.dept_id
END
```

Notes

- Multiple result sets with a different number of columns or incompatible data types cannot be returned from procedures in SQL Anywhere.
- When a RESULT clause is not specified (as is the case with Transact-SQL procedures), SQL Anywhere determines the result set from the first SELECT statement in the procedure. The first SELECT statement is identified without regard for IF statements or other control statements: you cannot have a procedure return one result set under one set of conditions an incompatible result set under other conditions.
- SQL Anywhere does not support the case where a Transact-SQL procedure creates a temporary table and issues a SELECT to return the contents of that table as the result set. In this case, you can convert the procedure to the Watcom-SQL dialect and directly specify the result set. SQL Anywhere also does not support the case where a Transact-SQL procedure creates a local variable and returns its value by including it in a SELECT list. The data type of the variable must be specified in the SELECT statement using a CONVERT or CAST function.

Variable and cursor declarations

The syntax of the DECLARE CURSOR and DECLARE statements is very similar in Transact-SQL and Watcom-SQL. This section describes the Transact-SQL DECLARE CURSOR statement.

Transact-SQL DECLARE CURSOR statement

The following subset of the Transact-SQL DECLARE CURSOR statement is supported by SQL Anywhere.

Syntax

DECLARE *cursor-name* **CURSOR FOR** *select_statement*
... [**FOR** { **READ ONLY** | **UPDATE** }]

SQL Server supports cursors opened for update of a list of columns from the tables specified in the select_statement. This is not supported in SQL Anywhere.

Notes

- In the Watcom-SQL dialect, a DECLARE CURSOR statement in a procedure, trigger, or batch must immediately follow the BEGIN keyword. In the Transact-SQL dialect, there is no such restriction.
- In SQL Server,when a cursor is declared in a procedure, trigger, or batch, it exists for the duration of the procedure, trigger, or batch. In SQL Anywhere, if a cursor is declared inside a compound statement, it exists only for the duration of that compound statement (whether it is declared in a Watcom-SQL or Transact-SQL compound statement).
- Watcom-SQL also supports the CREATE VARIABLE statement, which creates a variable that exists for the duration of a connection or until it is explicitly dropped using the DROP VARIABLE statement.
- You should be careful about using the DECLARE statement in ISQL to declare local variables. If you submit a set of statements separated by semicolons, ISQL parses them and sends them individually to the engine rather than as a batch. Each statement becomes its own "batch", and the declared variable is not accessible beyond the end of its own batch: that is, it is not accessible anywhere. You can prevent ISQL from breaking up the statements by enclosing them in a BEGIN and an END. Alternatively, you could omit the semicolons, and the batch will be sent as a Transact-SQL batch to the engine.

Transact-SQL DECLARE statement

The syntax of the Transact-SQL DECLARE statement for a single variable is the same as that for Watcom-SQL, except that the variable name must be preceded by an @ sign in Transact-SQL, while this is optional in the Watcom-SQL dialect.

Syntax

DECLARE *@variable_name data-type*

SQL Server allows several variables to be declared in a single statement. Only a single variable can be declared in one statement in SQL Anywhere.

Notes

- In the Watcom-SQL dialect, a DECLARE statement in a procedure, trigger, or batch must immediately follow the BEGIN keyword. In the Transact-SQL dialect, there is no such restriction.
- In SQL Server,when a variable is declared in a procedure or trigger, it exists for the duration of the procedure or trigger. In SQL Anywhere, if a variable is declared inside a compound statement, it exists only for the duration of that compound statement (whether it is declared in a Watcom-SQL or Transact-SQL compound statement).

Assigning values to variables in Transact-SQL

Watcom-SQL uses the SET statement to assign values to variables in a procedure. In Transact-SQL, values are assigned using the SELECT statement with an empty table-list. The following simple procedure illustrates how the Transact-SQL syntax works:

```
CREATE PROCEDURE multiply
                @mult1 int,
                @mult2 int,
                @result int output
AS
SELECT @result = @mult1 * @mult2
```

This procedure can be called as follows:

```
CREATE VARIABLE @product int ;

EXECUTE multiply 5, 6, @product OUTPUT;
```

The variable @product has a value of 30 after the procedure is executed.

For more information on using the SELECT statement to assign variables, see "Transact-SQL SELECT statement" in the chapter "Using Transact-SQL with SQL Anywhere". For more information on using the SET statement to assign variables, see "SET statement" in the chapter "Watcom-SQL Statements".

Error handling in Transact-SQL procedures

Default procedure error handling is different in the Watcom-SQL and Transact-SQL dialects. By default, Watcom-SQL dialect procedures exit when an error is encountered, returning a SQLSTATE and SQLCODE value to the calling environment. Explicit error handling can be built in to Watcom-SQL stored procedures using the EXCEPTION statement, or the procedure can be instructed by the ON EXCEPTION RESUME statement to continue execution at the next statement when an error is encountered.

When an error is encountered in a Transact-SQL dialect procedure, execution continues at the following statement. The global variable @@error holds the error status of the most recently executed statement. You can check this variable following a statement to force return from a procedure. For example, the following statement causes an exit if an error occurs.

```
IF @@error != 0 RETURN
```

When the procedure completes execution, a return value indicates the success or otherwise of the procedure. This return status is an integer, and can be accessed as follows:

```
DECLARE @status INT
EXECUTE @status = proc_sample
IF @status = 0
   PRINT 'procedure succeeded'
ELSE
   PRINT 'procedure failed'
```

The following table describes the built-in procedure return values and their meanings:

Value	Meaning
0	Procedure executed without error
-1	Missing object
-2	Data type error
-3	Process was chosen as deadlock victim
-4	Permission error
-5	Syntax error
-6	Miscellaneous user error
-7	Resource error, such as out of space
-8	Non-fatal internal problem

Value	Meaning
-9	System limit was reached
-10	Fatal internal inconsistency
-11	Fatal internal inconsistency
-12	Table or index is corrupt
-13	Database is corrupt
-14	Hardware error

The RETURN statement can be used to return integers other than these, with their own user-defined meanings.

Using the RAISERROR statement in procedures

The RAISERROR statement is a Transact-SQL statement for generating user-defined errors. It has a similar function to the Watcom-SQL SIGNAL statement. For a description of the RAISERROR statement, see "Transact-SQL RAISERROR statement" on page 620.

By itself the RAISERROR statement does not cause an exit from the procedure, but it can be combined with a RETURN statement or a test of the @@error global variable to control execution following a user-defined error.

Transact-SQL-like error handling in the Watcom-SQL dialect

You can make a Watcom-SQL dialect procedure handle errors in a Transact-SQL-like manner by supplying the ON EXCEPTION RESUME clause to the CREATE PROCEDURE statement\:

```
CREATE PROCEDURE sample_proc()
ON EXCEPTION RESUME
BEGIN
   ...
END
```

Explicit exception handling code is not executed if an ON EXCEPTION RESUME clause is present.

CHAPTER 31

Using the Open Server Gateway

About this chapter

This chapter describes the Open Server Gateway for SQL Anywhere: an application that enables SQL Server client applications to work with SQL Anywhere.

The chapter describes the uses of the Open Server Gateway, outlines its architecture, and describes how to configure and manage it. The chapter also provides a description of incompatibilities with SQL Server.

The chapter is intended for developers implementing client applications that will use the Open Server Gateway to manipulate SQL Anywhere databases, and for users of Sybase Open Client applications who wish to use them with SQL Anywhere also.

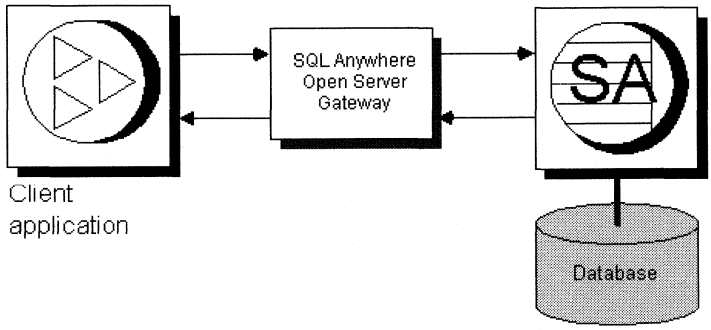
In order to run the Open Server Gateway, you must have Sybase Open Client and Open Server kits.

Contents

Topic	Page
Open Server Gateway overview	630
Open Server Gateway architecture	632
Data type mappings	634
Setting up the Open Server Gateway	642
Events handled by Open Server Gateway	646
Using SQL Anywhere with OmniCONNECT	649

Open Server Gateway overview

SQL Server client applications communicate with SQL Server using the Open Client libraries provided by Sybase. The client applications can instead work with SQL Anywhere using Open Server Gateway. The purpose of Open Server Gateway is to enable client applications to communicate with both SQL Server and SQL Anywhere. The Open Server Gateway is one part of the SQL Server compatibility offered by SQL Anywhere.



Replication Server support

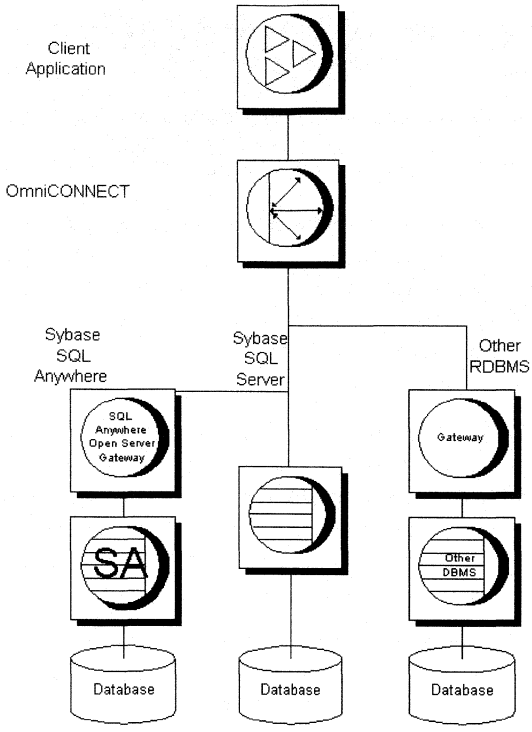
The Open Server Gateway is part of SQL Anywhere support for Sybase Replication Server: Replication Server connects to SQL Anywhere through the Open Server Gateway, enabling SQL Anywhere databases to act as replicate sites in Replication Server installations. Also, Sybase **isql** can send SQL statements to SQL Anywhere through Open Server Gateway.

For SQL Anywhere databases to act as a primary site in a Replication Server installation, you must also purchase the Replication Agent for Sybase SQL Anywhere, also called a Log Transfer Manager. This is available from Sybase as a separate product.

OmniCONNECT support

Sybase OmniCONNECT provides a unified view of disparate data within an organization, allowing users to access multiple data sources without having to know what the data looks like or where it is located. In addition, OmniCONNECT performs heterogeneous joins of data across the enterprise, enabling cross-platform table joins of targets such as DB2, Sybase SQL Server, Oracle, and VSAM.

Using the Open Server Gateway, Sybase SQL Anywhere can act as a data source for OnmiCONNECT. The archiecture of OmniCONNECT is illustrated in the figure.


Client
Application
OmniCONNECT
Sybase
SQL
Anywhere
Sybase
SQL
Server
Other
RDBMS
SQL
Anywhere
Open Server
Gateway
SA
Gateway
Other
DBMS
Database
Database
Database

Open Server Gateway architecture

The Open Server Gateway for SQL Anywhere allows database client applications built using the Sybase Open Client library to communicate with SQL Anywhere database servers.

The Open Server Gateway acts as a server application to the client application, responding to its requests. It also acts as a client application for SQL Anywhere database servers or engines, passing requests from client applications on to the SQL Anywhere server and receiving responses. An application that acts as both client and server is called a **gateway** application.

Event handling interface

When a client application sends a request to Open Server Gateway, it triggers an **event**, and the Open Server Gateway executes a routine to handle that event. Open Server Gateway functionality is described in terms of these events: the events to which it can respond, and the ways in which it responds to each event. For a description of Open Server Gateway event handling, see "Events handled by Open Server Gateway" on page 646.

Open Server Gateway returns results to a client application as messages, rows of data, result parameters, or status values. A single client request can have more than one set of results.

The Open Server Gateway is designed to allow client applications to work with both SQL Server and with SQL Anywhere databases. It handles events that SQL Server client applications generate, translates them where necessary so that the SQL Anywhere server can respond to them, and passes them on to the SQL Anywhere server. On receiving results from the server, Open Server Gateway translates them where necessary into a form that the client application can understand, and passes them back.

Examples

Events include the following:

- **SRV_LANGUAGE** A client has sent a language request, such as a SQL statement.
- **SRV_OPTION** A client has sent an option command.
- **SRV_CONNECT** A Client-Library client has called **ct_connect**. These events are examples of **standard events**. For a description of supported and unsupported standard events, see "Events handled by Open Server Gateway" on page 646.

What you need to use the Open Server Gateway

The Open Server Gateway communicates with client applications using network libraries supplied as part of the Sybase SQL Server products. In order to run Open Server Gateway, you must elect to install the Open Server components when installing SQL Anywhere.

Ensure your database is compatible

Any Sybase SQL Anywhere database can be accessed using the Open Server Gateway. However, you should ensure that your database is created for maximum compatibility with SQL Server. For more information about making your SQL Anywhere database as compatible as possible with SQL Server, see "Configuring SQL Anywhere for Transact-SQL compatibility" in the chapter "Using Transact-SQL with SQL Anywhere".

Maximum compatibility not required for Replication Server
If you are using your SQL Anywhere database strictly for Replication Server, it is not critical to deviate from SQL Anywhere default settings. If you want to use other Open Client applications, such as OmniCONNECT, then it is important to use the SQL Server compatible settings.

The Open Server Gateway does take some actions itself on connecting to help ensure SQL Server-compatible behavior. These actions are described in "Open Server Gateway actions on connecting", on page 644.

Data type mappings

The range of possible Open Client data types differs from the range of possible SQL Anywhere data types. For this reason, Open Server Gateway must map some data types between what data types are used by Open Client applications and expected in a database they connected with, and what data types are available in SQL Anywhere.

Data types understood by Open Client applications and those available in SQL Anywhere fall into the following categories:

- Data types that can exist in both the Open Client and SQL Anywhere, allowing values to be exchanged without problems.
- Data types that can exist in both Open Client applications and SQL Anywhere but which have different ranges of possible values.
- Data types that can contain the same values in both Open Client applications and SQL Anywhere but are named or described differently.
- Open Client data types not supported by SQL Anywhere.

Specifically which data types fit into these categories has been modified with release 5.502 of SQL Anywhere. The following sections detail the use of Open Client application data types with release 5.501, and release 5.502 of SQL Anywhere.

Problem-free Open Client data types

The following table lists Open Client data types that are supported and considered problem free for use with SQL Anywhere. Use these data types to avoid any data type problems in an Open Client application connecting to a SQL Anywhere database through Open Server Gateway.

Open Client applications connecting to SQL Anywhere 5.501 and earlier releases	Open Client applications connecting to SQL Anywhere 5.502 and later releases
tinyint	tinyint
smallint	smallint
int	int
real	real
char	char
varchar	varchar

Open Client applications connecting to SQL Anywhere 5.501 and earlier releases	Open Client applications connecting to SQL Anywhere 5.502 and later releases
datetime	datetime
text	text
image	image
decimal	numeric*
binary	varbinary
	money
	smallmoney

* The SQL Anywhere data types numeric (20,4) and numeric (10,4) data types have a different behavior from Numeric data types of other precisions. The Open Server Gateway maps these data types to the Open Client data types Money and Smallmoney. Form more information on data type mapping from SQL Anywhere to Open Client, see "Mapping data from SQL Anywhere to Open Client", page 637.

Replication Server: use problem-free data types

Replication Server's SUBCMP (or RSSUBCMP) utility insists that data types match precisely. If you plan to use this utility between a SQL Server and a SQL Anywhere database, use only data types listed in the column "Open Client applications connecting to SQL Anywhere 5.502 and later releases" of the above table.

New behavior in 5.502
The data types decimal and binary were previously considered to be problem free in release 5.501. The behavior of release 5.502 has been altered in that these data types are now considered to have description problems. Open Client applications using decimal and binary data types and connecting to SQL Anywhere through Open Server Gateway may have problems when upgrading to release 5.502.

For more information about data types with description problems, see "Open Client data types with description limitations", on page 640.

Release 5.501 compatibility switch

To obtain the Open Server Gateway data type mappings available in release 5.501 while using release 5.502, use the -c 5501 compatibility command-line switch.

For more information on Open Server Gateway command-line switches including the -c 5501 compatibility switch, see "The Open Server Gateway" in the chapter "SQL Anywhere Components"

Data type mappings between systems

With the exception of the data types listed in "Problem-free Open Client data types", many Open Client data types are mapped by Open Server Gateway to SQL Anywhere data types which have different names or underlying structures.

Return data type may differ from original

It is possible that some values passed by Open Client to SQL Anywhere as one data type, may be converted by Open Server Gateway into a SQL Anywhere-specific data type. When it is returned or fetched by the Open Client application it may be mapped by the Open Server Gateway to yet another data type, different from the value's original data type.

For this reason, the following table illustrates the data type mapping from Open Client applications to SQL Anywhere. A second table illustrates the data type mapping from SQL Anywhere (releases 5.501 and 5.502) to Open Client applications.

All data types mappings are listed, including those for which there are no issues (for example, data types that map directly from Open Client applications to SQL Anywhere and from SQL Anywhere to Open Client applications).

Mapping data from Open Client to SQL Anywhere

When a value in an Open Client application is passed to SQL Anywhere, it may be converted from an Open Client data type to a SQL Anywhere data type, depending on its original data type. The following table illustrates the mapping of data types performed by Open Server Gateway when an Open Client application passes a value to SQL Anywhere.

The mapping illustrated in the following table applies to all SQL Anywhere releases, except where noted.

Open Client Data type	SQL Anywhere data type
tinyint	tinyint
smallint	smallint
int	int
real (4 byte)	real

Open Client Data type	SQL Anywhere data type
float (8 byte) in host variables	double
float (8 byte) in SQL statements	real
char	char
varchar	varchar
binary	binary/varbinary**
varbinary	binary/varbinary**
bit	tinyint
numeric	decimal
decimal	decimal
text	long varchar
image	long binary
datetime	timestamp
smalldatetime	timestamp
timestamp (SQL statements only)	timestamp
money	decimal (20,4)
smallmoney	decimal (10,4)
sysname (SQL statements only)	varchar(30)
long char (host variables only)*	long varchar (max. 32767 bytes)
long binary (host variables only)*	long binary (max. 32767 bytes)
ALL OTHERS	ERROR

* Applies to release 5.502 and greater of SQL Anywhere only.

** SQL Anywhere Binary data types have an underlying structure that is identical to what is commonly understood to be a Varbinary data type. For more information search the online help for "BINARY data type".

Mapping data from SQL Anywhere to Open Client

The following table illustrates the mapping from SQL Anywhere data types to Open Client applications depending on the release of SQL Anywhere being used.

SQL Anywhere data type	Open client connecting to SQL Anywhere 5.501	Open client connecting to SQL Anywhere 5.502
tinyint	tinyint	tinyint
smallint	smallint	smallint
int	int	int
real	real (4 byte)	real (4 byte)
double	float (8 byte)	float (8 byte)
string	char	char
char	char	char
varchar	varchar	varchar
date	datetime	datetime
time	datetime	datetime
timestamp	datetime	datetime
long varchar	text	text
long binary	image	image
timestamp struct	ERROR	ERROR
variable	ERROR	ERROR
binary/varbinary	binary	varbinary
decimal*	decimal	numeric
decimal (20,4)	decimal (20,4)	money
decimal (10,4)	decimal (10,4)	smallmoney

* The data mapping of Decimal data types applies to all precisions, except Decimal (20,4) and Decimal (20,4) as noted in the last two rows of the above table.

Data types with value range limitations

The following table lists Open Server application data types that can be mapped to SQL Anywhere data types, but with some restriction in the range of possible values.

In most of these cases, the Open Client data type is mapped to a SQL Anywhere data type that has a greater range of possible values. As a result, it is possible to pass a value to SQL Anywhere that will be accepted and stored in a database, but one that is too large to be fetched by an Open Client application.

The tables in the section "Data type mappings between systems", on page 636, outline which Open Client data types are matched to SQL Anywhere data types and visa versa.

The data types listed in the following table have the potential to cause errors because of the range of possible values.

Example

For example, the Open Client Money and smallmoney data types do not span the entire numeric range of their underlying SQL Anywhere implementations. Therefore it is possible to have a value in a SQL Anywhere column which exceeds the boundaries of the Open Client data type money. When the client fetches any such offending values via the Open Server Gateway, an error will occur ("Open Server Gateway Error 22: Invalid data format. Possible overflow or underflow").

The implementation in SQL Anywhere of Open Client's timestamp data type, when such a value is passed in SQL Anywhere (the value is mapped to the SQL Anywhere datetime data type), is different from that of SQL Server's. The default value is NULL in SQL Anywhere , and no guarantees are made to ensure its uniqueness. By contrast, SQL Server ensures that the value is monotonically increasing in value, and so is unique.

By contrast, the SQL Anywhere timestamp data type contains year, month, day, hour, minute, second and fraction of second information, and the datetime data type has a greater range of possible values than the Open Client data types that are mapped to it by Open Server Gateway.

Data type	Implementation	Lower range	Upper range
money	Open Client applications	-922,337,203,685,477.5808	922,337,203,685,477.5807
decimal (20,4)	SQL Anywhere	-999,999,999,999,999.9999	999,999,999,999,999.9999
smallmoney	Open Client applications	-214,748.3648	214,748.3647
decimal (10,4)	SQL Anywhere	-999,999.9999	999,999.9999
timestamp	Open Client applications	N/A	N/A
datetime	Open Client applications	Jan 1, 1753	Dec 31, 9999

Data type	Implementation	Lower range	Upper range
smalldatetime	Open Client applications	Jan 1, 1900	June 6, 2079
datetime	SQL Anywhere	Jan 1, 0001	Dec 31, 9999
bit	Open Client applications	0	1
tinyint	SQL Anywhere	0	255

Other restrictions

SQL Anywhere defines sysname as varchar (30) and does not allow the Open Server client application to specify the length of the data field.

Open Client data types with description limitations

Except for the issues raised in "Data types with value range limitations", on page 638, the data types listed in the following table will not cause any problems so long as the Open Client application can handle having the data result returned in a different data type format. If these applications force the data to be converted to the original, expected data type, no problems will occur. However, most Open Client applications tend to bind to the data type described by the Open Server.

Open Client data type	SQL Anywhere 5.501 and previous releases	SQL Anywhere 5.502 and beyond
bit	tinyint	tinyint
sysname	varchar (30)	varchar (30)
timestamp	datetime	datetime
smalldatetime	datetime	datetime
decimal*	decimal	numeric
decimal (20,4)	decimal (20,4)	money
decimal (10,4)	decimal (10,4)	smallmoney
numeric (20,4)	decimal (20,4)	money
numeric (10,4)	decimal (10,4)	smallmoney
binary	binary	varbinary

* The data mapping of Decimal data types applies to all precisions, except Decimal (20,4) and Decimal (20,4) as noted in the above table.

Limited compatibility Open Client data types

The following tables lists Open Client data types for which there is no, or limited, support in SQL Anywhere.

Open Client application data type	Problem
float*	When used in SQL statements, 'float' is interpreted as 'real'
long	Not supported
sensitivity	Not supported
boundary	Not supported
void	Not supported

* FLOAT is supported sufficiently for use with Replication Server.

Setting up the Open Server Gateway

Whenever you run Open Server Gateway for SQL Anywhere, you provide a **server name** on the command line that identifies the Open Server Gateway to client applications. This server name must be recognized by client applications in order for them to communicate with the Open Server Gateway.

Typically, you may provide an Open Server Gateway name that is the same as the name of the SQL Anywhere server to which it is connected. You need to add each Open Server Gateway name to your list of Open Servers. You can do this from the Sybase SQLEDIT utility for managing network connection information for Open Servers.

This section describes the steps in using SQLEDIT to add an Open Server Gateway to your list of Open Servers. It is not complete SQLEDIT documentation. The SQLEDIT utility is provided with Sybase Net-Library.

SQLEDIT equivalent for NetWare
For NetWare, the sybinit utility provides the same services as SQLEDIT does on other platforms.

Required information

For each Open Server, you must provide the following information:

- **Server name** The name of the Open Server, which you set when starting Open Server Gateway.

  Use Open Server names of eight characters of less
  Server names of more than eight characters can cause problems on some file systems. It is recommended that you use server names of eight characters or less.

- **Driver** The Sybase Net-Library driver DLL to be used for the connection. A different driver is used for each network protocol.
- **Service type** The type of connection for the server. A **query** service holds connection information used by a client to log in to the server. A **master** service holds connection information used by a server to listen for connection requests from clients.
- **Connection information** Network address-related information for the specified server.

Add an Open Server

To add an Open Server, choose Add Server from the SQLEDIT Edit menu, and type a name for your Open Server in the space provided. Open Server names are always entered as upper case, but are case insensitive.

You may wish to use the same name for your Open Server Gateway as for the SQL Anywhere database server it is connected to. For example, if you are running a database server on the sample database, with the default name of SADEMO, you may wish to name your Open Server SADEMO.

The default server name for the Open Server Gateway is SQLAny.

Add a QUERY service

You need to add a QUERY service for the Open Server. To add a service, choose Add Service from the Edit menu. In Connection Information, provide the following information:

- The network driver used for the connection; for example, TCP/IP Sockets.
- The connection's address-related information.

Add a MASTER service

You need to add a MASTER service for the server. The entries you provide for the Network Driver and Connection Information should be the same as for the QUERY service.

A MASTER service is required when the program is running on the current machine. If the program is running on some other computer, you need only a QUERY service.

Starting the Open Server Gateway

Open Server Gateway is a client application for SQL Anywhere database engines or servers. With a SQL Anywhere database server running, you can run the Open Server Gateway so that client applications using Sybase Open Client architecture can communicate with a SQL Anywhere database server.

The executable name of the Open Server Gateway is DBOS50.EXE. For usage information about the Open Server Gateway, type

```
dbos50 -?
```

For a full description of the Open Server Gateway command-line switches, see "The Open Server Gateway" in the chapter "SQL Anywhere Components".

To start the Open Server Gateway for NT or OS/2, using the default command-line switches, type the following from a command line:

```
start DBOS50 server_name
```

where server_name is the name of the Open Server, as specified in the interfaces file. For an Open Server named sa_os, the command line would be

```
start DBOS50 sa_os
```

To load the NetWare Open Server Gateway, using the default command-line switches, type the following from a NetWare command line:

```
load DBOS50.nlm server_name
```

By default, the Open Server Gateway maintains a log file in the current directory, with name *server*.WOS where *server* is the name of the Open Server.

Use verbose mode for debugging only
The Open Server Gateway has a -v command-line switch to run in a **verbose mode**. In verbose mode, debugging messages are output, including the result sets of any queries executed and so on. Although the verbose mode is useful for debugging purposes, it has a significant effect on performance and should not be used in production environments.

Confirm the Open Server Gateway is configured properly

You can confirm that the Open Server Gateway is running properly by connecting to the database using the Sybase **isql** utility. Sybase isql connects to SQL Anywhere databases using the Open Server Gateway.

To start isql running on the SQL Anywhere server, type

start isql -U uid -P password -S server_name

where *uid* is a user ID with DBA permissions, and *server_name* is the name of your primary site server.

Potential PATH conflict
If the SQL Anywhere installation directory precedes your Sybase installation directory in your path, you will need to type the full path to avoid starting the SQL Anywhere ISQL instead.

To start isql running on the sademo Open Server Gateway, type

start isql -U dba -P sql -S sademo

Open Server Gateway actions on connecting

When Open Server Gateway connects to SQL Anywhere, it automatically sets relevant database options to values that are compatible with SQL Server default behavior. These options are set temporarily, for the duration of the Open Server Gateway connection only. They can be overridden by the client application at any time.

The database options that are set by Open Server Gateway on connection are as follows:

Option	Set to
ALLOW_NULLS_BY_DEFAULT	OFF
ANSI_BLANKS	ON
ANSI_INTEGER_OVERFLOW	ON
AUTOMATIC_TIMESTAMP	ON
CHAINED	OFF
QUOTED_IDENTIFIER	OFF
TSQL_HEX_CONSTANT	ON
TSQL_VARIABLES	ON

For more information

For more information about available SQL Anywhere database options, see "SET OPTION statement" in the chapter "Watcom-SQL Statements".

For more information about compatibility of SQL Anywhere and SQL Server options, see "Transact-SQL SET statement" in the chapter "Using Transact-SQL with SQL Anywhere" and "Setting options for Transact-SQL compatibility" in the chapter "Using Transact-SQL with SQL Anywhere".

For more information about making your SQL Anywhere database as compatible as possible with SQL Server, see "Creating a Transact-SQL-compatible database" in the chapter "Using Transact-SQL with SQL Anywhere".

Events handled by Open Server Gateway

Sybase Client Library applications send requests to Open Server Gateway. Each request triggers an **event**, and Open Server Gateway executes a routine to handle that event. The routine that handles an event is called an **event handler**.

Open Server Gateway functionality is described in terms of these events: the events to which it can respond, and the ways in which it responds to those events.

Events can be of two kinds:

- **Standard events.** Events defined internally in Open Server.
- **Programmer-defined events.** Events defined by the Open Server Gateway application. Open Server Gateway does not support any programmer-defined events.

Standard event handling in Open Server Gateway

The following list describes each Open Server standard event, and the Open Server Gateway handling of that event.

- **SRV_ATTENTION event** This usually occurs when a client calls **ct_cancel** to stop results processing prematurely. Open Server Gateway checks the status of SRV_ATTENTION between fetches, and stops processing if the event has been received.
- **SRV_BULK event** A client has inserted a binary large object or has issued a bulk copy request. Inserting binary large opjects is supported, but bulk copies are not supported by Open Server Gateway.
- **SRV_CONNECT event** A Client-Library client has called **ct_connect**. Connections to SQL Server are made to the server, and the USE SQL statement sets the current database. Connections to SQL Anywhere are made to an individual database. The -d and -e command-line switches for Open Server Gateway determine the server and database to which connections will be made when a connect event occurs. For more information on the command-line switches, see "The Open Server Gateway" in the chapter "SQL Anywhere Components".
- **SRV_CURSOR event** A client has sent a cursor request. This event is supported by Open Server Gateway.

- **SRV_DISCONNECT event** A request to disconnect a client connection has been made. This event is supported by Open Server Gateway.
- **SRV_DYNAMIC event** A client has sent a Dynamic SQL request. Dynamic SQL allows a client application to execute SQL statements containing variables whose values are determined at run-time. All the operation types available for the SRV_DYNAMIC event are supported except for the DESCRIBE_INPUT type. The SQL Anywhere Open Server Gateway does not support the CS_PROTO_DYNPROC capability.
- **SRV_LANGUAGE event** A client has sent a language request, such as a SQL statement. The SQL statement is passed on to SQL Anywhere. The success of failure of the event depends on whether the SQL statement is supported by SQL Anywhere.

  For information about Transact-SQL compatibility in SQL Anywhere, see the chapter "Using Transact-SQL with SQL Anywhere".
- **SRV_MSG event** A client has sent a message. The message ID and parameters are read, but no action is performed. The success return code is returned to the client application.
- **SRV_OPTION event** A client has sent an option command. Options that are supported by SQL Anywhere are passed on to the database server.
- **SRV_RPC event** A client has issued a remote procedure call. The procedure is specified as follows:

  ```
  owner-name.database-name.procedure-name
  (parameter-list)
  ```

  Open Server Gateway checks to see if the current database is the correct one, and if so executes the procedure. If the current database is not the correct one, a failure is returned.
- **SRV_START event** A call to **srv_run** triggers a SRV_START event. This event is not generated by client applications.
- **SRV_STOP event** A request to stop Open Server Gateway has been made. This event is not generated by client applications. The **dbosstop** utility is provided for shutting down the Open Server.
- **SRV_URGDISCONNECT event** This event is only triggered by an Open Server application calling **srv_event**. It is not supported by Open Server Gateway.

Known limitations of the Open Server Gateway

Using the Open Server Gateway, you can use a SQL Anywhere database in much the same way as you would a SQL Server. There are some limitations, including the following:

- **Commit Service** The Open Server Gateway does not support the SQL Server Commit Service.
- **Prepare Transaction statement** The Open Server Gateway does not support the SQL Server Prepare Transaction statement used by DB-Library in a two-phase commit application to see if a server is prepared to commit a transaction.
- **Data types** Not all SQL Anywhere and SQL Server data types are supported in the Open Server Gateway. For a description of data type compatibility, see "Data type mappings", on page 634.
- **Capabilities** A client/server connection's **capabilities** determine the types of client requests and server responses permitted for that connection. The following capabilities are not supported:
  - CS_REG_NOTIF
  - CS_CSR_ABS
  - CS_CSR_FIRST
  - CS_CSR_LAST
  - CS_CSR_PREV
  - CS_CSR_REL
  - CS_DATA_BOUNDARY
  - CS_DATA_SENSITIVITY
  - CS_PROTO_DYNPROC
  - CS_REQ_BCP

  For more information on capabilities, see the *Open Server Server-Library C Reference Manual.*

Using SQL Anywhere with OmniCONNECT

This section describes how to start the Open Server Gateway and SQL Anywhere to work with Sybase OmniCONNECT heterogeneous data access software. It does not provide any documentation on OmniCONNECT itself. See your OmniCONNECT documentation for details of how to configure a new data server and other aspects of managing data servers.

The Open Server Gateway allows Sybase SQL Anywhere data sources to appear for many purposes to client applications as if they were Sybase SQL Server data sources. Using the Open Server Gateway, you can incorporate a SQL Anywhere database into your heterogeneous data sources managed by OmniCONNECT. A SQL Anywhere database together with the Open Server Gateway can be treated, subject to limitations described below, as a SQL Server database.

Setting up SQL Anywhere as an OmniCONNECT remote server

This section describes how to set up the Open Server Gateway and SQL Anywhere to work with OmniCONNECT.

❖ **To set up SQL Anywhere as an OmniCONNECT remote server:**

1. Start the SQL Anywhere database engine or server running on the database you wish to use. The database must be created with SQL Anywhere Release 5.5.01 or later.
2. Configure and start the Open Server Gateway running against the database. For information on how to do this, see "Starting the Open Server Gateway", on page 643.
3. Run the OmniCONNECT system procedure sp_addserver to add the new remote server. The SQL Anywhere database should be added with server class SQL_server. OmniCONNECT detects the new server as a post-System 10 SQL Server.

   For example, if you started the Open Server Gateway on the sample database as follows:

   ```
   dbos50 -d sademo SQLAny_1
   ```

   then the following OmniCONNECT command line would add the new server:

   ```
   exec sp_addserver SQLAny_1 sql_server, SQLAny_1
   ```

4 If there is no user on the SQL Anywhere database with identical user ID and password to the OmniCONNECT user, you must add an external login. For example:

```
exec sp_addexternallogin SQLAny_1 sa, dba, sql
```

5 Define the objects as for other OmniCONNECT data sources. For example, to define a table named employee in the SQL Anywhere sample database on the Open Server Gateway server SQLAny_1 and owned by User3, you use the following OmniCONNECT command (all on one line):

```
exec sp_addobjectdef employee,
"SQLAny_1.sademo.user3.employee", "table"
```

Limitations of using SQL Anywhere with OmniCONNECT

The following are known limitations when using SQL Anywhere with OmniCONNECT:

- **Bulk inserts** Bulk inserts are not supported.
- **Object case sensitivity** Database identifiers (column names, table names, and so on) are treated as case insensitive. String constants are also case-insensitive by default, but for databases created with case-sensitive string comparisons (indicated by using the DBINIT -c command-line switch, or the appropriate SQL Central field), these are case sensitive.
- **defgen** The OmniCONNECT defgen utility for extracting table definitions is not supported by SQL Anywhere.
- **Unsupported data types** The NCHAR, NVARCHAR, and TIMESTAMP data types are not supported.
- **Open Server Gateway limitations** The limitations listed in "Known limitations of the Open Server Gateway", on page 648, also apply.

PART FIVE

The SQL Anywhere Programming Interfaces

SQL Anywhere supports four different programming interfaces. This part describes each interface in turn.

CHAPTER 32

Programming Interfaces

This section of the manual describes the various interfaces to SQL Anywhere. They are presented in order as follows:

1 Embedded SQL for the C programming language

2 ODBC call level interface

3 SQL Anywhere Dynamic Data Exchange Server

4 WSQL HLI Dynamic Link Library

Embedded SQL

Embedded SQL is a system in which SQL commands are embedded right in a C or C++ source file. A preprocessor is run which translates these statements into calls to a runtime library. Embedded SQL is an ANSI and IBM standard. It is portable to other databases and other environments. Embedded SQL is the lowest level interface to the SQL Anywhere database engine and is functionally equivalent in all operating environments. It provides all functionality that is available in the product. Embedded SQL is quite easy to work with, although it takes a little getting used to the idea of Embedded SQL statements (rather than function calls) in C code.

ODBC

ODBC (Open Database Connectivity) is a standard call level interface (CLI) developed by Microsoft. It is based on the SQL Access Group CLI specification. ODBC applications can run against any data source that provides an ODBC driver. You should use ODBC if you would like your application to be portable to other data sources that have ODBC drivers. Also, if you prefer working with an API, use ODBC. The ODBC interface is fairly low level—about the same as Embedded SQL. Almost all of the SQL Anywhere functionality is available with this interface. ODBC is available as a DLL under Windows, OS/2 and Windows NT and as a library for DOS and QNX.

Dynamic Data Exchange

The Dynamic Data Exchange (DDE) Server can be used from Windows applications that provide DDE client services. Some examples are Microsoft Word, Microsoft Excel and Lotus 1-2-3 for Windows. The DDE interface is set oriented — the entire results of a query are returned to the application at once. The results are returned on the clipboard or in memory. Use the DDE interface from any Windows application that has DDE support and can handle a set of information being returned.

High-level interface (HLI)

The WSQL HLI Dynamic Link Library (DLL) provides a simple, high-level interface to SQL Anywhere in Windows, OS/2 or Windows NT. It is either set or record-at-a-time oriented. You can choose how you would like to retrieve results. The record-at-a-time interface provides more control for use in systems like Visual Basic where data entry applications are being built. You can use the DLL from any application that supports linkage to a DLL. You can also use this DLL from any programming language. If this DLL does not provide everything you need, use the ODBC interface (also a DLL).

CHAPTER 33

The Embedded SQL Interface

About this chapter

This chapter describes the Embedded SQL interface to SQL Anywhere database engines.

> **Structure packing must be turned on**
> All Embedded SQL programs must be compiled with the structure packing option of the compiler set to one-byte alignment (usually the default).

Contents

Topic	Page
The C language SQL preprocessor	656
Embedded SQL interface data types	664
Host variables	667
The SQL communication area (SQLCA)	673
Fetching data	677
Static vs dynamic SQL	682
The SQL descriptor area (SQLDA)	690
SQL procedures in Embedded SQL	696
Library functions	701
Interface library DLL dynamic loading	723
Embedded SQL commands	727
Database examples	730
SQLDEF.H header file	747

The C language SQL preprocessor

Embedded SQL consists of SQL statements intermixed with C or C++ source code. These SQL statements are translated by the Embedded SQL preprocessor into C source code. This code, in conjunction with the database interface dynamic link library (normal library in DOS or QNX), will communicate the appropriate information to the database engine when the program is run. In the compile and link process, the preprocessor is run before the C program is compiled.

Supported compilers

The C language SQL preprocessor has been used in conjunction with the following compilers:

Windows 3.x compilers

- Watcom C/C++ 9.0 and above.
- Microsoft C 5.0, 5.1, 6.0
- Microsoft C++ 7.0
- Microsoft Visual C++ 1.0, 1.5
- Borland C++ 2.0, 3.0, 4.0, 4.5

DOS compilers

- Watcom C/C++ 9.5 and above.
- Microsoft C 6.0, 7.0
- Microsoft Visual C++ 1.0, 1.5
- Borland C++ 3.0, 4.0
- Turbo C 2.0

OS/2 compilers

- Watcom C/C++ 9.0 and above.
- IBM C Set ++ 2.0
- Borland C++ for OS/2 1.0

Windows 95 and NT compilers

- Watcom C/C++ 9.5 and above.
- Microsoft Visual C++ 1.0, 1.5, 2.0, 4.0, 4.1
- Borland C++ 4.5

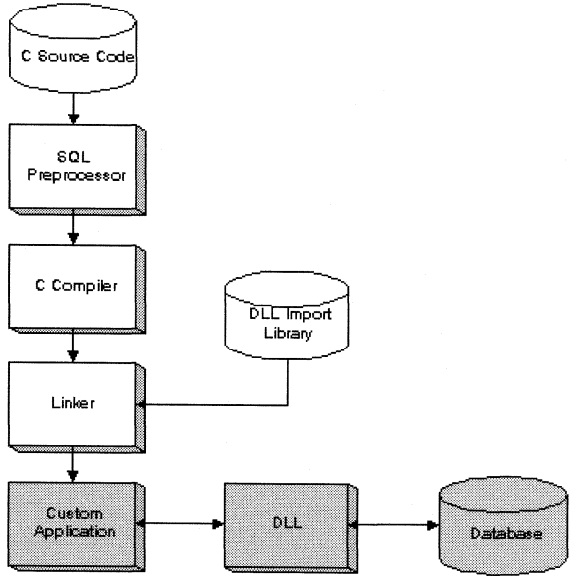
QNX compiler

- Watcom C 9.5 and above.

The following sections present the development process required to build Embedded SQL programs. A section on running the SQL preprocessor is followed by a simple example. The remaining sections present various topics fundamental to the creation of Embedded SQL programs.

Structure packing must be turned on
All Embedded SQL programs must be compiled with the structure packing option of the compiler set to one-byte alignment (usually the default).

Development process for Windows 3.x and 95, OS/2, or Windows NT



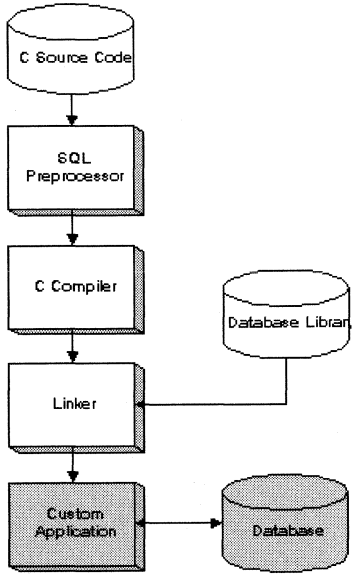
The SQL runtime interface is implemented as a DLL (dynamic link library) for Windows, OS/2 and Windows NT. One Windows import library works for all compilers and memory models. One OS/2 import library works for all compilers and memory models. For Windows 95 and Windows NT, there are import libraries for WATCOM C, for Microsoft Visual C++, and for Borland C++.

Once the program has been successfully preprocessed and compiled, it is linked with the import library for the SQL Anywhere dynamic link library (DLL) to form an executable file. When the database is running (see "The database engine" in the chapter "SQL Anywhere Components"), this executable file will use the SQL Anywhere DLL to interact with the database. The database does not have to be running when the program is preprocessed.

WATCOM C supports 32-bit application development under Windows 3.x. For this environment, an interface library (not a DLL) is provided.

The development process is then the same as DOS.

Development process for DOS or QNX



Once the program has been successfully preprocessed and compiled, it is linked with the SQL runtime library to form an executable file. When the database is running (see "The database engine" in the chapter "SQL Anywhere Components".) this executable file will interact with the database. The database does not have to be running when the program is preprocessed.

The SQL runtime library is available in various memory models. Not all libraries are provided on the distribution diskettes. "Embedded SQL libraries" on page 660 lists which libraries are provided, how they are named and where they can be found.

Running the SQL preprocessor

Normally, SQLPP is run as follows:

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

The SQL preprocessor processes a C program with Embedded SQL before the C compiler is run. SQLPP translates the SQL statements 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* already has a .C extension, then the output filename extension will be CC by default.

The command-line switches for SQLPP are documented in "SQL Anywhere Components". In summary, they are as follows:

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

-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

Embedded SQL header files

All header files are installed in the H subdirectory of the SQL Anywhere directory. For QNX they are in the INCLUDE subdirectory.

Structure packing must be turned on
All Embedded SQL programs must be compiled with the structure packing option of the compiler set to one-byte alignment (usually the default).

Filename	Description
SQLCA.H	Main SQL Anywhere header file included in all embedded SQL programs. This file includes the structure definition for the SQL Communication Area (SQLCA) and prototypes for all embedded SQL database interface functions.
SQLDA.H	SQL Descriptor Area structure definition included in embedded SQL programs that use dynamic SQL.
SQLDEF.H	Definition of Embedded SQL interface data types. This file also contains structure definitions and return codes needed for starting the database engine or SQL Anywhere Client from a C program.
SQLERR.H	Definitions for error codes returned in the **sqlcode** field of the SQLCA.
SQLSTATE.H	Definitions for ANSI/ISO SQL standard error states returned in the **sqlstate** field of the SQLCA.

Embedded SQL libraries

All interface libraries are installed in a subdirectory of the SQL Anywhere installation directory (usually C:\SQLANY50). For QNX the standard installation directory is /usr/lib/sqlany50.

The DOS\LIB directory contains the following import libraries:

- DBLIBWCC.LIB DOS WATCOM C compact memory model library.
- DBLIBWCH.LIB DOS WATCOM C huge memory model library.
- DBLIBWCL.LIB DOS WATCOM C large memory model library.
- DBLIBWCM.LIB DOS WATCOM C medium memory model library.
- DBLIBWCS.LIB DOS WATCOM C small memory model library.
- DBLIBMCC.LIB DOS Microsoft C compact memory model library.
- DBLIBMCH.LIB DOS Microsoft C huge memory model library.
- DBLIBMCL.LIB DOS Microsoft C large memory model library.
- DBLIBMCM.LIB DOS Microsoft C medium memory model library.
- DBLIBMCS.LIB DOS Microsoft C small memory model library.
- DBLIBBCC.LIB DOS Borland C compact memory model library.

- DBLIBBCH.LIB DOS Borland C huge memory model library.
- DBLIBBCL.LIB DOS Borland C large memory model library.
- DBLIBBCM.LIB DOS Borland C medium memory model library.
- DBLIBBCS.LIB DOS Borland C small memory model library.
- DBLIBWFG.LIB 32-bit Rational DOS4G DOS Extender WATCOM C library.
- DBLIBWFP.LIB 32-bit Pharlap DOS Extender WATCOM C library.

The WIN\LIB directory contains the following import libraries:

- DBLIBW.LIB Windows import library for DBLIBW.DLL
- DBLIBWFW.LIB 32-bit Windows WATCOM C library.
- DBLIBFWS.LIB 32-bit Windows WATCOM C stack calling convention library.

The OS2\LIB directory contains the following import library:

- DBLIB2.LIB OS/2 import library for DBLIB2.DLL

The WIN32\LIB directory contains the following import libraries:

- DBLIBTW.LIB Windows 95 and NT WATCOM C/C++ import library for the Embedded SQL interface library.
- DBLIBTB.LIB Windows 95 and NT Borland C++ import library for the Embedded SQL interface library.
- DBLIBTM.LIB Windows 95 and NT Microsoft Visual C++ import library for the Embedded SQL interface library.

The qnx/lib directory contains the following files:

- dblib3r.lib 32-bit QNX WATCOM C library.

A simple example

The following is a very simple example of an Embedded SQL program.

```
#include <stdio.h>
EXEC SQL INCLUDE SQLCA;
main()
{
    db_init( &sqlca );
    EXEC SQL WHENEVER SQLERROR GOTO error;
    EXEC SQL CONNECT "dba" IDENTIFIED BY "sql";
    EXEC SQL UPDATE employee
```

```
        SET emp_lname =    'Plankton'
        WHERE emp_id = 195;
    EXEC SQL COMMIT WORK;
    EXEC SQL DISCONNECT;
    db_fini( &sqlca );
    return( 0 );

    error:
    printf( "update unsuccessful -- sqlcode = %ld.n",
    sqlca.sqlcode );
    return( -1 );
}
```

This example connects to the database, updates the surname of employee number 1056, commits the change and exits. There is virtually no interaction between the SQL and C code. The only thing the C code is used for in this example is control flow. The WHENEVER statement is used for error checking. The error action (GOTO in this example) will be executed after any SQL statement that causes an error.

Note that the first section of this chapter uses UPDATE and INSERT examples because they are simpler. "Fetching data" on page 677 will discuss fetching data.

Structure of Embedded SQL programs

SQL statements are placed (embedded) within regular C or C++ code. All Embedded SQL statements start with the words EXEC SQL and end with a semicolon (;). Normal C language comments are allowed in the middle of Embedded SQL statements.

Every C program using Embedded SQL must contain the following statement before any other Embedded SQL statements in the source file.

```
EXEC SQL INCLUDE SQLCA;
```

The first Embedded SQL statement executed by the C program must be a CONNECT statement. The CONNECT statement is used to establish a connection with the database engine and to specify the userid that will be used for authorizing all statements executed during the connection (see "Authorization" on page 663 below).

> **There are two kinds of ordering on statements:**
> These are the order in which they appear in the source file, and the order in which they are executed when the program is run.

As noted above, the CONNECT statement must be the first executed. Note that some Embedded SQL commands do not generate any C code or do not involve communication with the database. These commands are thus allowed before the CONNECT statement. Most notably are the INCLUDE statement and the WHENEVER statement for specifying error processing.

Authorization

The CONNECT statement specifies a userid and password that will be used for checking the authorization of any dynamic SQL statements used in the program (defined in "Dynamic statements" on page 683). It will also be used for authorization of any static statements contained in modules which were preprocessed without the -l option on the SQLPP command line.

Static SQL statements in modules preprocessed with the -l option will have the authorization checked at execution time using the userid and password specified with the -l option. This is how privileged commands can be executed by non-privileged users under C program control.

Embedded SQL interface data types

In order to transfer information between a program and the database engine, every piece of data must have a data type. The following data types are supported by the embedded SQL programming interface. The Embedded SQL data type constants (prefixed with DT_) can be found in the SQLDEF.H header file (see "SQLDEF.H header file" on page 747). You can create a host variable (next section) of any one of these types. You can also use these types in an SQLDA structure (described later) for passing data to and from the database.

Data type constant	Description
DT_SMALLINT	16 bit, signed integer.
DT_INT	32 bit, signed integer.
DT_FLOAT	4 byte floating point number.
DT_DOUBLE	8 byte floating point number.

DT_DECIMAL	Packed decimal number. Decimal digits are packed into an array of characters, two digits per byte. The most significant digits are stored first. The low order nibble (half-byte) of the last byte is the sign (0xC for positive, 0xD for negative). If the precision is even, the first digit stored is a 0.
	`typedef struct DECIMAL {`
	`char array[1];`
	`} DECIMAL;`

DT_STRING	NULL-terminated character string.
DT_DATE	NULL-terminated character string that is a valid date.
DT_TIME	NULL-terminated character string that is a valid time.
DT_TIMESTAMP	NULL-terminated character string that is a valid timestamp.
DT_FIXCHAR	Fixed-length blank padded character string.

DT_VARCHAR	Varying length character string with a two byte length field. The length field must be set when supplying information to the database engine. The database engine will set the length field when you fetch information.
	`typedef struct VARCHAR {`
	`unsigned short int len;`
	`char array[1];`
	`} VARCHAR;`

DT_BINARY	Varying length binary data with a two byte length field. The length field must be set when supplying information

Data type constant	Description

	to the database engine. The database engine will set the length field when you fetch information.
	`typedef struct BINARY {`
	`unsigned short int len;`
	`char array[1];`
	`} BINARY;`

DT_TIMESTAMP_ STRUCT	SQLDATETIME structure with fields for each part of a timestamp.
	`typedef struct sqldatetime {`
	`unsigned short year; /* e.g. 1992 */`
	`unsigned char month; /* 0-11 */`
	`unsigned char day_of_week; /* 0-6 0=Sunday */`
	`unsigned short day_of_year; /* 0-365 */`
	`unsigned char day; /* 1-31 */`
	`unsigned char hour; /* 0-23 */`
	`unsigned char minute; /* 0-59 */`
	`unsigned char second; /* 0-59 */`
	`unsigned long microsecond; /* 0-999999 */`
	`} SQLDATETIME;`
	The SQLDATETIME struct can be used to retrieve fields of DATE, TIME, and TIMESTAMP type (or anything that can be converted to one of these). Often, applications have their own formats and date manipulation code. Fetching data in this structure makes it easier for a programmer to manipulate this data. Note that DATE, TIME and TIMESTAMP fields can also be fetched and updated with any character type. See the DATE_FORMAT, TIME_FORMAT, TIMESTAMP_FORMAT, and DATE_ORDER database options in "SET OPTION statement" in the chapter "Watcom-SQL Statements".

DT_VARIABLE	NULL-terminated character string. The character string must be the name of a SQL variable whose value will be used by the database engine (see "SET statement" in the chapter "Watcom-SQL Statements"). This data type is only used when supplying data to the database engine. It cannot be used when fetching data from the database engine.

The structures are defined in the SQLCA.**h** file. The VARCHAR, BINARY and DECIMAL types contain a one-character array and are thus not useful for declaring host variables but they are useful for allocating variables dynamically or typecasting other variables.

There are no corresponding embedded SQL interface data types for the various DATE and TIME database types. These database types are all fetched and updated using either the SQLDATETIME structure or character strings.

There are no embedded SQL interface data types for LONG VARCHAR and LONG BINARY database types. These database types are fetched and updated in pieces. For more information see "GET DATA statement" in the chapter "Watcom-SQL Statements" and "SET statement" in the chapter "Watcom-SQL Statements".

Host variables

Host variables are C variables which are identified to the SQL preprocessor. Host variables can be used to send values to the database engine or receive values from the database engine.

Host variables are quite easy to use, but they have some restrictions. "Static vs dynamic SQL" on page 682 describes dynamic SQL and the more general way of passing information to and from the database engine using a structure known as the SQL Descriptor Area (SQLDA). This section contains information on the following topic:

- "Host variable declarations" on page 667
- "C host variable types" on page 668
- "Host variable usage" on page 670
- "Indicator variables" on page 671

Host variable declarations

Host variables are defined by putting them into a **declaration section**. According to the IBM SAA and the ANSI Embedded SQL standards, host variables are defined by surrounding the normal C variable declarations with:

```
EXEC SQL BEGIN DECLARE SECTION;
/* C variable declarations */
EXEC SQL END DECLARE SECTION;
```

These host variables can then be used in place of value constants in any SQL statement. When the database engine executes the command, the value of the host variable will be used. Note that host variables cannot be used in place of table or column names; dynamic SQL is required for this. The variable name is prefixed with a colon (':') in an SQL statement to distinguish it from other identifiers allowed in the statement.

A standard SQL preprocessor does not scan C language code except inside a declare section. Thus, typedef types and structures are not allowed. Initializers on the variables are allowed inside a declare section.

The following is an example of the use of host variables on an INSERT command. The variables are filled in by the program and then inserted into the database:

```
EXEC SQL BEGIN DECLARE SECTION;
long employee_number;
```

```
char employee_name[50];
char employee_initials[8];
char employee_phone[15];
EXEC SQL END DECLARE SECTION;
/* program fills in variables with appropriate
values
*/
EXEC SQL INSERT INTO Employee
VALUES (:employee_number, :employee_name,
:employee_initials, :employee_phone );
```

For a more extensive example, see see "Static cursor example" on page 732.

C host variable types

Only a limited number of C data types are supported as host variables. Also, certain host variable types do not have a corresponding C type.

There are macros defined in the SQLCA.H header file that can be used to declare a host variable of these types: VARCHAR, FIXCHAR, BINARY, PACKED DECIMAL, or SQLDATETIME structure. They are used as follows:

```
EXEC SQL BEGIN DECLARE SECTION;
DECL_VARCHAR( 10 ) v_varchar;
DECL_FIXCHAR( 10 ) v_fixchar;
DECL_BINARY( 4000 ) v_binary;
DECL_DECIMAL( 10, 2 ) v_packed_decimal;
DECL_DATETIME v_datetime;
EXEC SQL END DECLARE SECTION;
```

The preprocessor recognizes these macros within a declaration section and will treat the variable as the appropriate type.

The following table illustrates C variable types allowed for host variables and their corresponding embedded SQL interface data types.

C Data Type	Embedded SQL Interface Type	Description
short i; short int I; unsigned short int i;	DT_SMALLINT	16 bit, signed integer
long l; long int l; unsigned long int l;	DT_INT	32 bit, signed integer
float f;	DT_FLOAT	4 byte floating point
double d;	DT_DOUBLE	8 byte floating point

C Data Type	Embedded SQL Interface Type	Description
DECL_DECIMAL(p,s)	DT_DECIMAL(p,s)	packed decimal
char a; /*n=1*/ DECL_FIXCHAR(n) a; DECL_FIXCHAR a[n];	DT_FIXCHAR(n)	fixed length character string blank padded
char a[n]; /*n>=1*/	DT_STRING(n)	NULL-terminated string
char *a;	DT_STRING(32767)	NULL-terminated string-see note
DECL_VARCHAR(n) a;	DT_VARCHAR(n)	varying length character string with 2 byte length field
DECL_BINARY(n) a;	DT_BINARY(n)	varying length binary data with 2 byte length field
DECL_DATETIME a;	DT_TIMESTAMP_STRUCT	SQLDATETIME structure

Pointers to char

A host variable declared as a pointer to char (*char *a*) is considered by the database interface to be 32,767 bytes long. Thus, you must ensure that any host variable of type pointer to char used to retrieve information from the database points to a buffer large enough to hold any value that could possibly come back from the database. Note that this is potentially quite dangerous as somebody could change the definition of the column in the database to be larger than it was when the program was written. This could cause random memory corruption problems. If you are using a 16-bit compiler, requiring 32,767 bytes could result in overflowing the program stack. It is better to use a declared array, even as a parameter to a function where it is passed as a pointer to char. This lets the PREPARE statements know the size of the array.

Scope of host variables

A standard host variable declaration section can appear anywhere that C variables can normally be declared. This includes the parameter declaration section of a C function. The C variables have their normal scope (available within the block in which they are defined). However, since the SQL preprocessor does not scan C code, it does not respect C blocks. As far as the SQL preprocessor is concerned, host variables are global; two host variables cannot have the same name (a warning will result). The SQL preprocessor has one exception to this rule. It will allow two host variables of the same name if they have identical types (including any necessary lengths).

Host variable usage

- Host variables can be used in SELECT, INSERT, UPDATE and DELETE statements in any place where a number or string constant is allowed.
- Host variables can be used in the INTO clause of SELECT and FETCH statements.
- Host variables can also be used in place of a statement name, a cursor name, or an option name in Embedded SQL specific commands.
- For CONNECT, DISCONNECT, and SET CONNECT a host variable can be used in place of a userid, password, connection name or database environment name.
- For SET OPTION and GET OPTION, a host variable can be used in place of a userid, option name or option value.
- Host variables cannot be used in place of a table name or a column name in any statement.

For releases starting with 5.0.02, you can declare SQLSTATE and SQLCODE as host variables within a declare section. This is provided for ISO/ANSI compatibility.

Before release 5.0.02, SQLSTATE and SQLCODE were macros for values in the sqlca, and this is still true if neither is specified in your program. However, if you declare SQLSTATE or SQLCODE anywhere in the program, then the macro definitions are removed. This means that if a program contains a declaration of SQLCODE/SQLSTATE, then a SQLSTATE/SQLSTATE must be within the scope of every embedded SQL statement in the program.

Examples

The following is a valid program:

```
INCLUDE SQLCA;
long SQLCODE;
sub1() {
   char SQLSTATE[6];
   exec sql CREATE TABLE ...
}
```

The following is an invalid program:

```
INCLUDE SQLCA;

sub1() {
   char SQLSTATE[6];
   exec sql CREATE TABLE...
}

sub2() {
```

```
    exec sql DROP TABLE...
    // No SQLSTATE in scope of this statement
}
```

The case of SQLSTATE and SQLCODE is important, and the ISO/ANSI standard requires that their definitions be exactly as follows:

```
long SQLCODE;
char SQLSTATE[6];
```

Indicator variables

The NULL value is a value that can be put into certain columns in the database. It represents either an unknown attribute or inapplicable information. See "NULL value" in the chapter "Watcom-SQL Statements" for a complete description of NULLs.

> **NULL in SQL and C**
> NULL is not to be confused with the C language constant by the same name (**NULL**). The C constant is used to represent a non-initialized or invalid pointer. When NULL is used in this book, it refers to the SQL database meaning given above. The C language constant will be referred to as the null pointer (lower case).

NULL is not the same as any value of the column's defined type. Thus, in order to pass NULL values to the database or receive NULL results back, something extra is required beyond regular host variables. **Indicator variables** are used for this purpose. An indicator variable is a host variable of type **short int** that is placed immediately following a regular host variable in an SQL statement. For example, the previous insert statement example could include an indicator variable as follows:

```
EXEC SQL BEGIN DECLARE SECTION;
short int employee_number;
char employee_name[50];
char employee_initials[6];
char employee_phone[15];
short int ind_phone;
EXEC SQL END DECLARE SECTION;

/*
program fills in empnum, empname,
initials and homephone
*/
if( /* phone number is unknown */ ) {
    ind_phone = -1;
} else {
    ind_phone = 0;
```

```
}
EXEC SQL INSERT INTO Employee
   VALUES (:employee_number, :employee_name,
   :employee_initials, :employee_phone:ind_phone );
```

In this example, a NULL is assigned to the phone number if the phone number is unknown. This is accomplished by assigning negative one (-1) to the indicator variable.

Indicator variables are also used when receiving data from the database. They are used to indicate that a NULL value was fetched (indicator is negative). If a NULL value is fetched from the database and an indicator variable is not supplied, then an error is generated (SQLE_NO_INDICATOR). Errors are explained in the next section.

Indicator variables are also used to indicate whether any fetched values were truncated to fit into the host variables (indicator has positive value). In this case, the indicator variable contains the actual length of the database value before truncation. If the length of the value is greater than 32767, then the value must be truncated and the indicator variable contains 32767.

For more information on retrieving long values, see "GET DATA statement" in the chapter "Watcom-SQL Statements".

See the following table for a more complete description of indicator variable usage.

Indicator Value	Supplying Value to database	Receiving value from database
> 0	Host variable value	Retrieved value was truncated — actual length in indicator variable
= 0	Host variable value	Fetch successful
= -1	NULL value	NULL result
= -2	NULL value	Conversion error SQLCODE will indicate a conversion error
< -2	NULL value	NULL result

The SQL communication area (SQLCA)

The **SQL Communication Area** (**SQLCA**) is an area of memory used on every database request for communicating statistics and errors from the application to the database engine and back to the application. There is a global SQLCA variable defined in the interface library (imports library for the DLLs). An external reference for this variable named **sqlca** of type SQLCA and an external reference for a pointer to this variable *sqlcaptr* are automatically generated by the preprocessor. The actual global variable is declared in the database library (imports library for DLLs).

Structure packing option
All Embedded SQL programs must be compiled with the structure packing option of the compiler set to one-byte alignment (usually the default).

```
typedef long int     an_sql_code;
typedef char an_sql_state[6];

struct sqlwarn{ unsigned char sqlwarn0;
   unsigned char  sqlwarn1;
   unsigned char  sqlwarn2;
   unsigned char  sqlwarn3;
   unsigned char  sqlwarn4;
   unsigned char  sqlwarn5;
   unsigned char  sqlwarn6;
   unsigned char  sqlwarn7;
   unsigned char  sqlwarn8;
   unsigned char  sqlwarn9;
};
typedef struct sqlca {
   unsigned char sqlcaid[8];
   long sqlcabc;
   an_sql_code sqlcode;
   short sqlerrml;
   unsigned char sqlerrmc[70];
   unsigned char sqlerrp[8];
   long sqlerrd[6];
   struct sqlwarn sqlwarn;
   an_sql_state   sqlstate;
} SQLCA;

#define SQLCODE sqlcaptr->sqlcode
#define SQLSTATE sqlcaptr->sqlstate
#define SQLIOCOUNT sqlcaptr->sqlerrd[1]
#define SQLCOUNT sqlcaptr->sqlerrd[2]
#define SQLIOESTIMATE sqlcaptr->sqlerrd[3]
```

The user references the SQLCA to test for a particular error code. The sqlcode and sqlstate fields contain error codes when a database request has an error (see below). As shown in there are some C macros defined for referencing the sqlcode field, the sqlstate field and some other fields.

The SQLCA is used as a handle for the application to database communication link. It is passed in to all database library functions that need to communicate with the database engine. It is implicitly passed on all Embedded SQL statements.

The fields in the SQLCA have the following meanings:

Field	Description
sqlcaid	An 8 byte character field that contains the string *"SQLCA "* as an identification of the SQLCA structure. This field helps in debugging when looking at memory contents.
sqlcabc	A long integer containing the length of the SQLCA structure (136 bytes).
sqlcode	A long integer specifying the error code when the database detects an error on a request. Definitions for the error codes can be found in the header file **sqlerr.h**. These error codes are fully explained in he chapter "SQL Anywhere Database Error Messages". The error code will be 0 for a successful operation, positive for a warning and negative for an error.
sqlerrml	The length of the information in the *sqlerrmc* field.
sqlerrmc	May contain one or more character strings to be inserted into an error message. See "SQL Anywhere Database Error Messages" for a list of the error codes and the error messages. Some error messages have a *%1* in them which is replaced with the text in this field. For example, if a table not found error is generated, this field will contain the table name which is inserted into the error message at the appropriate place.
sqlerrp	Reserved.
sqlerrd	A utility array of longs.
sqlwarn	Reserved.
sqlstate	The SQLSTATE status value. The ANSI SQL standard (SQL-92) defines a new type of return value from a SQL statement in addition to the SQLCODE value in previous standards. The SQLSTATE value is always a five character null-terminated string, divided into a two character class (the first two characters) and a three character subclass. Each character can be the digits '0' through '9' or the upper case alphabetic characters 'A' through 'Z'.

Field	Description
	Any class or subclass that begins with '0' through '4' or 'A' through 'H' is defined by the SQL standard; other classes and subclasses are implementation defined. The SQLSTATE value '00000' means that there has been no error or warning. Other SQLSTATE values used by SQL Anywhere are described in "SQL Anywhere Database Error Messages".

The sqlerror field array has the following elements.

Element	Description
sqlerrd[1] (SQLIOCOUNT)	The actual number of input/output operations required to complete a command. The database does not start this number at zero for each command. Your program can set this variable to zero before executing a sequence of commands. After the last command, this number will be the total number of input/output operations for the entire command sequence.
sqlerrd[2] (SQLCOUNT)	The number of rows affected by the command (for INSERT, UPDATE and DELETE). On a cursor OPEN, this field is 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 "SET OPTION statement" in the chapter "Watcom-SQL Statements"). On a FETCH cursor statement, this field is filled if a SQLE_NOTFOUND warning is returned. It will contain the number of rows that the a FETCH RELATIVE or FETCH ABSOLUTE statement has 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.

Element	Description
	On a GET DATA statement, this field holds the actual length of the value. In the case of a syntax error, SQLE_SYNTAX_ERROR, this field will contain the approximate character position within the command string where the error was detected..
sqlerrd[3] (SQLIOESTIMATE)	The estimated number of input/output operations required to complete the command. This field is filled in on an OPEN or EXPLAIN command.

Fetching data

Fetching data in Embedded SQL is done using the SELECT statement. There are two cases:

1 The SELECT statement returns at most one row.

2 The SELECT statement may return multiple rows.

The next two sections describe these cases.

Fetches retrieving multiple rows at a time are also allowed, and may improve performance. These are discussed in "Fetching more than one row at a time" on page 686.

Embedded SELECT

Single row queries retrieve at most one row from the database. This type of query is achieved by embedding a normal SQL SELECT statement with an INTO clause. The INTO clause follows the select list and precedes the FROM clause. It contains a list of host variables to receive the value for each select list item. There must be the same number of host variables as there are select list items.

The host variables may be accompanied by indicator variables to indicate NULL results.

When a SELECT statement is executed, the database engine retrieves the results of the select statement and places the results in the host variables. If the query results contain more than one row, the database engine will return an error. In this case, **cursors** must be used (see next section). If the query results in no rows being selected, a **row not found** warning is returned. Errors and warnings are returned in the SQLCA structure described in "The SQL communication area (SQLCA)" on page 673.

For example, the following code fragment returns 1 if a row from the employee table is successfully fetched, 0 if the row doesn't exist and -1 if an error occurs.

```
EXEC SQL BEGIN DECLARE SECTION;
   long        emp_id;
   char        name[41];
   char        sex;
   char        birthdate[15];
   short int   ind_birthdate;
EXEC SQL END DECLARE SECTION;
. . .
int find_employee( long employee )
```

```
{
    emp_id = employee;
    EXEC SQL SELECT emp_fname ||
             ' ' || emp_lname, sex, birth_date
             INTO :name, :sex,
                  birthdate:ind_birthdate
             FROM "dba".employee
             WHERE emp_id = :emp_id;
    if( SQLCODE == SQLE_NOTFOUND ) {
       return( 0 ); /* employee not found */
    } else if( SQLCODE < 0 ) {
       return( -1 ); /* error */
    } else {
       return( 1 ); /* found */
    }
}
```

Cursors in Embedded SQL

A cursor is used to retrieve rows one at a time from a query that has multiple rows in the result set. A **cursor** is a handle or an identifier for the SQL query and a position within the results. Managing a cursor is similar to managing files in a programming language. The following steps are used to manage cursors:

1 A cursor is declared for a particular select statement using the DECLARE statement.

2 The cursor is opened using the OPEN statement.

3 The FETCH statement is used to retrieve results one row at a time from the cursor.

4 Usually records are fetched until the Row Not Found warning is returned. Errors and warnings are returned in the SQLCA structure described in "The SQL communication area (SQLCA)" on page 673. The cursor is then closed using the CLOSE statement.

By default, cursors are automatically closed at the end of a transaction (on COMMIT or ROLLBACK). Cursors that are opened with a WITH HOLD clause will be kept open for subsequent transactions until they are explicitly closed.

The following is a simple example of cursor usage:

```
void print_employees( void )
{
    int status;
    EXEC SQL BEGIN DECLARE SECTION;
    char name[50];
    char sex;
```

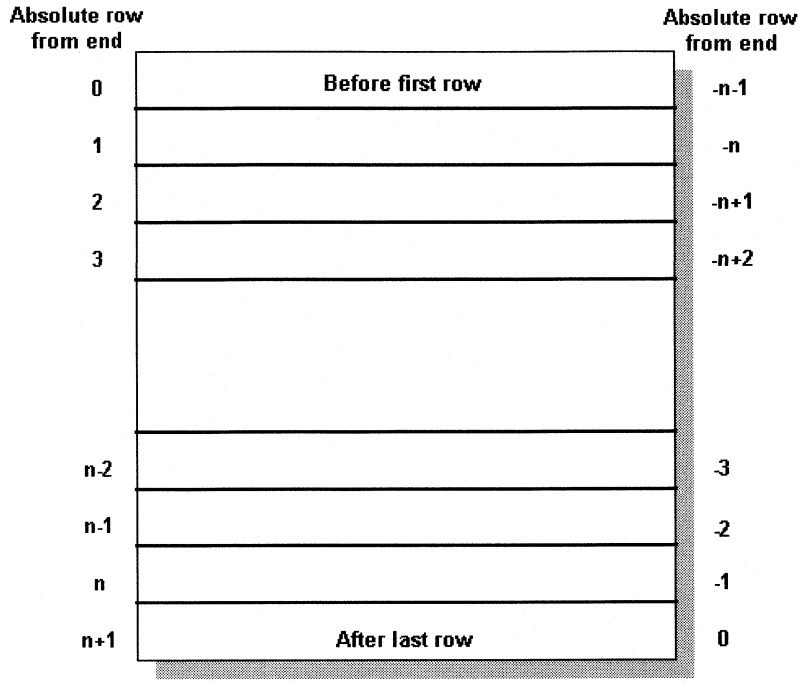
```
    char birthdate[15];
    short int ind_birthdate;
    EXEC SQL END DECLARE SECTION;
    EXEC SQL DECLARE C1 CURSOR FOR
       SELECT   emp_fname || ' ' || emp_lname,
                sex, birth_date
       FROM "dba".employee;
    EXEC SQL OPEN C1;
    for( ;; ) {
       EXEC SQL FETCH C1 INTO :name, :sex,
:birthdate:ind_birthdate;
       if( SQLCODE == SQLE_NOTFOUND ) {
          break;
       } else if( SQLCODE < 0 ) {
          break;
       }
       if( ind_birthdate < 0 ) {
          strcpy( birthdate, "UNKNOWN" );
       }
       printf( "Name: %s Sex: %c Birthdate:
                %s.n",name, sex, birthdate );
    }
EXEC SQL CLOSE C1;
}
```

For complete examples using cursors, see "Static cursor example" on page 732 and "Dynamic cursor example" on page 738.

Cursor positioning

A cursor is positioned in one of three places:

- On a row
- Before the first row
- After the last row



When a cursor is opened, it is positioned before the first row. The cursor position can be moved using the FETCH command (see "FETCH statement" in the chapter "Watcom-SQL Statements"). It can be positioned to an absolute position either from the start or from the end of the query results. It can also be moved relative to the current cursor position.

There are special *positioned* versions of the UPDATE and DELETE statements that can be used to update or delete the row at the current position of the cursor. If the cursor is positioned before the first row or after the last row, a No Current Row of Cursor error will be returned.

The PUT statement can be used to insert a row into a cursor.

Cursor positioning problems
Inserts and some updates to DYNAMIC SCROLL cursors can cause problems with cursor positioning. The database engine does 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 does 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" in the chapter "Monitoring and Improving Performance" for a description).

The UPDATE statement may cause a row to move in the cursor. This happens if the cursor has an ORDER BY that uses an existing index (a temporary table is not created).

Static vs dynamic SQL

There are two ways to embed SQL statements into a C program:

- "Static statements" on page 682
- "Dynamic statements" on page 683

Static statements

All standard SQL data manipulation and data definition statements can be embedded in a C program by prefixing them with EXEC SQL and suffixing the command with a semicolon (;). These statements are referred to as **static** statements. Static statements can contain references to host variables as described in the section.Host variables on page 667. All examples to this point have used static Embedded SQL statements. Remember that host variables can only be used in place of string or numeric constants. They cannot be used to substitute column names or table names (dynamic statements are required to do those operations).

The following commands can be used statically:

```
ALTER TABLE                    INSERT
CHECKPOINT                     PREPARE TO COMMIT
COMMIT                         RELEASE SAVEPOINT
COMMENT                        REVOKE
CONNECT                        ROLLBACK
CREATE                         ROLLBACK TO SAVEPOINT
DELETE                         SAVEPOINT
DISCONNECT                     SET CONNECTION
DROP                           SET OPTION
EXPLAIN                        UPDATE
GRANT                          VALIDATE TABLE
```

The SELECT statement is different in Embedded SQL than in interactive SQL (ISQL).

As described earlier, if the SELECT statement retrieves only one row from the database, then the statement can be coded as a normal SELECT command except that an INTO clause must follow the select list. If the select statement retrieves multiple rows from the database, then a **cursor** must be used for the select command as described previously.

Dynamic statements

Dynamic statements are SQL statements which are constructed in C language strings. (In the C language, strings are stored in arrays of characters.) These statements can then be executed using the Embedded SQL PREPARE and EXECUTE statements. These SQL statements cannot reference host variables in the same manner as static statements since the C language variables are not accessible by name when the C program is executing.

In order to pass information between the statements and the C language variables, a data structure called the **SQL Descriptor Area (SQLDA)** is used (see "The SQL descriptor area (SQLDA)" on page 690). This structure will be set up for you by the SQL preprocessor if you specify a list of host variables on the EXECUTE command in the USING clause. These variables correspond by position to place holders in the prepared command string.

A **place holder** is put in the statement to indicate where host variables are to be accessed. A place holder is either a question mark ('?') or a host variable reference as in static statements (a host variable name preceded by a colon). In the latter case, the host variable name used in the actual text of the statement serves only as a place holder indicating a reference to the SQL descriptor area.

A host variable used to pass information to the database is called a **bind variable**.

For example:

```
EXEC SQL BEGIN DECLARE SECTION;
   char comm[200];
   char address[30];
   char city[20];
   short int cityind;
   long empnum;
EXEC SQL END DECLARE SECTION;
. . .
   sprintf( comm, "update %s set address = :?,
                        city = :?"
                " where employee_number = :?",
                tablename );
EXEC SQL PREPARE S1 FROM :comm;
```

```
EXEC SQL EXECUTE S1 USING :address, :city:cityind,
:empnum;
```

This method requires the programmer to know how many host variables there are in the statement. Usually, this is not the case. So, you can set up your own SQLDA structure and specify that SQLDA in the USING clause on the EXECUTE command. The DESCRIBE BIND VARIABLES statement will return the host variable names of the bind variables found in a prepared statement. This makes it easier for a C program to manage the host variables. The general method is as follows:

```
EXEC SQL BEGIN DECLARE SECTION;
   char comm[200];
EXEC SQL END DECLARE SECTION;
. . .
sprintf( comm, "update %s set address = :address,
          city = :city"
         " where employee_number = :empnum",
         tablename );
EXEC SQL PREPARE S1 FROM :comm;
/* Assume that there are no more than 10 host
variables. See next example if you can't put
a limit on it */
sqlda = alloc_sqlda( 10 );
EXEC SQL DESCRIBE BIND VARIABLES FOR S1 USING
DESCRIPTOR sqlda;
/* sqlda->sqld will tell you how many host variables
there were. */
/* Fill in SQLDA_VARIABLE fields with values based
on
name fields in sqlda */
. . .
EXEC SQL EXECUTE S1 USING DESCRIPTOR sqlda;
free_sqlda( sqlda );
```

The SQLDA consists of an array of variable descriptors (see "The SQL descriptor area (SQLDA)" on page 690 for a complete description of the SQLDA structure). Each descriptor describes the attributes of the corresponding C program variable or location at which the database will store data into or retrieve data from:

- data type,
- length if **type** is a string type,
- precision and scale if **type** is a numeric type,
- memory address,
- indicator variable.

The indicator variable is used to pass a NULL value to the database or retrieve a NULL value from the database. The indicator variable is also used by the database engine to indicate truncation conditions encountered during a database operation. The indicator variable is set to a positive value when not enough space was provided to receive a database value. See "Indicator variables" on page 671 for more information.

Dynamic SELECT statement

A SELECT statement that returns only a single row can be PREPARED dynamically, followed by an EXECUTE with an INTO clause to retrieve the one-row result. SELECT statements that return multiple rows, however, are managed using dynamic cursors.

With dynamic cursors, results are put into a host variable list or an SQLDA that is specified on the fetch statement (FETCH INTO hostlist and FETCH USING DESCRIPTOR sqlda). Since the number of select list items is usually unknown to the C programmer, the SQLDA route is the most common. The DESCRIBE SELECT LIST statement will set up an SQLDA with the types of the select list items. Space is then allocated for the values using the fill_sqlda() function and the information is retrieved using the FETCH USING DESCRIPTOR statement. The typical scenario is:

```
EXEC SQL BEGIN DECLARE SECTION;
   char comm[200];
EXEC SQL END DECLARE SECTION;
   int actual_size;
   SQLDA * sqlda;

. . .
sprintf( comm, "select * from %s", table_name );
EXEC SQL PREPARE S1 FROM :comm;
/* Initial guess of 10 columns in result. If it is
   wrong, it will be corrected right after the first
   DESCRIBE by reallocating sqlda and doing DESCRIBE
   again. */
sqlda = alloc_sqlda( 10 );
EXEC SQL DESCRIBE SELECT LIST FOR S1 USING
DESCRIPTOR sqlda;
if( sqlda->sqld > sqlda->sqln ){
   actual_size = sqlda->sqld;
   free_sqlda( sqlda );
   sqlda = alloc_sqlda( actual_size );
   EXEC SQL DESCRIBE SELECT LIST FOR S1
      USING DESCRIPTOR sqlda;
}
fill_sqlda( sqlda );
EXEC SQL DECLARE C1 CURSOR FOR S1;
EXEC SQL OPEN C1;
```

```
EXEC SQL WHENEVER NOTFOUND {break};
for( ;; ){
   EXEC SQL FETCH C1 USING DESCRIPTOR sqlda;
   if( SQLCODE == SQLE_NOTFOUND ) break;
   /* do something with data */
}
EXEC SQL CLOSE C1;
EXEC SQL DROP STATEMENT S1;
```

Drop statements after use
You should ensure that statements are dropped after use, to avoid consuming unnecessary resources.

For a complete example using cursors for a dynamic select statement , see "Dynamic cursor example" on page 738. For details of the functions mentioned above, see "Library functions" on page 701.

Fetching more than one row at a time

The FETCH statement can be modified to fetch more than one row at a time, which may improve performance. This is called a **wide fetch**.

SQL Anywhere also supports wide puts and inserts. For information on these, see "PUT statement" in the chapter "Watcom-SQL Statements" and "EXECUTE statement" in the chapter "Watcom-SQL Statements".

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 will be placed in SQLDA variables 0 to (columns per row)-1 and so on.

The engine returns in SQLCOUNT the number of records fetched and will always return a SQLCOUNT greater than zero unless there is an error. Older versions of the engine or server will only return a single row and the SQLCOUNT will be set to zero. Thus a SQLCOUNT of zero with no error condition indicates one valid row has been fetched.

The following example code illustrates the use of wide fetches. The example code is not compilable as it stands.

```
EXEC SQL BEGIN DECLARE SECTION;
static unsigned             FetchWidth;
EXEC SQL END DECLARE SECTION;
```

```
static SQLDA * DoWideFetches( a_sql_statement_number
stat0,
                              unsigned
*num_of_rows,
                              unsigned
*cols_per_row )
/*****************************************************
******************/
// Allocate an SQLDA to be used for fetching from
the statement identified
// by "stat0".  "width" rows will be retrieved on
each FETCH request.
// The number of columns retrieved per row is
assigned to "cols_per_row".
{
    int                     num_cols;
    unsigned                i, j, offset;
    SQLDA *                 sqlda;
    EXEC SQL BEGIN DECLARE SECTION;
    a_sql_statement_number  stat;
    EXEC SQL END DECLARE SECTION;

    stat = stat0;
    sqlda = alloc_sqlda( 100 );
    if( sqlda == NULL ) return( NULL );
    EXEC SQL DESCRIBE :stat INTO sqlda;
    *cols_per_row = num_cols = sqlda->sqld;
    if( (num_cols * *num_of_rows) > sqlda->sqln ) {
        free_sqlda( sqlda );
        sqlda = alloc_sqlda( num_cols * width );
        if( sqlda == NULL ) return( NULL );
        EXEC SQL DESCRIBE :stat INTO sqlda;
    }
    sqlda->sqld = num_cols * *num_of_rows;
    offset = num_cols;
    for( i = 1; i < width; ++i ) {
        for( j = 0; j < num_cols; ++j, ++offset ) {
            sqlda->sqlvar[offset].sqltype = sqlda-
>sqlvar[j].sqltype;
            sqlda->sqlvar[offset].sqllen = sqlda-
>sqlvar[j].sqllen;
            memcpy( &sqlda->sqlvar[offset].sqlname,
                    &sqlda->sqlvar[j].sqlname,
                    sizeof( sqlda->sqlvar[0].sqlname
) );
        }
    }
    fill_sqlda( sqlda );
    return( sqlda );
}

long DoQuery( char * qry )
/************************/
{
    long                    rows;
```

```
    unsigned                cols_per_row;
    SQLDA *                 sqlda;
    EXEC SQL BEGIN DECLARE SECTION;
    a_sql_statement_number  stat;
    static unsigned         num_of_rows;
    EXEC SQL END DECLARE SECTION;

    rows = 0L;
    FetchWidth = 20;

    EXEC SQL WHENEVER SQLERROR GOTO err;

    stmt = qry;
    EXEC SQL PREPARE :stat FROM :stmt;

    EXEC SQL DECLARE QCURSOR CURSOR FOR :stat FOR
READ ONLY;

    EXEC SQL OPEN QCURSOR;
    sqlda = DoWideFetches( stat, &num_of_rows,
&cols_per_row );
    if( sqlda == NULL ) {
        printf( "Maximum allowable fetch width
exceeded\n" );
        return( SQLE_NO_MEMORY );
    }

    for( ;; ) {
        EXEC SQL FETCH QCURSOR INTO DESCRIPTOR sqlda
ARRAY :FetchWidth;
        if (SQLCODE != SQLE_NOERROR) break;
    if( SQLCOUNT == 0 ) {
        rows += 1;
    } else {
        rows += SQLCOUNT;
    }
    }

    EXEC SQL CLOSE QCURSOR;
    EXEC SQL DROP STATEMENT :stat;
    free_sqlda( sqlda );

err:
    if (SQLCODE != SQLE_NOERROR) {
        printf( "Error detected\n" );
    }

    return (SQLCODE);
}
```

Notes on using wide fetches

- In 16-bit environments (including extended 32-bit DOS and Windows 3.x) the limit on the size of a SQLDA is 1450 columns. Thus the number of rows times the number of columns per row must be no more than 1450. In these environments, the SQLDA itself (not including data items) must fit in a single 64K segment. **alloc_sqlda** will return NULL in those environments on an attempt to allocate a SQLDA that is too large.
- In the function DoWideFetches, the SQLDA memory is allocated using the **alloc_sqlda** function, which allows space for indicator variables, rather than using the **alloc_sqlda_noind** function. This is required, as when fewer than the requested number of rows are fetched (at the end of the cursor, for example) the SQLDA items corresponding to the rows not fetched are returned as NULLs by setting the indicator value. If no indicator variables are present, an error is generated (SQLE_NO_INDICATOR: no indicator variable for NULL result).
- If a row being fetched has been updated, generating an SQLE_ROW_UPDATED_WARNING warning, the fetch stops on the row which caused the warning and the values for all rows processed to that point (including the row that caused the warning) are returned. SQLCOUNT contains the number of rows fetched and includes the row that caused the warning. All remaining SQLDA items are marked as NULL.
- If a row being fetched has been deleted or is locked, generating an SQLE_NO_CURRENT_ROW or SQLE_LOCKED error, SQLCOUNT contains the number of rows read prior to the error, not including the row that caused the error, but the SQLDA does not contain values for those rows, since SQLDA values are not returned on errors. The SQLCOUNT value can be used to reposition the cursor, if necessary, to read the rows.

The SQL descriptor area (SQLDA)

The SQLDA (SQL Descriptor Area) is an interface structure that is used for dynamic SQL statements. The structure passes information to and from the database regarding host variables and select statement results. The SQLDA is defined in the header file SQLDA.H.

There are functions in the database interface library or DLL that you can use to manage SQLDAs (see "SQLDA management functions" on page 709).

When host variables are used with static SQL statements, the preprocessor actually constructs an SQLDA for those host variables. It is this SQLDA that is actually passed to and from the database engine.

Structure packing option
All Embedded SQL programs must be compiled with the structure packing option of the compiler set to one-byte alignment (usually the default).

```
#define SQL_MAX_NAME_LEN 30
typedef short int a_sql_type;
struct sqlname {
   short int length;
   unsigned char data[ SQL_MAX_NAME_LEN ];
 };
struct sqlvar {
   short int sqltype;
   short int sqllen;
   void _sqldafar *sqldata;
   short int _sqldafar *sqlind;
   struct sqlname sqlname;
 };
struct sqlda{
   unsigned char sqldaid[8];
   long int sqldabc;
   short int sqln;
   short int sqld;
   struct sqlvar sqlvar[1];
};
#define SCALE(sqllen) ((sqllen)/256)
#define PRECISION(sqllen) ((sqllen)&0xff)
#define SET_PRECISION_SCALE(sqllen,precision,scale)
\
   sqllen = (scale)*256 + (precision)
typedef struct sqlda SQLDA;
typedef struct sqlvar SQLVAR, SQLDA_VARIABLE;
typedef struct sqlname SQLNAME, SQLDA_NAME;
#define SQLDASIZE(n) ( sizeof( struct sqlda ) + \
         (n-1) * sizeof( struct sqlvar) )
```

The SQLDA fields have the following meanings:

Field	Description
sqldaid	An 8 byte character field that contains the string "SQLDA " as an identification of the SQLDA structure. This field helps in debugging when looking at memory contents.
sqldabc	A long integer containing the length of the SQLDA structure (depends on the size of the array of *sqlvar*'s — 16 + 44*n bytes).
sqln	The number of variable descriptors in the sqlvar array (i.e. how much actual space there is).
sqld	The number of variable descriptors which are valid (i.e. contain information describing a host variable). This field is set by the DESCRIBE statement and sometimes by the programmer when supplying data to the database engine.
sqlvar	An array of descriptors of type struct sqlvar each describing a host variable.

The *sqlvar* fields have the following meanings:

Field	Description
sqltype	The type of the variable described by this descriptor (see "Embedded SQL interface data types" on page 664). The low order bit indicates whether NULL values are allowed. Valid types and constant definitions can be found in the SQLDEF.H header file (see "SQLDEF.H header file" on page 747). This field is filled by the DESCRIBE statement. You can set this field to any type when supplying data to the database engine or retrieving data from the database engine and any necessary type conversion will be done automatically.
sqllen	The length of the variable. What the length actually means depends upon the type information and how the SQLDA is being used. For decimal types, this field is divided into two 1-byte fields. The high byte is the precision and the low byte is the scale. The precision is the total number of digits and the scale is the number of digits that appear after the decimal point. For more discussion of the length field, see "Length field values" on page 692.
sqldata	A four-byte pointer to the memory occupied by this variable. The memory pointed to by the sqldata field must correspond to the sqltype and sqllen fields. See "Embedded SQL interface data types" on page 664 for storage formats. For update and insert commands, this variable will not be involved in the operation if the sqldata pointer is a null pointer. For a fetch, no data is returned if the sqldata pointer is the

Field	Description
	null pointer.
sqlind	A four-byte pointer to the indicator value. An indicator value is a short int. A negative indicator value indicates a NULL value. A positive indicator value indicates that this variable has been truncated by a fetch statement, and the indicator value contains the length of the data before truncation (see "Indicator variables" on page 671). If the sqlind pointer is the null pointer, then there is no indicator variable pertaining to this host variable. The sqlind field is also used by the DESCRIBE statement to indicate parameter types.
sqlname	A VARCHAR structure containing a length and character buffer. It is filled in on a DESCRIBE statement and is not otherwise used. This field has a different meaning for the two formats of the describe statement: On a DESCRIBE SELECT LIST command, indicator variables, if present, are filled with a flag indicating whether the select list item is updatable or not. More information on this flag can be found in the SQLDEF.H header file (see "SQLDEF.H header file" on page 747). ♦ **SELECT LIST** the name buffer is filled with the column heading of the corresponding item in the select list. ♦ **BIND VARIABLES** the name buffer is filled with the name of the host variable that was used as a bind variable, or "?" if an unnamed parameter marker is used.

Length field values

The **sqllen** field length of the SQLVAR struct in an SQLDA is used in 3 different kinds of interactions with the database engine. The following tables detail each of these interactions. These tables list the interface constant types (the **DT_** types) found in the SQLDEF.H header file. These constants would be placed in the SQLDA **sqltype** field. The types are described in "Embedded SQL interface data types" on page 664.

Note that in static SQL, an SQLDA is still used but it is generated and completely filled in by the SQL preprocessor. In this static case, the table gives the correspondence between the static C language host variable types and the interface constants.

DESCRIBE

The following table indicates the values of the sqllen and sqltype structure members returned by the describe command for the various database types (both select list and bind variable describe statements). In the case of a user-defined database data type, the base type is described.

Your program can use the types and lengths returned from a DESCRIBE, or you may use another type. The database engine will perform type conversions between any two types. The memory pointed to by the sqldata field must correspond to the sqltype and sqllen fields.

Database field type	Embedded SQL type returned	Length returned on describe
CHAR(n)	DT_FIXCHAR	n
VARCHAR(n)	DT_VARCHAR	n
BINARY(n)	DT_BINARY	n
SMALLINT	DT_SMALLINT	2
INT	DT_INT	4
TINYINT	DT_TINYINT	1
DECIMAL(p,s)	DT_DECIMAL	high byte of length field in SQLDA set to p and low byte set to s
REAL	DT_FLOAT	4
FLOAT	DT_FLOAT	4
DOUBLE	DT_DOUBLE	8
DATE	DT_DATE	length of longest formatted string
TIME	DT_TIME	length of longest formatted string
TIMESTAMP	DT_TIMESTAMP	length of longest formatted string
LONG VARCHAR	DT_VARCHAR	32767
LONG BINARY	DT_BINARY	32767

Supplying a value

The following table indicates how lengths of values are specified when supplying data to the database engine in the SQLDA.

Only the data types shown in the table are allowed in this case. Note that the DT_DATE, DT_TIME and DT_TIMESTAMP types are treated the same as DT_STRING when supplying information to the database; the value must be a NULL-terminated character string in an appropriate date format.

Embedded SQL Data Type	What program must do to set length when supplying data to the database
DT_STRING	length determined by terminating '\0'
DT_VARCHAR(n)	length taken from field in VARCHAR structure
DT_FIXCHAR(n)	length field in SQLDA determines length of string
DT_BINARY(n)	length taken from field in BINARY structure
DT_SMALLINT	-
DT_INT	-
DT_DECIMAL(p,s)	high byte of length field in SQLDA set to p and low byte set to s
DT_FLOAT	-
DT_DOUBLE	-
DT_DATE	length determined by terminating '\0'
DT_TIME	length determined by terminating '\0'
DT_TIMESTAMP	length determined by terminating '\0'
DT_TIMESTAMP_STRUCT	-
DT_VARIABLE	length determined by terminating '\0'

Retrieving a value

The following table indicates the values of the length field when retrieving data from the database using an SQLDA. The **sqllen** field is never modified when retrieving data.

Only the interface data types shown in the table are allowed in this case. Note again that the DT_DATE, DT_TIME and DT_TIMESTAMP data types are treated the same as DT_STRING when retrieving information from the database; the value will be formatted as a character string in the current date format.

Embedded SQL Data Type	What program must set length field (sqllen) to when receiving	How database returns length information results from the database after fetching a value
DT_STRING	length of buffer	'\0' at end of string
DT_VARCHAR(n)	maximum length of VARCHAR structure (n+2)	len field of VARCHAR structure set to actual length
DT_FIXCHAR(n)	length of buffer	padded with blanks to length of buffer
DT_BINARY(n)	maximum length of BINARY structure (n+2)	len field of BINARY structure set to actual length
DT_SMALLINT	-	-
DT_INT	-	-
DT_DECIMAL(p,s)	high byte set to p and low byte set to s	-
DT_FLOAT	-	-
DT_DOUBLE	-	-
DT_DATE	length of buffer	'\0' at end of string
DT_TIME	length of buffer	'\0' at end of string
DT_TIMESTAMP	length of buffer	'\0' at end of string
DT_TIMESTAMP_STRUCT	-	-

SQL procedures in Embedded SQL

Simple procedures

Database procedures can be both created and called from Embedded SQL. A CREATE PROCEDURE statement can be embedded just like any other DDL statement. A CALL statement can also be embedded or it can be prepared and executed. Here is a simple example of both creating and executing a stored procedure in embedded SQL:

```
EXEC SQL CREATE PROCEDURE pettycash( IN amount
DECIMAL(10,2) )
BEGIN

   UPDATE account
   SET balance = balance - amount
   WHERE name = 'bank';

   UPDATE account
   SET balance = balance + amount
   WHERE name = 'pettycash expense';

END;
EXEC SQL CALL pettycash( 10.72 );
```

If you wish to pass host variable values to a stored procedure or retrieve the output variables, you will have to prepare and execute the CALL statement. The next example illustrates the use of host variables. Note the use of both the USING and INTO clause on the execute statement.

```
EXEC SQL BEGIN DECLARE SECTION;
   double   hv_expense;
   double   hv_balance;
EXEC SQL END DECLARE SECTION;

EXEC SQL CREATE PROCEDURE pettycash(
         IN expense  DECIMAL(10,2),
         OUT endbalance DECIMAL(10,2) )
   BEGIN
      UPDATE account
      SET balance = balance - expense
      WHERE name = 'bank';

      UPDATE account
      SET balance = balance + expense
      WHERE name = 'pettycash expense';

      SET endbalance = ( SELECT balance FROM account
                         WHERE name = 'bank' );
   END;
```

```
EXEC SQL PREPARE S1 FROM 'CALL pettycash( ?, ? )';

EXEC SQL EXECUTE S1 USING :hv_expense INTO
:hv_balance;
```

Procedures with result sets

Database procedures can also contain SELECT statements. The procedure is declared using a RESULT clause to specify the number, name and types of the columns in the result set. Note that result set columns are different from output parameters. For procedures with result sets, the CALL statement can be used in place of a SELECT statement in the cursor declaration:

```
EXEC SQL BEGIN DECLARE SECTION;
   char  hv_name[100];
EXEC SQL END DECLARE SECTION;

EXEC SQL CREATE PROCEDURE female_employees()
   RESULT( name char(50) )
   BEGIN
      SELECT emp_fname || emp_lname FROM employee
      WHERE sex = 'f';
   END;

EXEC SQL PREPARE S1 FROM 'CALL female_employees()';

EXEC SQL DECLARE C1 CURSOR FOR S1;
EXEC SQL OPEN C1;
for(;;) {
   EXEC SQL FETCH C1 INTO :hv_name;
   if( SQLCODE != SQLE_NOERROR ) break;
   printf( "%s\\n", hv_name );
}
EXEC SQL CLOSE C1;
```

In this example, the procedure has been invoked with an OPEN statement rather than an EXECUTE statement. The OPEN statement causes the procedure to execute until it reaches a SELECT statement. At this point, C1 is a cursor for the SELECT statement within the database procedure. You can use all forms of the FETCH command (backward and forward scrolling) until you are finished with it. The CLOSE statement terminates execution of the procedure. Note that if there had been another statement following the SELECT in the procedure, it would not have been executed. In order to execute statements following a SELECT, use the RESUME cursor-name command. The RESUME command will either return the warning SQLE_PROCEDURE_COMPLETE or it will return SQLE_NOERROR indicating there is another cursor. The next example illustrates a two select procedure:

```
EXEC SQL CREATE PROCEDURE people()
RESULT( name char(50) )
BEGIN

   SELECT emp_fname || emp_lname
   FROM employee;

   SELECT fname || lname
   FROM customer;
END;

EXEC SQL PREPARE S1 FROM 'CALL female_employees()';

EXEC SQL DECLARE C1 CURSOR FOR S1;
EXEC SQL OPEN C1;
while( SQLCODE == SQLE_NOERROR ) {
   for(;;) {
      EXEC SQL FETCH C1 INTO :hv_name;
      if( SQLCODE != SQLE_NOERROR ) break;
      printf( "%s\\n", hv_name );
   }
   EXEC SQL RESUME C1;
}
EXEC SQL CLOSE C1;
```

Dynamic cursors for CALL statements

These examples have used static cursors. Full dynamic cursors can also be used for the CALL statement (see "Dynamic SELECT statement" on page 685).

The DESCRIBE statement works fully for procedure calls. A DESCRIBE OUTPUT will produce an SQLDA having a description for each of the result set columns. If the procedure does not have a result set, then the SQLDA will have a description for each INOUT or OUT parameter to the procedure. A DESCRIBE INPUT statement will produce an SQLDA having a description for each IN or INOUT parameter to the procedure.

DESCRIBE ALL is a new form of the DESCRIBE statement that will describe IN, INOUT, OUT and RESULT set parameters. DESCRIBE ALL 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). See "DESCRIBE statement" in the chapter "Watcom-SQL Statements" for a complete description of this functionality.

Multi-Threaded or Reentrant Code

You can use Embedded SQL statements in multi-threaded or reentrant code. However, if you use a single connection, you are restricted to one active request per connection. In a multi-threaded application, you should not use the same connection to the database on each thread unless you use a semaphore to control access.

There are no restrictions on using separate connections on each thread that wishes to use the database. The SQLCA is used by the runtime library to distinguish between the different thread contexts. Thus, each thread wishing to use the database must have it's own SQLCA. Any given database connection will only be accessible from one SQLCA.

When to use multiple SQLCAs

You can use the multiple SQLCA support in any of the supported Embedded SQL environments, but it is only required in reentrant code.

The following list details the environments where multiple SQLCAs must be used:

- **Multi-threaded OS/2, Windows NT or Windows 95 application** An OS/2 or Windows NT application can have multiple threads using embedded SQL. If both threads use the same SQLCA, a context switch can cause both threads to be using the SQLCA at the same time. Each thread must have its own SQLCA. This can also happen when you have a DLL that uses embedded SQL and is called by more than one thread in your application.
- **Windows DLL** A Windows DLL has only one data segment. While the database engine is processing a request from one application, it may yield to another application that makes a request to the database engine. If your DLL uses the global SQLCA, both applications will be using it at the same time. Each Windows application must have its own SQLCA.
- **Windows NT, Windows 95 or OS/2 DLL with one data segment** An OS/2 or Windows NT DLL can be created with only one data segment or one data segment for each application. If your DLL has only one data segment, you cannot use the global SQLCA for the same reason a Windows DLL cannot use the global SQLCA. Each application must have its own SQLCA.

Single SQLCA can handle multiple connections
You do not need to use multiple SQLCAs to connect to more than one database or have more than one connection to a single database.

Using multiple SQLCAs

To manage multiple SQLCAs in your application,

1 You must use the command line switch on the SQL preprocessor that generates reentrant code ("-r"). The reentrant code is a little larger and a little slower because statically initialized global variables cannot be used. However, these affects are minimal.

2 Each SQLCA used in your program must be initialized with a call to db_init and cleaned up at the end with a call to db_fini.

> ***Caution***
> *Failure to call db_fini for each db_init on NetWare can cause the database server to fail, and the NetWare file serve to fail.*

3 The embedded SQL statement SET SQLCA ("SET SQLCA statement" in the chapter "Watcom-SQL Statements") is used to tell the SQL preprocessor to use a different SQLCA for database requests. Usually, a statement such as: EXEC SQL SET SQLCA 'task_data->sqlca'; is used at the top of your program or in a header file to set the SQLCA reference to point at task specific data. This statement does not generate any code and thus has no performance impact. It changes the state within the preprocessor so that any reference to the SQLCA will use the given string.

Connection Management with Multiple SQLCAs

Each SQLCA can have one unnamed connection. Each SQLCA has an active or current connection (see "SET CONNECTION statement" in the chapter "Watcom-SQL Statements"). All operations on a given database connection must use the same SQLCA that was used when the connection was established.

> **Record locking**
> Operations on different connections are subject to the normal record locking mechanisms and may cause each other to block and possibly to deadlock. For information on locking, see the chapter "Using Transactions and Locks".

Library functions

The SQL preprocessor generates calls to functions found in the interface library or DLL. In addition to the calls generated by the SQL preprocessor, there are several routines that are provided for the user to make database operations easier to perform. Prototypes for these functions are included by the EXEC SQL INCLUDE SQLCA command.

Following is a detailed description of these various functions by category. The prototypes given are those used in DOS and QNX.

DLL entry points

The DLL entry points are the same except that the prototypes have a modifier appropriate for DLLs:

- **Windows:** FAR PASCAL
- **OS/2:** _System
- **Windows NT:** __stdcall

All pointers passed as parameters to the Windows DLL entry points or returned by these functions are far pointers.

Example

For example, the first prototype listed below is db_init. For Windows, it would be:

```
unsigned short FAR PASCAL db_init( struct sqlca far *sqlca );
```

Passing null pointers
Care should be taken passing the null pointer as a paramcter in Windows if your program is compiled in the small or medium memory models. You should use the **_sql_ptrchk_()** macro defined in SQLCA.H for any pointer parameter which is a variable that might contain the null pointer. This macro converts a null near pointer into a null far pointer.

The following categories of functions are listed:

- "Interface initialization functions" on page 702
- "Connection and engine management functions" on page 703
- "SQLDA management functions" on page 709
- "Backup functions" on page 711
- "Other functions" on page 714

In addition, this section contains information on the following topics:

- "Aborting a request" on page 715

- "Windows 3.x request management" on page 716
- "Multi-Threaded or Reentrant Code" on page 699
- "Memory allocation in DOS and QNX" on page 718
- "DOS interrupt processing and request management" on page 719

Interface initialization functions

This section lists the functions concerned with initializing and releasing the interface.

db_init function

Prototype

```
unsigned short db_init( struct sqlca *sqlca );
```

Description

Initializes the database interface library or DLL. This function must be called before any other library call is made and before any Embedded SQL command is executed. Resources required by the interface library for your program are allocated and initialized on this call. Use db_fini to free the resources at the end of your program. If there are any errors during processing, they will be returned in the SQLCA and 0 will be returned. If there are no errors, a non-zero value is returned and you can begin using Embedded SQL commands and functions.

In most cases, this function should be called only once (passing the address of the global sqlca variable defined in the SQLCA.H header file). If you are writing a DLL or an application that has multiple threads using embedded SQL, you need to call db_init once for each SQLCA being used (see "Multi-Threaded or Reentrant Code" on page 699).

> ***Caution***
> *Failure to call **db_fini** for each **db_init** on NetWare can cause the database server to fail, and the NetWare file serve to fail.*

db_fini function

Prototype

```
unsigned short db_fini( struct sqlca *sqlca );
```

Frees resources used by the database interface or DLL. You must not make any other library calls or execute any Embedded SQL commands after db_fini is called. Any errors during processing will be returned in the SQLCA. If there are any errors during processing, they will be returned in the SQLCA and 0 will be returned. If there are no errors, a non-zero value is returned.

In most cases, this function should be called only once (passing the address of the global **sqlca** variable defined in the SQLCA.H header file). If you are writing a DLL or an application that has multiple threads using embedded SQL, you need to call db_fini once for each SQLCA being used (see "Multi-Threaded or Reentrant Code" on page 699).

> ***Caution***
> *Failure to call **db_fini** for each **db_init** on NetWare can cause the database server to fail, and the NetWare file serve to fail.*

Connection and engine management functions

The following functions provide a means to start and stop the database engine or SQL Anywhere Client (DBCLIENT.EXE), start or stop a database on an existing database engine or network database server; and connect to or disconnect from a database.

All of these functions take a NULL-terminated string as the second argument. 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

Verbose keyword	Short form
AutoStop	AutoStop
Start	Start
Unconditional	UNC
DataSourceName	DSN

Each function uses a subset of these parameters, but every function will allow any parameter to be set. An example connection parameter string is:

```
"UID=dba;PWD=sql;DBF=c:\sqlany50\sademo.db"
```

The SQLCONNECT environment variable can be used to specify default values for unspecified parameters (see "Registry entries and environment variables" in the chapter "SQL Anywhere Components").

db_string_connect function

Prototype unsigned db_string_connect(struct sqlca * sqlca, char * parms);

Description Provides extra functionality beyond the Embedded SQL CONNECT command. This function will do the following steps:

- Start the database engine if there is not one running with the name EngineName (calls db_start_engine). The AutoStop parameter will determine if the engine will automatically stop when the last used database is shutdown.
- If the database named by DatabaseName or DatabaseFile is not currently running, send a request to the engine or network server to start a database using the DatabaseFile, DatabaseName, and DatabaseSwitches parameters. The AutoStop parameter will determine if the database automatically shuts down when the last connection to the database is disconnected.
- Send a connection request to the database engine or network server based on the Userid, Password, and ConnectionName parameters.

The return value will be true (non-zero) if a connection was successfully established and false (zero) otherwise. Error information for starting the engine, starting the database or connecting will be returned in the SQLCA.

db_string_disconnect function

Prototype unsigned db_string_disconnect(struct sqlca * sqlca, char * parms);

Description

Disconnects the connection identified by the ConnectionName parameter. All other parameters are ignored. If no ConnectionName parameter is specified in the string, the unnamed connection will be disconnected. This is equivalent to the Embedded SQL DISCONNECT command. The boolean return value will be true if a connection was successfully ended. Error information will be returned in the SQLCA.

This function will shut down the database if the database was started with the AutoStop=**yes** parameter and there are no other connections to the database. It will stop the engine if the engine was started with the AutoStop=**yes** parameter and there are no other databases running. It will stop the SQL Anywhere Client (DBCLIENT.EXE) if it was started with the AutoStop=**yes** parameter and there are no remaining connections through the SQL Anywhere Client.

db_start_engine function

Prototype

```
unsigned db_start_engine( struct sqlca * sqlca,
    char * parms );
```

Description

Start the database engine or SQL Anywhere Client if it is not running. This function carries out the following steps:

- Look for a local database engine that has a name that matches the EngineName parameter. If no EngineName is specified, look for the default local database engine (the first database engine started).
- Look for the SQL Anywhere Client.
- If DatabaseName is specified, look for a local database engine that has a name that matches the DatabaseName parameter.
- If DatabaseName is not specified and DatabaseFile is specified, look for a local database engine that matches the root of the file name. For example, if DatabaseFile is C:\SQLANY50\SADEMO.DB, then look for a local engine named sademo.
- If no matching local engine is found and the SQL Anywhere Client (DBCLIENT) is not running, start a database engine or SQL Anywhere Client using the Start parameter (or the default start command).

The AutoStop parameter determines if the engine automatically stops when the last database is shut down or if the SQL Anywhere Client automatically stops when its last connection is gone.

The boolean return value is true if a database engine or SQL Anywhere Client was either found or successfully started. Error information will be returned in the SQLCA.

The following call to db_start_engine starts the database engine and names it **sademo**, but does not load the database, despite the DBF connection parameter:

```
db_start_engine( &sqlca, "DBF=c:\sqlany50\sademo.db;
Start=DBENG50W" );
```

If you wish to start a database as well as the engine, you must include the database file in the START connection parameter:

```
db_start_engine( &sqlca,"ENG=eng_name;START=DBENG50W
c:\sqlany50\sademo.db" );
```

This call starts the engine, names it eng_name, and starts the sademo database on that engine.

db_start_database function

Prototype

unsigned db_start_database(struct sqlca * sqlca, char * parms);

Description

Start a database on an existing engine or network server if the database is not already running. This function will do the following steps:

- Look for a local database engine that has a name that matches the EngineName parameter. If no EngineName is specified, look for the default local database engine.
- Look for the SQL Anywhere Client.
- If no matching engine or SQL Anywhere Client is found, this function fails.
- Send a request to the engine or network server to start a database using the DatabaseFile, DatabaseName and DatabaseSwitches parameters. The AutoStop parameter will determine if the database automatically shuts down when the last connection is disconnected.

The boolean return value will be true if the database was already running or successfully started. Error information will be returned in the SQLCA.

db_stop_database function

Prototype

void db_stop_database(struct sqlca * sqlca,
    char * parms);

Description

Stop a database identified by DatabaseName on an engine or network server identified by EngineName. If EngineName is not specified, the default engine or network server will be used.

By default, this function will not stop a database that has existing connections. If Unconditional is *yes*, the database will be stopped regardless of existing connections.

db_stop_engine function

Prototype

```
void db_stop_engine( struct sqlca * sqlca,
    char * parms );
```

Description

Terminates execution of the database engine or SQL Anywhere Client (DBCLIENT.EXE). This function will do the following steps:

- Look for a local database engine that has a name that matches the EngineName parameter. If no EngineName is specified, look for the default local database engine.
- Look for the SQL Anywhere Client.
- If no matching engine or SQL Anywhere Client is found, this function fails.
- Send a request to the engine to tell it to checkpoint and shut down all databases.
- Unload the database engine or SQL Anywhere Client.

By default, this function will not stop a database engine or SQL Anywhere Client that has existing connections. If Unconditional is *yes*, the database engine or SQL Anywhere Client will be stopped regardless of existing connections.

This function can be used directly from a C program instead of spawning DBSTOP.

db_find_engine function

Prototype

```
unsigned short db_find_engine( struct sqlca *sqlca, char *name );
```

Description Returns an unsigned short value indicating status information about the database engine whose name is *name*. If neither an engine nor multi-user server can be found with the specified name, the return value will be 0. A non-zero value indicates that the engine or client is currently running. Each bit in the return value conveys some information. Constants representing the bits for the various pieces of information are defined in the SQLDEF.H header file (see "SQLDEF.H header file" on page 747). If a null pointer is specified for *name*, then information is returned about the default database environment.

db_build_parms function

Prototype struct a_dblib_info *db_build_parms(struct sqlca *sqlca, char *connectstr, char *startstr);

Description Historical. See db_string_connect.

db_destroy_parms function

Prototype void db_destroy_parms(struct sqlca *sqlca, a_dblib_info *parms);

Description Historical. See db_string_connect.

db_free_parms function

Prototype void db_free_parms(a_dblib_info *parms);

Description Historical. See db_string_connect.

db_parms_connect function

Prototype unsigned db_parms_connect(struct sqlca *sqlca, a_dblib_info *info);

Description Historical. See db_string_connect.

db_parms_disconnect function

Prototype void db_parms_disconnect(struct sqlca *sqlca, a_dblib_info *info);

Description Historical. See db_string_disconnect.

db_start function

Prototype unsigned db_start(struct sqlca *sqlca, a_dblib_info *info);

Description Historical. See db_start_engine and db_start_database.

db_stop function

Prototype void db_stop(struct sqlca *sqlca, char *name, short unconditional);

Description Historical. See db_stop_engine and db_stop_database.

SQLDA management functions

The following functions are used to manage SQL Descriptor Areas (SQLDAs). "The SQL descriptor area (SQLDA)" on page 690 describes the SQLDA structure in detail.

alloc_sqlda_noind function

Prototype struct sqlda *alloc_sqlda_noind(unsigned numvar);

Description Allocates an SQLDA with descriptors for *numvar* variables. The **sqln** field of the SQLDA will be initialized to *numvar*. Space is not allocated for indicator variables; the indicator pointers are set to the null pointer. A null pointer will be returned if memory cannot be allocated.

alloc_sqlda function

Prototype struct sqlda *alloc_sqlda(unsigned numvar);

Description Allocates an SQLDA with descriptors for *numvar* variables. The **sqln** field of the SQLDA will be initialized to *numvar*. Space is allocated for the indicator variables, the indicator pointers are set to point to this space, and the indicator value is initialized to zero. A null pointer will be returned if memory cannot be allocated.

fill_sqlda function

Prototype struct sqlda *fill_sqlda(struct sqlda *sqlda);

Description Allocates space for each variable described in each descriptor of *sqlda* and assigns the address of this memory to the **sqldata** field of the corresponding descriptor. Enough space is allocated for the database type and length indicated in the descriptor. Returns *sqlda* if successful and returns the null pointer if there is not enough memory available.

sqlda_string_length function

Prototype unsigned long sqlda_string_length(struct sqlda *sqlda, int varno);

Description Returns the length of a C string (type DT_STRING) required to hold the variable **sqlda->sqlvar[varno]** (no matter what its type is).

sqlda_storage function

Prototype unsigned long sqlda_storage(struct sqlda *sqlda, int varno);

Description Returns the amount of storage required to store any value for the variable described in **sqlda->sqlvar[varno]**.

fill_s_sqlda function

Prototype struct sqlda *fill_s_sqlda(struct sqlda *sqlda, unsigned int maxlen);

Description Much the same as fill_sqlda except that it changes all the data types in *sqlda* to type DT_STRING (see "SQLDEF.H header file" on page 747). Enough space is allocated for the strings to hold the string representation of the type originally specified by the SQLDA up to a maximum of *maxlen* bytes. The length fields in the SQLDA (**sqllen**) are modified appropriately. Returns *sqlda* if successful and returns the null pointer if there is not enough memory available.

free_filled_sqlda function

Prototype void free_filled_sqlda(struct sqlda *sqlda);

Description Free all space allocated to this *sqlda* including the memory allocated to each **sqldata** pointer. Any null pointer will not be freed. The indicator variable space is also freed as allocated in fill_sqlda.

free_sqlda_noind function

Prototype void free_sqlda_noind(struct sqlda *sqlda);

Description Free the space allocated to this *sqlda* but do not free the memory referenced by each **sqldata** pointer. The indicator variable pointers are ignored.

free_sqlda function

Prototype void free_sqlda(struct sqlda *sqlda);

Description Free the space allocated to this *sqlda* but do not free the memory referenced by each **sqldata** pointer. The indicator variable space is also freed as allocated in fill_sqlda.

Backup functions

The db_backup function provides support for online backup. The SQL Anywhere backup utility makes use of this function. You should only need to write a program to use this function if your backup requirements are not satisfied by the SQL Anywhere backup utility.

Every database contains one or more files. Normally, a database contains two files, the main database file and the transaction log. Each file is divided up into fixed size pages, and the size of these pages is specified when the database was created. Backup works by opening a file, and then making a copy of each page in the file. Backup performs a checkpoint on startup. The database file(s) is backed up as of this checkpoint. Any changes made while the backup is running are recorded in the transaction log and will be backed up with the transaction log. This is why you always backup the transaction log last.

Authorization You must be connected to a user ID with DBA authority or REMOTE DBA authority (SQL Remote) to use the backup functions.

db_backup function

Prototype

```
void db_backup( struct sqlca * sqlca, int op, int file_num, unsigned long
    page_num, struct sqlda * sqlda);
```

Authorization Must be connected to a user ID with DBA authority or REMOTE DBA authority (SQL Remote).

Description The action performed by the db_backup function depends on the value of the *op* parameter:

- **DB_BACKUP_START** Must be called before a backup can start. Only one backup can be running at one time against any given database engine. Database checkpoints are disabled until the backup is complete (db_backup is called with an *op* value of DB_BACKUP_END). If the backup cannot start, the SQLCODE will be SQLE_BACKUP_NOT_STARTED. Otherwise, the SQLCOUNT field of the *sqlca* will be set to the size of each database page. (Backups are processed one page at a time.)

  The *file_num*, *page_num* and *sqlda* parameters are ignored.

- **DB_BACKUP_OPEN_FILE** Open the database file specified by *file_num*, which allows pages of the specified file to be backed up using DB_BACKUP_READ_PAGE. Valid file numbers are 0 through DB_BACKUP_MAX_FILE for the main database files, DB_BACKUP_TRANS_LOG_FILE for the transaction log file, and DB_BACKUP_WRITE_FILE for the database write file if it exists. If the specified file does not exist, the SQLCODE will be SQLE_NOTFOUND. Otherwise, SQLCOUNT contains the number of pages in the file, SQLIOESTIMATE contains a 32 bit value (POSIX **time_t**) which identifies the time that the database file was created, and the operating system file name is in the *sqlerrmc* field of the SQLCA.

  The *page_num* and *sqlda* parameters are ignored.

- **DB_BACKUP_READ_PAGE** Read one page of the database file specified by *file_num*. The *page_num* should be a value from 0 to one less than the number of pages returned in SQLCOUNT by a successful call to db_backup with the DB_BACKUP_OPEN_FILE operation. Otherwise, SQLCODE will be set to SQLE_NOTFOUND. The *sqlda* descriptor should be set up with one variable of type DT_BINARY pointing to a buffer that is large enough to hold binary data of the size returned in SQLCOUNT on the call to db_backup with the DB_BACKUP_START operation. (Note that DT_BINARY data contains a two byte length followed by the actual binary data, so the buffer must be two bytes longer than the page size.)

  Although the DBBACKUP program reads the pages sequentially in the database file, the pages could be read in any order.

  Application must save buffer
  This call makes a copy of the specified database page into the buffer, but it is up to the application to save the buffer on some backup media.

- **DB_BACKUP_READ_RENAME_LOG** This action is the same as DB_BACKUP_READ_PAGE except that after the last page of the transaction log has been returned, the database engine will rename the transaction log and start a new one. If the database engine is unable to rename the log at the current time (there are incomplete transactions), you will get the SQLE_BACKUP_CANNOT_RENAME_LOG_YET error. In this case, don't use the page returned, but instead reissue the request until you receive SQLE_NOERROR and then write the page. Continue reading the pages until you receive the SQLE_NOTFOUND condition. Note that the SQLE_BACKUP_CANNOT_RENAME_LOG_YET error may be returned multiple times and on multiple pages. In your retry loop, you should add a delay so as not to slow the server down with too many requests.

  When you receive the SQLE_NOTFOUND condition, the transaction log has been backed up successfully and the file has been renamed. The name for the old transaction file will be returned in the *sqlerrmc* field of the SQLCA.

- **DB_BACKUP_CLOSE_FILE** Must be called when processing of one file is complete to close the database file specified by *file_num*.

  The *page_num* and *sqlda* parameters are ignored.

- **DB_BACKUP_END** Must be called at the end of the backup. No other backup can start until this backup has ended. Checkpoints are enabled again.

  The *file_num*, *page_num* and *sqlda* parameters are ignored.

The algorithm used by the DBBACKUP program is as follows. Note that this is NOT C code, and does not include error checking.

```
db_backup( ... DB_BACKUP_START ... )
allocate page buffer based on page size in SQLCODE
sqlda = alloc_sqlda( 1 )
sqlda->sqld = 1;
sqlda->sqlvar[0].sqltype = DT_BINARY
sqlda->sqlvar[0].sqldata = allocated buffer
for file_num = 0 to DB_BACKUP_MAX_FILE
db_backup( ... DB_BACKUP_OPEN_FILE, file_num ... )
if SQLCODE == SQLE_NO_ERROR
/* The file exists */
num_pages = SQLCOUNT
file_time = SQLE_IO_ESTIMATE
open backup file with name from sqlca.sqlerrmc
for page_num = 0 to num_pages - 1
db_backup( ... DB_BACKUP_READ_PAGE,
file_num, page_num, sqlda )
write page buffer out to backup file
```

```
next page_num
close backup file
db_backup( ... DB_BACKUP_CLOSE_FILE, file_num ... )
end if
next file_num
backup up file DB_BACKUP_WRITE_FILE as above
backup up file DB_BACKUP_TRANS_LOG_FILE as above
free page buffer
db_backup( ... DB_BACKUP_END ... )
```

db_delete_file function

Prototype

void db_delete_file(struct sqlca * sqlca,
    char * filename);

Authorization

Must be connected to a user ID with DBA authority or REMOTE DBA authority (SQL Remote).

Description

The db_delete_file function requests the database engine to delete *filename*. This can be used after backing up and renaming the transaction log (see DB_BACKUP_READ_RENAME_LOG above) to delete the old transaction log. Note that this request will work both with the single-user engine and the multi-user server. You must be connected to a userid with DBA authority.

Other functions

sql_needs_quotes function

Prototype

unsigned int sql_needs_quotes(struct sqlca *sqlca, char *str);

Description

Returns a boolean indicating whether or not the string *str* requires double quotes around it when it is used as a SQL identifier. This function actually formulates a request to the database engine to determine if quotes are needed.

sqlerror_message function

Prototype

char *sqlerror_message(struct sqlca *sqlca, char * buffer, int max);

Description

Return a pointer to a string containing an error message. The error message contains text for the error code in the SQLCA. If no error was indicated, a null pointer is returned. The error message will be placed in the buffer supplied truncated to length *max* if necessary.

Aborting a request

db_abort_request function

Prototype int db_abort_request(void);

Description Historical. See db_cancel_request.

db_cancel_request function

Prototype int db_cancel_request(struct sqlca *sqlca);

Description Abort the currently active database engine request. This function will check to make sure a database engine request is active before sending the abort request. The return value indicates whether or not an abort request was sent; in other words, whether or not a database request was active. A non-zero return value does not mean that the request was aborted. There are a few critical timing cases where the abort request and the response from the database or server "cross". In these cases, the abort simply has no effect.

Note that db_cancel_request can be called asynchronously (from an interrupt handler, a Window procedure in Windows, or from another OS/2 or Windows NT thread). This function and db_is_working are the only functions in the database interface library that can be called asynchronously using an SQLCA that might be in use by another request.

In DOS, the default CTRL+BREAK handling mechanism in the database interface library calls this routine when a CTRL+BREAK is detected.

db_is_working function

Prototype unsigned db_is_working(struct sqlca *sqlca);

Description Returns 1 if your application has a database request in progress that uses the given sqlca, and 0 if there is no request in progress that uses the given sqlca. This function can be called asynchronously (from an interrupt handler, a Window procedure in Windows, or from another OS/2 or Windows NT thread). This function and db_cancel_request are the only functions in the database interface library that can be called asynchronously using an SQLCA that might be in use by another request.

db_working function

Prototype unsigned db_working(void);

Description Historical. See db_is_working.

Windows 3.x request management

This section does not apply to Windows NT or Windows 95.

You may have noticed when using ISQL under Windows that the Windows system is still active while ISQL is waiting for a response from the database. This is not the default behavior of the interface DLL. Default behavior puts the hour glass cursor on the screen and forces the user to wait for completion of the request.

The proper way to provide this active behavior for an application would be through the use of asynchronous database requests. The current release of SQL Anywhere does not support asynchronous requests but you can achieve application activity in Windows while a database request is in progress by providing a callback function. The key difference is that in this callback function you must not do another database request (except db_cancel_request). You can use the db_is_working function in your message handlers to determine if you have a database request in progress.

This callback function in your application will be called repeatedly while the database engine or SQL Anywhere Client is busy processing a request. You can then process Windows messages by calling GetMessage or PeekMessage. (These function calls allow Windows to be active.) The DBL50 DLL will continually call this function until the response from the database engine is received.

Response in Windows message
The response from the engine or SQL Anywhere Client comes to your application via a Windows message. You must either dispatch this message (the interface DLL will receive the message), or you must call the db_process_message function with each message you receive while in this callback function. The function will return TRUE if the message was the response; otherwise you can process the message normally by dispatching it.

The following function is used to register your application callback functions:

db_register_a_callback function

Prototype

```
void db_register_a_callback( struct sqlca *sqlca, a_db_callback_index index,
    FARPROC callback );
```

Description

This function is used to register Windows callback functions. To remove a callback, pass a null pointer as the *callback* function. Following is a list of the allowable values for *index* and the appropriate function prototype. Note that you should call MakeProcInstance with your function address and pass that to the db_register_callback function.

If you don't register any callback functions, then the default action is to do nothing. Your application will block waiting for the database response and Windows will change the cursor to an hour glass.

The following values are allowed for the *index* parameter:

- **DB_CALLBACK_START** void FAR PASCAL db_start_request(struct sqlca *sqlca);

  This function is called just before a database request is sent to the engine or SQL Anywhere Client.

- **DB_CALLBACK_FINISH** void FAR PASCAL db_finish_request(struct sqlca *sqlca);

  This function is called after the response to a database request has been received by the interface DLL.

- **DB_CALLBACK_WAIT** void FAR PASCAL db_wait_request(struct sqlca *sqlca);

  This function is called repeatedly by the interface DLL while the database engine or SQL Anywhere Client is busy processing your database request.

  The following is a sample DB_CALLBACK_WAIT callback function:

```
void FAR PASCAL db_wait_request( struct sqlca *sqlca
)
{ MSG msg
   f( GetMessage( &msg, NULL, 0, 0 ) ) {
      ( !db_process_a_message( sqlca, &msg ) ) {
         if( !TranslateAccelerator( hWnd, hAccel,
&msg ) ) {
         TranslateMessage( &msg )
         DispatchMessage( &msg )
         }
      }
   }
}
```

db_process_a_message function

Prototype int db_process_a_message(struct sqlca *sqlca, MSG *msg);

Description This function is called from within your db_wait_request callback function to determine if the message you received from Windows was in fact the response to the active database request. The return value will be TRUE if *msg* is the response. Simply return from the callback function and the embedded SQL library DLL will process the response by returning to the call that generated the original request.

Multiple SQLCA management

db_set_sqlca function

Prototype void db_set_sqlca(struct sqlca *sqlca);

Description Historical (DOS and Windows). See "SET SQLCA statement" in the chapter "Watcom-SQL Statements".

db_get_sqlca function

Prototype struct sqlca *db_get_sqlca(void);

Description Historical (DOS and Windows). See "SET SQLCA statement" in the chapter "Watcom-SQL Statements".

Memory allocation in DOS and QNX

In DOS and QNX, all database library functions use the following functions to allocate and free memory. On all other operating systems, the interface DLL manages its own memory.

These functions are in a module by themselves and simply call the normal C memory functions: malloc, realloc and free. In DOS and QNX, you can use different memory management functions by redefining DBAlloc, DBRealloc and DBFree in your application code (all three must be redefined together). Refer to the documentation for your C compiler and linker to see how to force the new definitions of these functions to be used instead of the versions supplied in the database library. Usually, this is a matter of naming your replacement file before the library in the linker command file.

DBAlloc function

Prototype void *DBAlloc(unsigned int size);

Description Allocates a piece of memory *size* bytes long and returns a pointer to it. This function is called by the interface library in order to dynamically allocate memory. The default routine in the library calls malloc.

DBRealloc function

Prototype void *DBRealloc(void *ptr, unsigned int size);

Description Reallocates the piece of memory pointed to by *ptr* to be *size* bytes long and returns a pointer to it. The contents of the original piece are present in the new piece of memory. This function is called by the interface library in order to dynamically reallocate memory. The default routine in the library calls realloc.

DBFree function

Prototype void DBFree(void *ptr);

Description Frees the dynamic memory piece pointed to by *ptr*. This function is called by the interface library in order to dynamically free a piece of memory. The default routine in the library calls free.

DOS interrupt processing and request management

The database interface library contains default handling mechanisms for CTRL+BREAK and the CRITICAL DEVICE ERROR ("Abort,Retry,Ignore?" errors) in DOS. These handlers modify the default behavior as follows:

- **Ctrl+Break** DOS's default handling for the break key is to terminate the active program. The default database library handler modifies the behavior so that if the database engine is running when CTRL+BREAK is pressed, the active request will be terminated, however, no programs will be terminated.
- **critical device error** DOS normally prompts the user with "Abort,Retry,Ignore?" when a critical error is detected. The default handler automatically fails the device request. The database engine will detect the error and enter a state where it will not process any further requests. This prevents further problems from occurring. You should stop the database engine, correct any device problems and restart the database.

By default, the interrupt handlers are installed before each request to the database engine and deinstalled after each request is completed. Both the installation mechanism and the function performed by the interrupt handler can be modified. Be cautious though, noting the above mentioned DOS default behavior for CTRL+BREAK and critical errors. In particular, the default behavior for CTRL+BREAK is not appropriate; terminating the database engine in that way will cause your machine to hang up.

The functions db_sending_request and db_finished_request described below are called before and after a database engine request. If you wish to provide different handlers for the CTRL+BREAK or critical device error interrupts, you should replace these functions with ones of your own.

You may want to catch these interrupts once when your application starts up by providing your own handlers for the interrupts. This will eliminate the overhead of catching the interrupts on every database request. This is the way that ISQL handles the CTRL+BREAK and critical error interrupts.

db_sending_request function

Prototype void db_sending_request(void);

Description This function is called by the database library before a request is sent to the engine. The default action provided by the version of this routine in the interface library is:

```
db_catch_break( db_break_handler );
db_catch_critical( db_critical_handler );
```

To modify CTRL+BREAK or critical error handling, provide your own version of this function.

db_finished_request function

Prototype void db_finished_request(void);

Description This function is called by the database library after a request to the engine is completed. The default action provided by the version of this routine in the interface library is:

```
db_release_critical();
db_release_break();
```

To modify CTRL+BREAK or critical error handling, provide your own version of this function.

The following functions are used by db_sending_request() and db_finished_request() in catching and processing the CTRL+BREAK and critical device error interrupts. They can also be used by your replacements for these routines.

db_catch_break function

Prototype void db_catch_break(void (interrupt far *rtn)(void));

Description This function installs *rtn* as the break handling routine. It will save the previous break handler so that it can be restored by db_release_break().

db_release_break function

Prototype void db_release_break(void);

Description This function deinstalls the break handler previously installed by db_catch_break and reinstalls the old break handler. Note that if some other program has replaced the break handler since your call to db_catch_break, then that handler will get lost when the old break handler is reinstalled.

db_break_handler function

Prototype void interrupt db_break_handler(void);

Description This is the default break handler for the database interface library. It calls db_cancel_request to abort the active database request.

db_catch_critical function

Prototype void db_catch_critical(void (interrupt far *rtn)(void));

Description This function installs *rtn* as the critical device error handling routine. It will save the previous handler so that it can be restored by db_release_critical().

db_release_critical function

Prototype void db_release_critical(void);

Description This function uninstalls the critical device error handler previously installed by db_catch_critical and reinstalls the old handler. Note that if some other program has replaced the critical error handler since your call to db_catch_critical, then that handler will get lost when the old critical error handler is reinstalled.

function

Prototype

void interrupt db_critical_handler(void);

Description

This is the default critical device error handler for the database interface library. The action specified in the library routine is to move a 3 into the AH register and issue an IRET instruction. This causes DOS to report a failure to the task that issued the I/O request. Note that by default, this handler is installed and deinstalled around each request to the database engine. Thus the task that issued the I/O request must have been the database engine. The engine expects to receive the failure status and will recover appropriately.

Interface library DLL dynamic loading

The interface library DLL can be loaded dynamically without having to link against the import library for the DLL, using the ESQLDLL.C module in the SRC subdirectory of your installation directory. Using ESQLDLL.C is recommended as it is easier to use and more robust in its ability to locate the interface DLL.

❖ To use the dynamic loading feature:

1 Your program must call db_init_dll to load the DLL, and must call db_fini_dll to free the DLL. db_init_dll must be called before any function in the database interface, and no function in the interface can be called after db_fini_dll.

For more details on db_init_dll and db_fini_dll.

You must still call the db_init and db_fini library functions.

2 You must #include the ESQLDLL.H header file before the EXEC SQL INCLUDE SQLCA statement or #include <SQLCA.H> line in your embedded SQL program.

3 You must ensure that a SQL OS macro is defined when compiling ESQLDLL.C. The header file SQLCA.H, which is included by this file, attempts to determine the appropriate macro and defines it. However, certain combinations of platforms and compilers may cause this to fail. In this case, you must add a #define to the top of this file, or make the definition by using a compiler option.

Macro	Platforms
_SQL_OS_WINNT	Windows 95 and Windows NT
_SQL_OS_WINDOWS	Windows 3.x
_SQL_OS_OS232	32-bit OS/2

4 Compile ESQLDLL.C.

5 Instead of linking against the imports library, link the object module ESQLDLL.OBJ with your embedded SQL application objects.

Example

The following example illustrates how to use ESQLDLL.C and ESQLDLL.H.:

```
#include <stdio.h>
#include "esqldll.h"
```

```
EXEC SQL INCLUDE SQLCA;
#include "sqldef.h"
#include <windows.h>

EXEC SQL BEGIN DECLARE SECTION;
int             x;
a_sql_statement_number      stat1;
EXEC SQL END DECLARE SECTION;

#define TRUE 1
#define FALSE 0


void printSQLError( void )
{
   char     buffer[200];

   sqlerror_message( &sqlca, buffer, sizeof(buffer)
);
   #ifdef _SQL_OS_WINDOWS
   printf( "Error %ld -- %Fs\n", SQLCODE, buffer );
   #else
   printf( "Error %ld -- %s\n", SQLCODE, buffer );
   #endif
}

char *dllpaths[] = { "s:\\jasonhi\\",
          NULL };


#include <windows.h>

int main( void )
{

   struct sqlda _fd_ *  sqlda1;
   int          result;
   char     string[200];

   printf( "Initing DLL\n" );

   result = db_init_dll( dllpaths );
   switch( result ) {
   case ESQLDLL_OK:
      printf("OK\n");
      break;
      case ESQLDLL_DLL_NOT_FOUND:
      printf("DLL NOT FOUND\n");
      return( 10 );
   case ESQLDLL_WRONG_VERSION:
      printf("WRONG VERSION\n");
      return( 10 );
   }
   if( !db_init( &sqlca ) ) {
      printf("db_init failed.\n");
```

```
        db_fini_dll();
    return( 10 );
    }

    #ifdef _SQL_OS_WINNT
    result = db_string_connect( &sqlca,
"UID=dba;PWD=sql;DBF=d:\\sqlany50\\sample.db");

    #elif defined( _SQL_OS_WINDOWS )
    result = db_string_connect( &sqlca,
"UID=dba;PWD=sql;DBF=c:\\wsql50\\sample.db");

    #elif defined( _SQL_OS_OS232 )
    result = db_string_connect( &sqlca,
"UID=dba;PWD=sql;DBF=h:\\wsql50\\sample.db");

    #endif

    if( ! result ) {
        printf( "db_string_connect returned = %d\n",
result );
    printSQLError();
    }
    sqlda1 = alloc_descriptor( sqlcaptr, 20 );
    EXEC SQL PREPARE :stat1 FROM
        'select * from employee
        WHERE empnum = 80921';
    if( SQLCODE != 0 ) {
        printSQLError();
    }
    EXEC SQL DECLARE curs CURSOR FOR :stat1;
    if( SQLCODE != 0 ) {
        printSQLError();
    }
    EXEC SQL OPEN curs;
    if( SQLCODE != 0 ) {
        printSQLError();
    }
    EXEC SQL DESCRIBE :stat1 INTO sqlda1;
    if( SQLCODE != 0 ) {
        printSQLError();
    }
    sqlda1->sqlvar[1].sqltype = 460;
    fill_sqlda( sqlda1 );
    EXEC SQL FETCH FIRST curs INTO DESCRIPTOR sqlda1;
    if( SQLCODE != 0 ) {
        printSQLError();
    }
    printf( "name = %Fs\n", (char _fd_ *)sqlda1-
>sqlvar[1].sqldata );
    x = sqlda1->sqld;
    printf( "COUNT = %d\n", x );

    free_filled_sqlda( sqlda1 );
    db_string_disconnect( &sqlca, "" );
```

```
    db_fini( &sqlca );
    db_fini_dll();

    return( 0 );
}
```

Embedded SQL commands

EXEC SQL
ALL Embedded SQL statements must be preceded with EXEC SQL and end with a semicolon.

Embedded SQL commands are broken into two groups:

1 Standard SQL commands that are part of the SQL language. These commands are further broken into two groups, the data manipulation commands and the data definition commands. They are used by simply placing them in a C program enclosed with EXEC SQL and a semi-colon. CONNECT, DELETE, SELECT, SET and UPDATE have additional formats only available in Embedded SQL. The additional formats fall into the second category of Embedded SQL specific commands. All commands are described in detail in "Watcom-SQL Language Reference".

2 Data Manipulation Commands:

Statement	Description
CALL	Invoke a database procedure
CHECKPOINT	Cause the database engine to write all modified information
COMMIT	Commit changes to the database
DELETE	Delete rows from a database table
INSERT	Insert rows into database tables
PREPARE TO COMMIT	First phase of two-phase commit
RELEASE SAVEPOINT	Release a savepoint
ROLLBACK	Undo changes to database
ROLLBACK TO SAVEPOINT	Undo changes to database
SAVEPOINT	Establish a savepoint within a transaction
SELECT	Retrieve information from the database
UPDATE	Update rows in the database.

3 Data Definition Commands: :

Statement	Description
ALTER	Modify a database table definition

Statement	Description
COMMENT ON	Fill in remarks column for tables and columns
CREATE	Create database objects
DROP	Remove objects from the database
GRANT	Create userids, grant privileges to users
REVOKE	Delete userids, revoke privileges from users
SET OPTION	Set database options
VALIDATE TABLE	Check validity of table contents and all indexes on table

In addition to the standard SQL data manipulation and data definition commands, there are several SQL commands that are specific to Embedded SQL and can only be used in a C program. These Embedded SQL commands are also described in "Watcom-SQL Language Reference".

- **CLOSE** close a cursor
- **CONNECT** connect to the database
- **Declaration Section** declare host variables for database communication
- **DECLARE** declare a cursor
- **DELETE (positioned)** delete the row at the current position in a cursor
- **DESCRIBE** describe the host variables for a particular SQL statement
- **DISCONNECT** disconnect from database engine
- **DROP STATEMENT** free resources used by a prepared statement
- **EXECUTE** execute a particular SQL statement
- **EXPLAIN** explain the optimization strategy for a particular cursor
- **FETCH** fetch a row from a cursor
- **GET OPTION** get the setting for a particular database option
- **INCLUDE** include a file for SQL preprocessing
- **OPEN** open a cursor
- **PREPARE** prepare a particular SQL statement
- **PUT** insert a row into a cursor
- **SET CONNECTION** change active connection

- **SET OPTION** change a database option value
- **SET SQLCA** use an SQLCA other than the default global one
- **UPDATE (positioned)** update the row at the current location of a cursor
- **WHENEVER** specify actions to occur on errors in SQL statements

Database examples

Two embedded SQL examples are included with the SQL Anywhere installation. The static cursor Embedded SQL example, CUR.SQC, demonstrates the use of static SQL statements. The dynamic cursor Embedded SQL example, DCUR.SQC, demonstrates the use of dynamic SQL statements. In addition to these examples, you may find other programs and source files as part of the installation of SQL Anywhere which demonstrate features available for particular platforms.

This section contains information on the following topics:

- "Building the examples" on page 730
- "Running the example programs" on page 731
- "Static cursor example" on page 732
- "Dynamic cursor example" on page 738

Building the examples

All of the examples are installed as part of the SQL Anywhere installation. They are placed in the CXMP subdirectory of the SQL Anywhere installation directory (usually C:\SQLANY50). Under QNX, they are found in /usr/sqlany50/sample.

Along with the sample program is a batch file, MAKEALL.BAT, that can be used to compile the sample program for the various environments and compilers supported by SQL Anywhere. For OS/2 the command is MAKEALL.CMD. For QNX, use the shell script makeall. The format of the command is as follows:

```
makeall {Example} {Platform} {Compiler} [DOS
Extender]
```

The first parameter is the name of the example program that you want to compile. It will be one of:

- **cur** static cursor example
- **dcur** dynamic cursor example

The second parameter is the platform in which the program will be run. The platform can be one of the following:

- **DOS** compile for 16-bit DOS.

- **DOS286** compile for 16-bit protected mode using Pharlap/286 DOS Extender.
- **DOS32** compile for 32-bit Extended DOS. You must specify a DOS Extender.
- **WINDOWS** compile for 16-bit Windows.
- **WIN32** compile for 32-bit Windows using the Watcom C 32-bit Windows support.
- **WINNT** compile for Windows NT.
- **OS232** compile for 32-bit OS/2.
- **NETWARE** compile for Netware NLM.
- **QNX** compile for 16-bit or 32-bit QNX.

The third parameter is the compiler to use to compile the program. The compiler can be one of:

- **WC** use Watcom C or Watcom C/32
- **MC** use Microsoft C
- **BC** use Borland C
- **CS** use IBM C Set++

The fourth parameter is optional, and specifies the DOS Extender being used to run the program. This option only applies when the platform is DOS286 or DOS32. The DOS Extender can be one of:

- **4G** Rational DOS/4GW
- **P3** Pharlap/386
- **P2** Pharlap/286

Running the example programs

All example programs present a console-type user interface where it prompts you for a command. The various commands manipulate a database cursor and print the query results on the screen. Simply type the letter of the command you wish to perform. Some systems may require you to press ENTER after the letter. The commands are similar to the following depending on which program you run:

- **p** print current page

- **u** move up a page
- **d** move down a page
- **b** move to bottom page
- **t** move to top page
- **i** insert a row (DCUR only)
- **n** new table (DCUR only)
- **q** quit
- **h** help (this list of commands)

Windows and Windows NT examples

The Windows versions of the example programs are real Windows programs. However, to keep the user interface code relatively simple, some simplifications have been made. In particular, these applications attempt to mimic the console type interface of the DOS examples. As such, they do not repaint their Windows on WM_PAINT messages except to reprint the prompt.

To run these programs, execute them by selecting Run from the Program Manager File menu and typing:

```
c:\sqlany50\cxmp\curwin.exe
```

or create a Program Manager icon for them.

Static cursor example

This example demonstrates the use of cursors. The particular cursor used here retrieves certain information from the employee table in the sample database. The cursor is declared statically, meaning that the actual SQL statement to retrieve the information is "hard coded" into the source program. This is a good starting point for learning how cursors work. The next example ("Dynamic cursor example" on page 738) takes this first example and converts it to use dynamic SQL statements.

See "Database examples" on page 730 for where the source code can be found and how to build this example program.

The C program with the Embedded SQL is shown below. The program looks much like a standard C program except there are Embedded SQL instructions that begin with EXEC SQL. In order to reduce the amount of code duplicated by the **cur** and **dcur** example programs (and the **odbc** example), the mainlines and the data printing functions have been placed into a separate file. This is MAINCH.C for character mode systems, and MAINWIN.C for Windows and Windows NT. The example programs each supply the following three routines which are called from the mainlines.

- **WSQLEX_Init** connect to the database and open the cursor
- **WSQLEX_Process_Command** process commands from the user, manipulating the cursor as necessary.
- **WSQLEX_Finish** close the cursor and disconnect from the database.

The function of the mainline is to:

1 call the WSQLEX_Init routine

2 loop getting commands from the user and calling WSQL_Process_Command until the user quits

3 call the WSQLEX_Finish routine

Connecting to the database is accomplished with the Embedded SQL CONNECT command supplying the appropriate userid and password.

The open_cursor routine both declares a cursor for the specific SQL command and also opens the cursor.

Printing a page of information is accomplished by the print routine. It loops PageSize times fetching a single row from the cursor and printing it out. Note that the fetch routine checks for warning conditions (end of cursor) and prints appropriate messages when they arise. Also, the SQL cursor is repositioned by this program to the row before the one that is displayed at the top of the current page of data.

The move, top and bottom routines use the appropriate format of the FETCH statement to position the cursor. Note that this form of the FETCH statement doesn't actually get the data — it only positions the cursor. Also, a general relative positioning routine move has been implemented to move in either direction depending on the sign of the parameter.

When the user quits, the cursor is closed and the database connection is also released. In this case, this is accomplished with the ROLLBACK WORK statement followed by a DISCONNECT.

```
/* CUR.SQC      General SQL code for Cursex example
(all platforms)
*/
```

```
#include <stdio.h>
#include <ctype.h>
EXEC SQL INCLUDE SQLCA;
#include "sqldef.h"
#include "example.h"

extern int              PageSize;

typedef struct a_student {
    unsigned long       studnum;
    char                name[ 41 ];
    char                sex[ 2 ];       /* M or F
plus null char */
    char                birthdate[ 15 ];
} a_student;

static void printSQLError()
{
    char                buffer[ 200 ];

    Displaytext( 0, "SQL error -- %s\n",
        sqlerror_message( &sqlca, buffer, sizeof(
buffer ) ) );
}

static int warning( char *msg )
{
    if( SQLCODE == SQLE_NOTFOUND ) {
        Displaytext( 2, "Not found - past count %ld
-- %s\n",

SQLCOUNT, msg );
    } else {
        Displaytext( 2, "Unexpected warning %ld --
%s\n",

SQLCODE, msg );
    }
    return( TRUE );
}

EXEC SQL WHENEVER SQLERROR { printSQLError();
return( FALSE ); };

static int connect()
{
    EXEC SQL CONNECT "DBA" IDENTIFIED BY "SQL";
    return( TRUE );
    /* errors will return FALSE: - see WHENEVER
above */
}

static int release()
{
```

```
    EXEC SQL ROLLBACK WORK;
    EXEC SQL DISCONNECT;
    db_fini( &sqlca );
    return( TRUE );
}

static int open_cursor()
{
    EXEC SQL DECLARE C1 CURSOR FOR
        SELECT studnum, initials || ' ' || surname,
sex, birthdate
        FROM admin.student;
    EXEC SQL OPEN C1;
    return( TRUE );
}

static int close_cursor()
{
    EXEC SQL CLOSE C1;
    return( TRUE );
}

static int fetch_row(
    EXEC SQL BEGIN DECLARE SECTION;
    unsigned long       *studnum,
    char                *name,
    char                *sex,
    char                *birthdate
    EXEC SQL END DECLARE SECTION;
    )
{
    EXEC SQL FETCH RELATIVE 1 C1
             INTO :studnum, :name, :sex, :birthdate;

    if( SQLCODE ) {
        warning( "Fetching" );
        return( FALSE );
    } else {
        return( TRUE );
    }
}

static int move(
    EXEC SQL BEGIN DECLARE SECTION;
    int                 relpos
    EXEC SQL END DECLARE SECTION;
    )
{
    EXEC SQL FETCH RELATIVE :relpos C1;
    return( TRUE );
}

static int top()
{
    EXEC SQL FETCH ABSOLUTE 0 C1;
```

```
    return( TRUE );
}

static int bottom()
{
    EXEC SQL FETCH ABSOLUTE -1 C1;
    return( TRUE );
}

static void help()
{
    Displaytext( 0, "Cursex Demonstration Program
Commands:\n" );
    Displaytext( 0, "p - Print current page\n" );
    Displaytext( 0, "u - Move up a page\n" );
    Displaytext( 0, "d - Move down a page\n" );
    Displaytext( 0, "b - Move to bottom page\n" );
    Displaytext( 0, "t - Move to top page\n" );
    Displaytext( 0, "q - Quit\n" );
    Displaytext( 0, "h - Help (this screen)\n" );
}

static void print()
{
    a_student           s;
    int                 i;
    int                 status;

    for( i = 0; i < PageSize; ) {
        ++i;
        status = fetch_row( &s.studnum, s.name,
s.sex, s.birthdate );
        if( status ) {
            Displaytext( 0, "%6ld", s.studnum );
            Displaytext( 10, "%-30.30s", s.name );
            Displaytext( 30, "%-3.3s", s.sex );
            Displaytext( 40, "%-15.15s\n",
s.birthdate );
        } else {
            break;
        }
    }
    move( -i );
}

extern int WSQLEX_Init()
{
    if( !db_init( &sqlca ) ) {
        Display_systemerror(
            "Unable to initialize database
interface\n" );
        return( FALSE );
    }
    if( !db_find_engine( &sqlca, NULL ) ) {
```

```
        Display_systemerror( "Database
Engine/Station not running" );
        return( FALSE );
    }
    if( !connect() ) {
        Display_systemerror( "Could not connect" );
        return( FALSE );
    }
    open_cursor();
    help();
    return( TRUE );
}

extern void WSQLEX_Process_Command( int selection )
{
    switch( tolower( selection ) ) {
        case 'p':       print();
                        break;

        case 'u':       move( -PageSize );
                        print();
                        break;

        case 'd':       move( PageSize );
                        print();
                        break;

        case 't':       top();
                        print();
                        break;

        case 'b':       bottom();
                        move( -PageSize );
                        print();
                        break;

        case 'h':       help();
                        break;

        default:        Displaytext( 0, "Invalid
command, press 'h' for help\n" );
    }
}

extern int WSQLEX_Finish()
{
    close_cursor();
    release();
    return( TRUE );
}
```

Dynamic cursor example

This example demonstrates the use of cursors for a dynamic SQL SELECT statement. It is a slight modification of the previous example. If you have not yet looked at "Static cursor example" on page 732 it would be helpful to do so before looking at this example.

See "Database examples" on page 730 for where the source code can be found and how to build this example program.

The **dcur** program allows the user to specify the table he wishes to look at with the 'n' command. The program then presents as much information from that table as will fit on the screen. The SQL SELECT statement is built up in a program array using the C library function rintf.

When this program is run, it prompts for a connection string of the form:

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

The C program with the Embedded SQL is shown below. The program looks much like the previous example with the exception of the connect, open_cursor and print functions.

The connect function uses the Embedded SQL interface functions db_string_connect to connect to the database. This function provides the extra functionality to support the connection string used to connect to the database.

The open_cursor routine first builds the select statement:

```
SELECT * FROM tablename
```

where *tablename* is a parameter passed in to the routine. It then prepares a dynamic SQL statement using this string.

The Embedded SQL DESCRIBE command is used to fill in the SQLDA structure for the results of the select statement.

Size of the SQLDA
An initial guess is taken for the size of the SQLDA (3) — if this is not big enough, then the actual size of the select list returned by the database engine is used to allocate an SQLDA of the right size. The SQLDA structure is then filled with buffers to hold strings representing the results of the query. Note that the fill_s_sqlda routine converts all data types in the SQLDA to DT_STRING (see "SQLDEF.H header file" on page 747) and allocates buffers of the appropriate size.

A cursor is then declared and opened for this statement. The rest of the routines for moving and closing the cursor remain the same.

The fetch routine is slightly different: it puts the results into the SQLDA structure instead of into a list of host variables. The print routine has changed significantly to print results from the SQLDA structure up to the width of the screen. The print routine also uses the name fields of the SQLDA to print headings for each column.

```
/* DCUR.SQC     General SQL code for Dcursex example
(all platforms)
*/

#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <stdlib.h>
EXEC SQL INCLUDE SQLCA;
EXEC SQL INCLUDE SQLDA;
#include "sqldef.h"
#include "example.h"

#define MAX_COL_WIDTH   50

extern int              PageSize;
char                    TableName[MAX_TABLE_NAME];
SQLDA                   _fd_ *SqlDA;

#ifdef _SQL_OS_NETWARE
    #define Stringncopy(x,y,z)      strncpy(x,y,z)
    #define Stringcat(x,y)          strcat(x,y)
#else
    #ifdef __SMALLDATA__
        #define Stringncopy(x,y,z)
_fstrncpy(x,y,z)
        #define Stringcat(x,y)        _fstrcat(x,y)
    #else
        #define Stringncopy(x,y,z)    strncpy(x,y,z)
        #define Stringcat(x,y)        strcat(x,y)
    #endif
#endif

EXEC SQL BEGIN DECLARE SECTION;
char                    CursName[ 20 ];
EXEC SQL END DECLARE SECTION;

static void printSQLError()
{
    char                buffer[ 200 ];

    Displaytext( 0, "SQL error -- %s\n",
        sqlerror_message( &sqlca, buffer, sizeof(
buffer ) ) );
}

static int warning( char *msg )
{
```

```
    if( SQLCODE == SQLE_NOTFOUND ) {
        if( SQLCOUNT >= 0 ) {
            Displaytext( 0, "Not found - past bottom
of table\n" );
        } else {
            Displaytext( 0, "Not found - past top of
table\n" );
        }
    } else {
        Displaytext( 0,
            "Unexpected warning %ld -- %s\n",
SQLCODE, msg );
    }
    return( TRUE );
}

EXEC SQL WHENEVER SQLERROR { printSQLError();
return( FALSE ); };

static int open_cursor()
{
    EXEC SQL BEGIN DECLARE SECTION;
    char                    buff[ 100 ];
    a_sql_statement_number  stat;
    EXEC SQL END DECLARE SECTION;
    int                 n;

    sprintf( buff, "select * from %s", TableName );

    /* Note that database fills in statement number
on prepare */
    EXEC SQL PREPARE :stat FROM :buff;

    /* Note that we must initialize the cursor name
*/
    strcpy( CursName, "table_cursor" );
    EXEC SQL DECLARE :CursName CURSOR FOR :stat;

    EXEC SQL OPEN :CursName;

    SqlDA = alloc_sqlda( 3 );
    EXEC SQL DESCRIBE :stat INTO SqlDA;
    if( SqlDA->sqld > SqlDA->sqln ) {
        n = SqlDA->sqld;
        free_sqlda( SqlDA );
        SqlDA = alloc_sqlda( n );
        EXEC SQL DESCRIBE :stat INTO SqlDA;
    }
    fill_s_sqlda( SqlDA, 1000 );
    return( TRUE );
}

static int close_cursor()
{
    EXEC SQL CLOSE :CursName;
```

```
    free_filled_sqlda( SqlDA );
    SqlDA = NULL;
    return( TRUE );
}

static int fetch_row()
{
    EXEC SQL FETCH RELATIVE 1 :CursName USING
DESCRIPTOR SqlDA;

    if( SQLCODE < 0  ||  SQLCODE == SQLE_NOTFOUND )
{
        warning( "Fetching" );
        return( FALSE );
    } else if( SQLCODE != 0 ) {
        warning( "Fetching" );
        return( TRUE );
    } else {
        return( TRUE );
    }
}

static int move(
    EXEC SQL BEGIN DECLARE SECTION;
    int                 relpos
    EXEC SQL END DECLARE SECTION;
    )
{
    EXEC SQL FETCH RELATIVE :relpos :CursName;
    if( SQLCODE == SQLE_NOTFOUND && SQLCOUNT == 0 )
{
    } else if( SQLCODE ) {
        warning( "Moving" );
        return( FALSE );
    }
    return( TRUE );
}

static int top()
{
    EXEC SQL FETCH ABSOLUTE 0 :CursName;
    return( TRUE );
}

static int bottom()
{
    EXEC SQL FETCH ABSOLUTE -1 :CursName;
    return( TRUE );
}

static void help()
{
    Displaytext( 0, "DCursex Demonstration Program
Commands:\n" );
    Displaytext( 0, "p - Print current page\n" );
```

```
    Displaytext( 0, "u - Move up a page\n" );
    Displaytext( 0, "d - Move down a page\n" );
    Displaytext( 0, "b - Move to bottom page\n" );
    Displaytext( 0, "t - Move to top page\n" );
    Displaytext( 0, "i - Insert a new row\n" );
    Displaytext( 0, "n - New table\n" );
    Displaytext( 0, "q - Quit\n" );
    Displaytext( 0, "h - Help (this screen)\n" );
}

static int col_width( SQLDA _fd_ *da, int col )
{
    int                 col_name_len;
    int                 data_len;
    SQLDA_VARIABLE      _fd_ *sqlvar;

    sqlvar = &da->sqlvar[ col ];
    col_name_len = sqlvar->sqlname.length;
    data_len = sqlvar->sqllen;
    if( data_len > col_name_len ) {
        col_name_len = data_len;
    }
    if( strlen( NULL_TEXT ) > col_name_len ) {
        col_name_len = strlen( NULL_TEXT );
    }
    if( col_name_len > MAX_COL_WIDTH ) {
        return( MAX_COL_WIDTH );
    }
    return( col_name_len );
}

static void print_headings( SQLDA _fd_ *da )
{
    int                 i;
    int                 width;
    int                 total;
    char                colname[ SQL_MAX_NAME_LEN +
1 ];
    char _fd_           *sqlname;

    total = 0;
    for( i = 0; i < da->sqld; ++i ) {
        width = col_width( da, i );
        sqlname = da->sqlvar[ i ].sqlname.data;
        Stringncopy( colname, sqlname,
            da->sqlvar[ i ].sqlname.length );
        colname[ da->sqlvar[ i ].sqlname.length ] =
'\0';
        Displaytext( total, "%-*.*s", width, width,
colname );
        total += width+1;
    }
    Displaytext( 0, "\n" );
}
```

```
static void print_data( SQLDA _fd_ *da )
{
    int                 i;
    int                 width;
    int                 total;
    SQLDA_VARIABLE _fd_ *sqlvar;
    char _sqldafar      *data;

    total = 0;
    for( i = 0; i < da->sqld; ++i ) {
        width = col_width( da, i );
        sqlvar = &da->sqlvar[ i ];
        if( *( sqlvar->sqlind ) < 0 ) {
            data = NULL_TEXT;
        } else {
            data = (char _sqldafar *)sqlvar-
>sqldata;
        }
        #if _sqlfar_isfar
            Displaytext( total, "%-*.*Fs", width,
width,
                (char far *)data );
        #else
            Displaytext( total, "%-*.*s", width,
width, data );
        #endif
        total += width+1;
    }
    Displaytext( 0, "\n" );
}

static void print()
{
    int                 i;

    if( SqlDA == NULL ) {
        Displaytext( 0, "*** Error: Cursor not
open\n" );
        return;
    }
    print_headings( SqlDA );
    for( i = 0; i < PageSize; ) {
        ++i;
        if( fetch_row() ) {
            print_data( SqlDA );
        } else {
            break;
        }
    }
    move( -i );
}

static int insert()
{
    char        prompt[ 80 ];
```

```
    char *      tempptr;
    int         templen;
    int         i;

    for( i = 0; i < SqlDA->sqld; i++ ) {
        strcpy( prompt, "Enter a value for '" );
        Stringcat( prompt, SqlDA-
>sqlvar[i].sqlname.data);
        strcat( prompt, "'" );
        tempptr = (char *) SqlDA->sqlvar[i].sqldata;
        templen = SqlDA->sqlvar[i].sqllen;
        Getvalue( prompt, tempptr, templen );
        if( strlen( (char *)SqlDA->sqlvar[i].sqldata
) == 0
            && (SqlDA->sqlvar[i].sqltype &
DT_NULLS_ALLOWED) != 0 ) {
            *SqlDA->sqlvar[i].sqlind = -1;
        } else {
            *SqlDA->sqlvar[i].sqlind = 0;
        }
    }
    EXEC SQL PUT :CursName USING DESCRIPTOR SqlDA;
    return( TRUE );
}

extern int WSQLEX_Init()
{
    char                parmstr[ 251 ];

    if( !db_init( &sqlca ) ) {
        Display_systemerror(
            "Unable to initialize database
interface\n" );
        return( FALSE );
    }
    Getvalue( "Enter connection string", parmstr,
250 );
    if( strlen( parmstr ) == 0 ) {
        strcpy( parmstr,

"UID=dba;PWD=sql;DBF=c:\\wsql\sample.db;ENG=sample"
);
    }
    db_string_connect( &sqlca, parmstr );
    if( SQLCODE != SQLE_NOERROR ) {
        printSQLError();
        db_fini( &sqlca );
        return( FALSE );
    }
    Getvalue( "Enter table name", TableName,
MAX_TABLE_NAME );
    open_cursor();
    help();
    return( TRUE );
}
```

```
extern void WSQLEX_Process_Command( int selection )
{
    switch( tolower( selection ) ) {
        case 'p':       print();
                        break;

        case 'u':       move( -PageSize );
                        print();
                        break;

        case 'd':       move( PageSize );
                        print();
                        break;

        case 't':       top();
                        print();
                        break;

        case 'b':       bottom();
                        move( -PageSize );
                        print();
                        break;

        case 'h':       help();
                        break;

        case 'n':       close_cursor();
                        Getvalue( "Enter table
name",
                            TableName,
MAX_TABLE_NAME );
                        open_cursor();
                        break;

        case 'i':       insert();
                        break;

        default:        Displaytext( 0,
                            "Invalid command, press
'h' for help\n" );
    }
}

extern int WSQLEX_Finish()
{
    close_cursor();
    EXEC SQL ROLLBACK WORK;
    EXEC SQL DISCONNECT;
    db_fini( &sqlca );
    return( TRUE );
}
```

NT Service examples

The example programs CUR.SQC and DCUR.SQC, when compiled for Windows NT, run optionally as services.

The two files containing the example code for NT services are the source file NTSVC.C and the header file NTSVC.H. The code allows the linked executable to be run as either a regular executable or as an NT service.

❖ **To run one of the compiled examples as an NT service:**

1. Start the SQL Anywhere Service Manager.
2. Click New. The Service Type window is displayed.
3. Click Sample Application, and click OK. The Edit SQL Anywhere Service Configuration window is displayed.
4. Enter a service name.
5. Click Path, and select the sample program (CURWNT.EXE or DCURWNT.EXE) from the CXMP subdirectory of the installation directory.
6. Click OK to install the service.
7. Click Start on the main window to start the service.

When run as a service, the programs display the normal user interface if possible. They also write the output to the Application Event Log. If it is not possible to start the user interface, the programs print one page of data to the Application Event Log and stop.

These examples have been tested with the Watcom C/C++ 10.5 compiler and the Microsoft Visual C++ 2.0 compiler.

SQLDEF.H header file

```
/*************************************************
**
Copyright (C) 1988-1993, by WATCOM International
Corp.
All rights reserved. No part of this software may be
reproduced in any form or by any means - graphic,
electronic or mechanical, including photocopying,
recording, taping or information storage and
retrieval
systems - except with the written permission of
WATCOM International Corp.
*************************************************
*/

#ifndef II_SQLDEF
#define II_SQLDEF

#include "sqlca.h"
#include "sqlda.h"
#include "sqlerr.h"
#include "sqlstate.h"


/*****************************************/
/* general definition of database constants */
/*****************************************/

#define DB_MAX_IDENTIFIER_LEN   128


/*************************************************/
/* definitions for connection parms to dblib      */
/*    - used in functions: db_build_parms         */
/*                         db_free_parms          */
/*                         db_start               */
/*                         db_parm_connect        */
/*************************************************/

typedef struct a_dblib_info {
    char                _sqlfar *userid;
    char                _sqlfar *password;
    char                _sqlfar *database_name;
    char                _sqlfar *connection_name;
    char                _sqlfar *dbstartcommline;
} a_dblib_info;


/*************************************************/
/* constants for return values of DOS executables */
/*************************************************/

typedef short int an_exit_code;
```

```
#define EXIT_OKAY                    0   /*
everything is okay */
#define EXIT_FAIL                    1   /* general
failure */
#define EXIT_BAD_DATA                2   /* invalid
file format, etc.*/
#define EXIT_FILE_ERROR              3   /* file not
found, */
                                         /* unable to
open, etc. */
#define EXIT_OUT_OF_MEMORY           4   /* out of
memory */
#define EXIT_BREAK                   5   /*
terminated by user */
#define EXIT_COMMUNICATIONS_FAIL     6   /* failed
communications */
#define EXIT_MISSING_DATABASE        7   /* missing
required db name */
#define EXIT_PROTOCOL_MISMATCH       8   /*
client/server protocol */
                                         /* mismatch
*/
#define EXIT_UNABLE_TO_CONNECT       9   /* unable to
connect to db */
#define EXIT_ENGINE_NOT_RUNNING      10  /* database
not running */
#define EXIT_SERVER_NOT_FOUND        11  /* server
not running */
#define EXIT_TIMEOUT                 254 /* stop time
reached */
#define EXIT_USAGE                   255 /* invalid
command line */


/***********************************************/
/* constants for return value of db_find_engine */
/***********************************************/

#define DB_ENGINE                0x0001
#define DB_CLIENT                0x0002
#define DB_CAN_MULTI_DB_NAME     0x0004
#define DB_DATABASE_SPECIFIED    0x0008
#define DB_ACTIVE_CONNECTION     0x0010
#define DB_CONNECTION_DIRTY      0x0020
#define DB_CAN_MULTI_CONNECT     0x0040
#define DB_NO_DATABASES          0x0080

/*********************************/
/* host language type definitions */
/*********************************/

typedef unsigned short  a_sql_data_type;

#define DT_TYPES        0x03fe
```

```
#define DT_FLAGS        0xfc01

#define DT_NULLS_ALLOWED        0x0001

/* The following flags are used in the */
/* indicator variable on a describe */

#define DT_UPDATABLE            0x2000

#define DT_DESCRIBE_INPUT       0x1000

#define DT_PROCEDURE_IN         0x4000
#define DT_PROCEDURE_OUT        0x8000


#define DT_NOTYPE       0
#define DT_SMALLINT     500
#define DT_INT          496
#define DT_DECIMAL      484
#define DT_FLOAT        482
#define DT_DOUBLE       480
#define DT_DATE         384
#define DT_STRING       460
#define DT_FIXCHAR      452
#define DT_VARCHAR      448
#define DT_LONGVARCHAR  456
#define DT_TIME         388
#define DT_TIMESTAMP    392
#define DT_TIMESTAMP_STRUCT 390
#define DT_BINARY       524
#define DT_LONGBINARY   528
#define DT_VARIABLE     600

/**************************/
/* Backup operation types */
/**************************/

#define DB_BACKUP_START             1
#define DB_BACKUP_OPEN_FILE         2
#define DB_BACKUP_READ_PAGE         3
#define DB_BACKUP_CLOSE_FILE        4
#define DB_BACKUP_END               5
#define DB_BACKUP_READ_RENAME_LOG   6

/**************************/
/* Backup file constants  */
/**************************/

#define DB_BACKUP_MAX_FILE          12
#define DB_BACKUP_WRITE_FILE        -1
#define DB_BACKUP_TRANS_LOG_FILE    -2

/**************************/
/* db_environment constants */
/**************************/
```

```
#define DB_ENV_FLAGS                0       /* Short
int */
#define DB_ENV_ENGVERSION           1       /*
String */
#define DB_ENV_ENGNAME              2       /*
String */
#define DB_ENV_DBNAME               3       /*
String */
#define DB_ENV_SRVNAME              4       /*
String */
#define DB_ENV_SRVVERSION           5       /*
String */

#endif
```

CHAPTER 34

ODBC Programming

About this chapter

The **Open Database Connectivity (ODBC)** interface is a C programming language interface defined by Microsoft Corporation as a standard interface to database management systems in the Windows and Windows NT environments.

> **ODBC reference online**
> The primary documentation for ODBC application development provided with SQL Anywhere is the ODBC application programming interface (API) online Help.

Applications using the ODBC interface can work with many different database systems. SQL Anywhere supports the ODBC API on DOS, OS/2, and QNX in addition to the Windows and Windows NT environments. By having multi-platform ODBC support, SQL Anywhere makes portable database application development much easier.

This chapter presents information for those developing ODBC interfaces to SQL Anywhere. Users of application development systems that already have ODBC support do not need to read this chapter.

Contents

Topic	Page
ODBC C language programming	752
ODBC programming for the Macintosh	762

ODBC C language programming

The ODBC interface is defined by a set of function calls, called the ODBC API (Application Programming Interface).

To write ODBC applications for SQL Anywhere, you need:

- The SQL Anywhere software.
- A C compiler capable of creating programs for your environment.
- The Microsoft ODBC Software Development Kit (Optional) (available on the Developer Network CD) which provides additional tools for testing ODBC applications as well as sample drivers.

Fundamentals

Although the interface to ODBC is through function calls, database access is specified using SQL statements passed as strings to ODBC functions.

The following fundamental objects are used for every ODBC program. Each object is referenced by a **handle**.

- **Environment** Every ODBC application must allocate exactly one environment using SQLAllocEnv, and must free it at the end of the application using SQLFreeEnv. An environment handle is of type **HENV**.
- **Connections** An application can have several connections associated with its environment. Each connection must be allocated using SQLAllocConnect and freed using SQLFreeConnect. Your application can connect to a data source using either SQLConnect, SQLDriverConnect or SQLBrowseConnect, and disconnect using SQLDisconnect. A connection handle is of type HDBC.
- **Statements** Each connection can have several statements, each allocated using SQLAllocStmt and freed using SQLFreeStmt. Statements are used both for cursor operations (fetching data) and for single statement execution (e.g. INSERT, UPDATE and DELETE). A statement handle is of type HSTMT.

All access to these objects is through function calls; the application cannot directly access any information about the object from its handle. In the Windows and Windows NT environments, all the function calls are described in full detail in the ODBC API help file (ODBCAPI.HLP) provided with the Windows and Windows NT installations of SQL Anywhere.

Compiling and linking an ODBC application

Every C source file using ODBC functions must include one of the following lines:

- **Windows 3.x** #include "winodbc.h"
- **DOS** #include "dosodbc.h"
- **32-bit DOS** #include "ds32odbc.h"
- **OS/2** #include "os2odbc.h"
- **Windows 95 and NT** #include "ntodbc.h"
- **QNX** #include "qnxodbc.h"
- **32-bit QNX** #include "qnxodbc.h"

These files all include the main ODBC include file ODBC.H, which defines all of the functions, data types and constant definitions required to write an ODBC program. The file ODBC.H and the environment specific include files are installed in the H subdirectory of the SQL Anywhere installation directory (usually C:\SQLANY50).

Once your program has been compiled, you must link with the appropriate library file to have access to the ODBC functions:

- **Windows 3.x** ODBC.LIB, which defines the entry points into the Driver Manager ODBC.DLL. If you connect to a SQL Anywhere database in Windows, ODBC.DLL will load the SQL Anywhere ODBC driver, WOD50W.DLL.
- **DOS** WODBCWCL.LIB, which is a large model WATCOM C library. You will also need the SQL Anywhere interface library DBLIBWCL.LIB.

  WODBCMCL.LIB, which is a large model Microsoft C library. You will also need the SQL Anywhere interface library DBLIBMCL.LIB.

  WODBCBCL.LIB, which is a large model Borland C library. You will also need the SQL Anywhere interface library DBLIBBCL.LIB.

- **32-bit DOS** WODBCWFG.LIB, which is a 32-bit flat model WATCOM C library. You will also need the SQL Anywhere interface library DBLIBWFG.LIB.
- **OS/2** WODBC2.LIB, which defines the entry points into the SQL Anywhere ODBC driver, WOD502.DLL.
- **Windows 95 and NT** WODBC32.LIB, which defines the entry points into the Driver Manager ODBC32.DLL. If you connect to a SQL Anywhere database in Windows NT, ODBC32.DLL will load the SQL Anywhere ODBC driver, WOD50T.DLL.
- **32-bit QNX** odbc3r.lib, which is a 32-bit WATCOM C library. You will also need the SQL Anywhere interface library dblib3r.lib.

A first example

The following is a simple ODBC program:

```
{
   HENV  env;
   HDBC  dbc;
   HSTMT stmt;

   SQLAllocEnv( &env );
   SQLAllocConnect( env, &dbc );
   SQLConnect( dbc, "sademo", SQL_NTS,
               "dba", SQL_NTS, "sql", SQL_NTS );
   SQLSetConnectOption( dbc, SQL_AUTOCOMMIT, FALSE
);
   SQLAllocStmt( dbc, &stmt );

   /* Delete all the order items for order 2015 */
   SQLExecDirect( stmt,
      "delete from sales_order_items where id=2015",
       SQL_NTS );

   /* Use rollback to undo the delete */
   SQLTransact( env, dbc, SQL_ROLLBACK );
   SQLFreeStmt( stmt, SQL_DROP );
   SQLDisconnect( dbc );
   SQLFreeConnect( dbc );
   SQLFreeEnv( env );
}
```

Notes

SQL_NTS Every string passed to ODBC has a corresponding length. If the length is unknown, you can pass SQL_NTS indicating that it is a 'Null Terminated String' whose end is marked by the null character ('\0').

SQLExecDirect Executes the SQL statement specified in the parameter.

SQLTransact Used to perform COMMIT and ROLLBACK statements marking the end of a transaction. You should not use SQLExecDirect to perform COMMIT or ROLLBACK.

Error checking

The previous example did not check for any errors. Errors in ODBC are reported using the return value from each of the ODBC API function calls and the SQLError function.

Every ODBC API function returns a RETCODE which is one of the following status codes:

- **SQL_SUCCESS** No error.
- **SQL_SUCCESS_WITH_INFO** The function completed, but a call to SQLError will indicate a warning. The most common case for this status is when a value being returned is too long for the buffer provided by the application.
- **SQL_ERROR** The function did not complete due to an error. You can call SQLError to get more information on the problem.
- **SQL_INVALID_HANDLE** An invalid environment, connection or statement handle was passed as a parameter. This often happens if a handle is used after it has been freed, or if the handle is the null pointer.
- **SQL_NO_DATA_FOUND** There is no information available. The most common use for this status is when fetching from a cursor, and it indicates that there are no more rows in the cursor.
- **SQL_NEED_DATA** Data is needed for a parameter. This is an advanced feature described in the help file under SQLParamData and SQLPutData.

Every environment, connection and statement handle can have one or more errors or warnings associated with it. Each call to SQLError returns the information for one error and removes the information for that error. If you do not call SQLError to remove all errors, the errors are removed on the next function call which passes the same handle as a parameter.

The following program fragment uses SQLError and return codes:

```
HDBC dbc;
HSTMT stmt;
RETCODE retcode;
UCHAR errmsg[100];
```

```
. . .

retcode = SQLAllocStmt( dbc, &stmt );
if( retcode == SQL_ERROR ) {
   SQLError( env, dbc, SQL_NULL_HSTMT, NULL, NULL,
            errmsg, sizeof(errmsg), NULL );

   /* Assume that print_error is defined */
   print_error( "Failed SQLAllocStmt", errmsg );
   return;
}

/* Delete items for order 2015 */
retcode = SQLExecDirect( stmt,
      "delete from sales_order_items
      where id=2015", SQL_NTS );
if( retcode == SQL_ERROR ) {
   SQLError( env, dbc, stmt, NULL, NULL,
            errmsg, sizeof(errmsg), NULL );
   /* Assume that print_error is defined */
   print_error( "Failed to delete items", errmsg );
   return;
}
. . .
```

Note that each call to SQLError passes in three handles for an environment, connection and statement. The first call uses SQL_NULL_HSTMT to get the error associated with a connection. Similarly, a call with both SQL_NULL_DBC and SQL_NULL_HSTMT will get any error associated with the environment handle.

The return value from SQLError may seem confusing. It returns SQL_SUCCESS if there is an error to report (not SQL_ERROR), and SQL_NO_DATA_FOUND if there are no more errors to report.

The examples pass the null pointer for some of the parameters to SQLError. The help file contains a full description of SQLError and all its parameters.

Cursors in ODBC

ODBC cursors are similar to cursors in Embedded SQL (see "Cursors in Embedded SQL" in the chapter "The Embedded SQL Interface"). A cursor is opened using SQLExecute or SQLExecDirect, rows are fetched using SQLFetch or SQLExtendedFetch and the cursor is closed using SQLDropStmt.

To get values from a cursor, the application can use either SQLBindCol before a fetch or SQLGetData after a fetch.

The following code fragment opens and reads a cursor. Error checking has been omitted to make the example easier to read.

```
. . .
HDBC dbc;
HSTMT stmt;
RETCODE retcode;
long emp_id;
char emp_lname[20];

SQLAllocStmt( dbc, &stmt );
SQLExecDirect( stmt,
               "select emp_id,emp_lname
               from employee", SQL_NTS );
SQLBindCol( stmt, 1, SQL_C_LONG, &emp_id,
               sizeof(emp_id), NULL );
SQLBindCol( stmt, 2, SQL_C_CHAR, &emp_lname,
               sizeof(emp_lame), NULL );

for(;;) {
   retcode = SQLFetch( stmt );
   if( retcode == SQL_NO_DATA_FOUND ) break;
   print_employee( emp_id, emp_lname);
}

/* Using SQL_CLOSE closes the cursor
   but does not free the statement */
SQLFreeStmt( stmt, SQL_CLOSE );
. . .
```

Procedures and triggers in ODBC

Procedures can be both created and called using ODBC. Triggers can also be created. Refer to "Using Procedures, Triggers, and Batches" for a full description of stored procedures and triggers.

There are two types of procedures in SQL Anywhere; those that return result sets and those that do not. Use SQLNumResultCols to tell the difference. The number of result columns will be zero if the procedure does not return a result set. If there is a result set, you can fetch the values using SQLFetch or SQLExtendedFetch just like any other cursor.

Parameters to procedures should be passed using parameter markers (question marks). Use SQLSetParam to assign a storage area for each parameter marker whether it is an INPUT, OUTPUT or INOUT parameter.

The following example creates and calls a procedure. The procedure takes one INOUT parameter, and increments its value. In the example, the variable "num_col" will have the value zero, since the procedure does not return a result set. Error checking has been omitted to make the example easier to read.

```
HDBC dbc;
HSTMT stmt;
long i;
SWORD num_col;

/* Create a procedure */
SQLAllocStmt( dbc, &stmt );
SQLExecDirect( stmt,
      "CREATE PROCEDURE Increment( INOUT a INT )" \
      " BEGIN" \
         " SET a = a + 1" \
      " END", SQL_NTS );

/* Call the procedure to increment 'i' */
i = 1;
SQLSetParam( stmt, 1, SQL_C_LONG, SQL_INTEGER, 0,
            0, &i, NULL );
SQLExecDirect( stmt, "CALL Increment( ? )",
            SQL_NTS );
SQLNumResultCols( stmt, &num_col );
do_something( i );
```

The following example calls a procedure that returns a result set. In the example, the variable "num_col" will have the value two, since the procedure returns a result set with two columns. Again, error checking has been omitted to make the example easier to read.

```
HDBC dbc;
HSTMT stmt;
SWORD num_col;
RETCODE retcode;
char emp_id[ 10 ];
char emp_lame[ 20 ];

/* Create the procedure */
SQLExecDirect( stmt,
      "CREATE PROCEDURE employees()" \
      " RESULT( emp_id CHAR(10), emp_lname
CHAR(20))"\
      " BEGIN" \
      " SELECT emp_id, emp_lame FROM employee" \
      " END", SQL_NTS );

/* Call the procedure - print the results */
SQLExecDirect( stmt, "CALL employees()", SQL_NTS );
SQLNumResultCols( stmt, &num_col );
SQLBindCol( stmt, 1, SQL_C_CHAR, &emp_id,
            sizeof(emp_id), NULL );
```

```
SQLBindCol( stmt, 2, SQL_C_CHAR, &emp_lname,
            sizeof(emp_lname), NULL );

for( ;; ) {
   retcode = SQLFetch( stmt );
   if( retcode == SQL_NO_DATA_FOUND ) {
   retcode = SQLMoreResults( stmt );
   if( retcode == SQL_NO_DATA_FOUND ) break;
} else {
   do_something( emp_id, emp_lname );
}
}
```

ODBC conformance

The SQL Anywhere ODBC driver supports all of the ODBC Version 2.5 API functions (Core, Level 1 and Level 2) under Windows 95 and Windows NT. ODBC Version 2.1 support is provided for Windows 3.x. However, the driver cannot handle the following ODBC features:

- **Asynchronous statement execution** Specifying the SQL_ASYNC_ENABLE value for fOption to SQLSetStmtOption and SQLSetConnectOption is an error, and SQLCancel is supported only as an equivalent to SQLFreeStmt(SQL_CLOSE).
- **Unsupported functions** SQL strings passed as a parameter to SQLPrepare, SQLExecDirect or SQLNativeSql which use one of the following database functions return an error because the functions are not supported by SQL Anywhere.
  - REPLACE
  - TIMESTAMPADD
  - TIMESTAMPDIFF
  - DATABASE

The sample program

A sample ODBC program, ODBC.C, is supplied in the CXMP subdirectory of the SQL Anywhere installation directory (usually C:\SQLANY50). The program performs the same actions as the embedded SQL dynamic cursor example program. See "Database examples" in the chapter "The Embedded SQL Interface" for a description of the associated embedded SQL program.

Building the sample program

Along with the sample program is a batch file, MAKEALL.BAT, that can be used to compile the sample program for the various environments and compilers supported by SQL Anywhere. For OS/2 the command is MAKEALL.CMD. For QNX, use the shell script makeall.. The format of the command is as follows:

```
makeall odbc {Platform} {Compiler} [DOS Extender]
```

The first parameter is "odbc", meaning compile the ODBC example. The same batch file also compiles the embedded SQL programs.

The second parameter is the platform in which the program will be run. The platform can be one of:

- **DOS** compile for 16-bit DOS.
- **DOS32** compile for 32-bit DOS. Use 4G to specify Rational DOS/4GW as the DOS Extender parameter.
- **WINDOWS** compile for 16-bit Windows.
- **WIN32** compile for 32-bit Windows using the Watcom C 32-bit Windows support.
- **WINNT** compile for Windows NT or Windows 95.
- **OS232** compile for OS/2.
- **QNX** compile for 16-bit or 32-bit QNX.

The third parameter is the compiler to use to compile the program. The compiler can be one of:

- **WC** use Watcom C or Watcom C/32
- **MC** use Microsoft C
- **BC** use Borland C
- **CS** use IBM C Set++

Building the sample program as an NT service

The example program ODBC., when compiled for Windows NT, runs optionally as a service.

The two files containing the example code for NT services are the source file NTSVC.C and the header file NTSVC.H. The code allows the linked executable to be run as either a regular executable or as an NT service.

To run the compiled example as an NT service:

1 Start the SQL Anywhere Service Manager.

2 Click New. The Service Type window is displayed.

3 Click Sample Application, and click OK. The Edit SQL Anywhere Service Configuration window is displayed.

4 Enter a service name.

5 Click Path, and select the sample program (ODBCNT.EXE) from the CXMP subdirectory of the installation directory.

6 Click OK to install the service.

7 Click Start on the main window to start the service.

When run as a service, the program displays the normal user interface if possible. It also writes the output to the Application Event Log. If it is not possible to start the user interface, the program prints one page of data to the Application Event Log and stops.

This example has been tested with the Watcom C/C++ 10.5 compiler and the Microsoft Visual C++ 2.0 compiler.

ODBC programming for the Macintosh

When the Macintosh version of SQL Anywhere becomes available, developers of client applications other than PowerBuilder applications should follow the guideline in this section. Failure to do so could have severe effects on performance.

You should define a callback function to be used by the SQL Anywhere ODBC driver when it encounters an event it does not know how to handle. This callback function is particularly important if an update event gets into the application's queue. If this update event is not explicitly handled, and hence remains in the queue, the interprocess communication between the SQL Anywhere ODBC driver and the engine or client slows down dramatically, because of the high priority assigned to update events by the Macintosh operating system.

To define the callback function, you should include the provided C language header file EVENTCB.H. This header file has the following contents:

```
#define WSQL_OPT_REGISTER_CALLBACK_EVENTS 1900

typedef void pascal  (MAC_EVENT_CALLBACK)(
EventRecord * theEvent );

typedef MAC_EVENT_CALLBACK *
   MAC_EVENT_CALLBACK_PTR;
```

You should define the callback function as follows (the name may be changed).

```
#include "eventcb.h"
static void pascal EventHandler( EventRecord *
theEvent )
{
   switch( theEvent->what ) {
      // The usual event processing goes here.
      // MUST process updateEvt fully, even if
      // just discarding as shown here.
      case updateEvt:
         BeginUpdate( (WindowPtr) theEvent->message
);
         EndUpdate( (WindowPtr) theEvent->message );
         break;
   }
}
```

Other events may also be handled; *updateEvt* must be handled to avoid performance degradation.

The following fragment illustrates how to use the callback function.

```
extern void main()
{
```

```
    // ...
    SQLAllocConnect( env, &connection );
       // normal connection stuff
       SQLSetConnectOption( connection,
    WSQL_OPT_REGISTER_CALLBACK_EVENTS,
       (UDWORD) EventHandler );
    // ...
}
```

CHAPTER 35

The WSQL DDE Server

About this chapter

WSQL DDE Server is a Windows application which enables you to access and alter data in SQL Anywhere databases using **Dynamic Data Exchange** (**DDE**). This means that applications which can use DDE can access data directly, without having to use other database-querying applications such as ISQL. Many applications such as Excel and programming environments such as Visual Basic support the DDE protocol.

WSQL DDE Server can put query results on the clipboard, or return a memory buffer containing the query results. It can also send commands to the database engine, such as COMMIT or UPDATE.

Some applications have a limit on the number of characters which they can send at one time using DDE. For this reason, WSQL DDE Server can build a command or query piece by piece before executing it.

Contents

Topic	Page
DDE concepts	766
Using WSQL DDE Server	768
Excel and WSQL DDE Server	772
Word and WSQL DDE Server	774
Visual Basic and WSQL DDE Server	775

DDE concepts

The following sections provide an introduction to dynamic data exchange (**DDE**).

What is DDE?

DDE is a method of **interprocess communication**, that is, a method for passing data back and forth between applications. DDE uses a **client/server** system: client applications such as Excel request data from server applications, such as WSQL DDE Server.

DDE conversations

An organized exchange of data using DDE is called a **conversation**. DDE conversations are distinguished by a three-level identification scheme: each conversation employs a **service name**, a **topic name**, and an **item name**.

At the top level is the **service name**. The WSQL DDE Server application uses the service name *WSQLDDE*. All conversations between a client application and WSQL DDE Server must use this service name.

The second level is the **topic name**. The topic name identifies the user and (optionally) specifies which SQL Anywhere database he wishes to query. The topic name must be of the form *UserName,Password,Database* or *UserName,Password*. (In the latter case, the default database environment is chosen.)

The third level is the **item name**. The item name further specifies the action to be performed by WSQL DDE Server. For instance, *Connect* and *Clear* are valid item names when sending data to WSQL DDE Server, while *Column_Names_And_Data* is a valid item name when requesting information from WSQL DDE Server.

Attempts to start a DDE conversation without a service name or a topic name are called **wildconnects**, and are not supported by WSQL DDE Server.

Sending and receiving data and commands using DDE

There are three main ways for a client to exchange information with a server: poking, executing, and requesting. These three operations are called **DDE transactions** (not to be confused with database transactions).

- **Poke** send information from the client to the server. Typically, the client will poke data specifying a query to the server, and set the item name to a command acting upon the data.
- **Execute** send the command directly to the database engine. No data is returned by a DDE execute transaction, and item names are ignored.
- **Request** lets WSQL DDE Server know that the client wants to receive a memory buffer containing the results of a query. (The query must have already been added to the query buffer.) WSQL DDE Server does not support warm or hot request links (see below).

DDE links

Methods of data exchange are often called **links**. There are three types of DDE links:

- A **cold link** occurs when the client requests data and the server immediately supplies the data.
- A **warm link** occurs when the server advises the client that some data that the client is interested in has changed. It is then the client's responsibility to request the data. WSQL DDE Server does not support warm links.
- A **hot link** occurs when the server sends a piece of data to the client as soon as the data changes. WSQL DDE Server does not support hot links.

Using WSQL DDE Server

The WSQL DDE Server executable has the file name WSQLDDE.EXE, and will usually be run from your c:\sqlany50 directory.

The WSQL DDE Server window

The WSQL DDE Server window consists of an output window and a menu bar. The output window is updated every time WSQL DDE Server processes a command. It also displays any errors which have occurred. (If an error occurs, a message box will also inform you.) Every line of the output window begins with the DDE conversation number of the application which sent the command or caused the error. The output window will display the last five hundred commands and errors.

The menu bar has one item, File, which has the following selections:

- **About WSQL DDE Server** Display general information about WSQL DDE Server.
- **Clear Output Window** Erase the contents of the output window.
- **Exit** Leave WSQL DDE Server. If there are still active DDE conversations, then a message box will ask you to confirm your decision to exit.

Initiating a DDE conversation with WSQL DDE Server

A DDE conversation must be initiated before exchanging data using WSQL DDE Server. Two pieces of information are used to initiate a conversation with WSQL DDE Server: the **service name** and the **topic name**.

- The **service name** must be WSQLDDE.
- The **topic name** identifies the user, and optionally, the database to be used. It is a string of the form *User,Password,Database* or just *User,Password*. (Optionally, periods may be used instead of commas.)

Database connection is separate
The database is not activated or connected to when the DDE conversation is initialized. This occurs when the Connect command is sent.

Communicating with the WSQL DDE Server

Once a conversation is initiated, there are three ways of communicating with WSQL DDE Server: **poking**, **executing** and **requesting**. The three methods are documented on the following pages.

Poking commands to WSQL DDE Server

Poking sends two pieces of data from the client application to WSQL DDE Server: a **data buffer** and an **item name** specifying what to do with the data.

POKE *item-name data-buffer*

There are five WSQL DDE Server functions which you can access through poking: *Connect*, *Disconnect*, *Add*, *Clear* and *Clip*.

Connecting to a database

POKE CONNECT

Before any queries or commands can be sent to the database engine, you must connect to the database. To do this, set the item name to *Connect*. WSQL DDE Server will attempt to connect to the database environment mentioned in the DDE topic. If the database engine is not running, then WSQL DDE Server will attempt to start it.

If there is no database name mentioned in the topic, then WSQL DDE Server will try to connect to an already-running database engine. (If there is no running engine, then the connect will fail.)

The contents of the data buffer are ignored.

One connection only
It is not possible to have more than one connection per conversation.

Disconnecting from a database

POKE DISCONNECT

To terminate the current connection, set the item name to *Disconnect*. The contents of the data buffer are ignored.

Terminate the conversation separately
This command terminates the connection to the database, but does not terminate the DDE conversation (though it will usually be the last command sent in a DDE conversation).

Adding to the query buffer

POKE ADD *text*

Some applications (such as Excel) can only poke 255 characters at a time. Many SQL queries, however, are longer than this. For this reason, WSQL DDE Server allows an application to build a complete query string by adding together many shorter strings. To add text to the query buffer, set the item name to *Add* and the data buffer to the text you wish to add to the buffer.

Each DDE conversation has its own query buffer.

Clearing the query buffer

POKE CLEAR *text*

To reset the query buffer, set the item name to *Clear* and the data buffer to the string you wish to place at the beginning of the query buffer.

Putting query results on the clipboard

POKE
CLIP_DATA
| CLIP_COLUMN_NAMES
| CLIP_COLUMN_NAMES_AND_DATA
... [text]

There are three very similar commands which put the results of a query on the clipboard. Each adds the contents of the data buffer to the query buffer, and then evaluates the query stored in the query buffer. The results of the query are formatted and placed on the clipboard. Most applications have a Paste command which will retrieve the query results from the clipboard.

The three commands are:

Clip_Data place the results of the query on the clipboard in text format. Each datum will be separated by a tab character. Each row will be terminated by a newline character.

Clip_Column_Names place the names of the columns referenced in the query onto the clipboard. Column names will be separated by tab characters, and the row will be terminated by a newline character.

Clip_Column_Names_And_Data does both operations, appending the data after the column names.

Executing database commands

EXECUTE *[text]*

Executing a command adds the command to the query buffer and sends the contents of the query buffer directly to the database engine. It is very similar to doing an EXECUTE IMMEDIATE from Embedded SQL. Executed commands do not return results to the application, so queries cannot be executed. Item names are ignored when doing a DDE execute transaction.

Requesting

REQUEST

DATA
| COLUMN_NAMES
| COLUMN_NAMES_AND_DATA

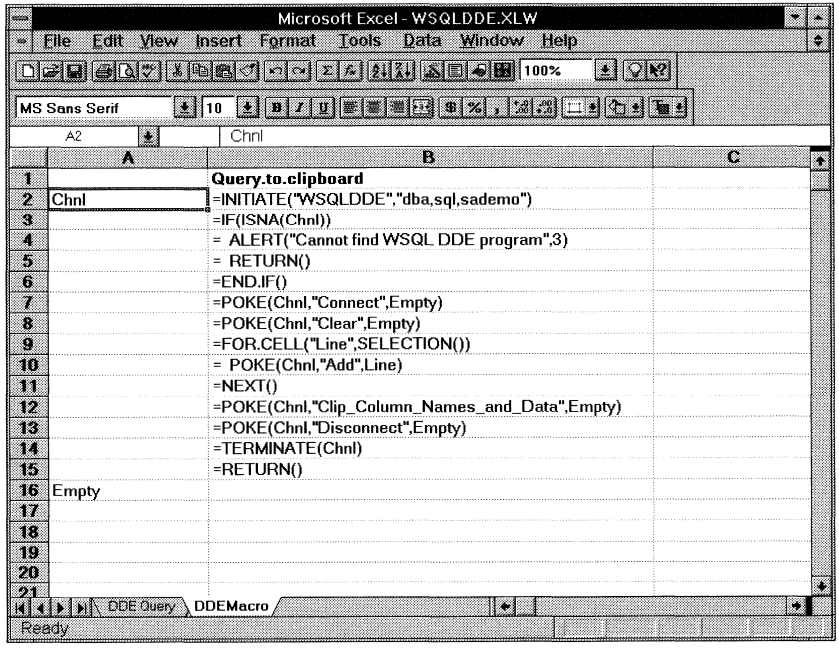
Requesting data gives the calling application a block of memory containing the requested data, as opposed to placing the data on the clipboard for the application to access. (This is useful for applications that do not support clipboard access.)

WSQL DDE Server only supports cold request links. (That is, it is the application's responsibility to request data. See the section on DDE above for more information about link types.) WSQL DDE Server places the results of the query currently in the query buffer into a memory block and then sends the address of the memory to the application.

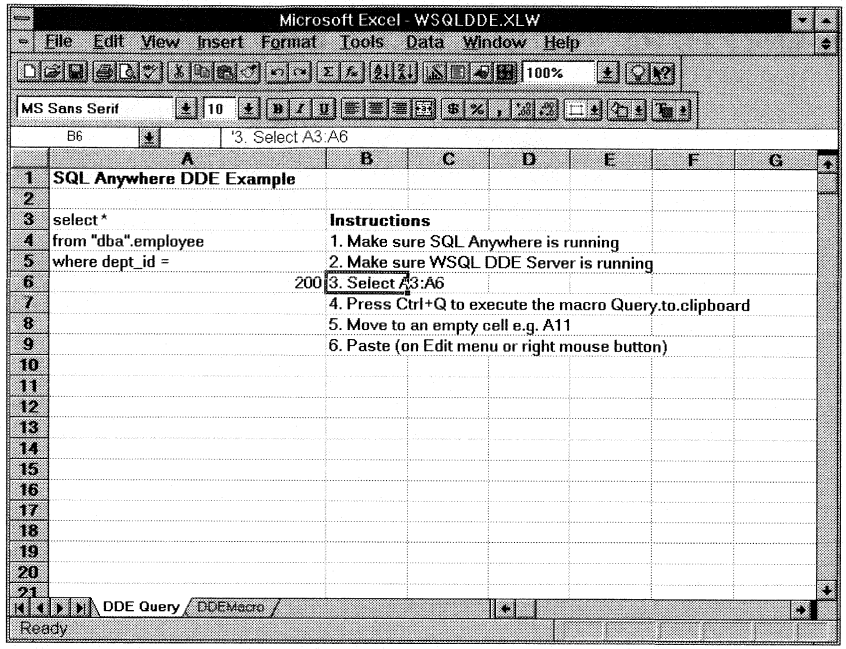
There are three valid item names for the request transaction: *Data*, *Column_Names* and *Column_Names_And_Data*. The structure of the data in the buffer is identical to that of the data placed on the clipboard when doing a *Clip_Data*, *Clip_Column_Names* or *Clip_Column_Names_And_Data* command. (See above.)

Excel and WSQL DDE Server

There are a number of Excel Macro Sheet DDE functions that can access the WSQL DDE Server DDE Server. The **Query.to.clipboard** macro shown in demonstrates the use of Macro Sheets to initiate and terminate a DDE conversation, connect and disconnect from a database, and clip data. (Excel supports Execute, although this example doesn't use it.)



This macro is included in the WSQLDDE.XLW workbook in the directory C:\SQLANY50\ACCXMP\EXCEL. You should start the database engine and the WSQL DDE Server before loading the workbook. You will see the sheet in Follow the instructions to clip data from the database and paste it on the sheet.


Microsoft Excel - WSQLDDE.XLW
File Edit View Insert Format Tools Data Window Help
100%
MS Sans Serif
10
B6
'3. Select A3:A6
SQL Anywhere DDE Example
select *
from "dba".employee
where dept_id =
200
Instructions
1. Make sure SQL Anywhere is running
2. Make sure WSQL DDE Server is running
3. Select A3:A6
4. Press Ctrl+Q to execute the macro Query.to.clipboard
5. Move to an empty cell e.g. A11
6. Paste (on Edit menu or right mouse button)
DDE Query
DDEMacro
Ready

Word and WSQL DDE Server

Microsoft Word Basic supports DDE. The following example contains sample Word Basic code to access the WSQL DDE Server DDE server. The two different ways of retrieving the data are demonstrated: using the clipboard and receiving the data directly.

```
Sub MAIN
   ' Initialize a DDE channel to the database
   chan = DDEInitiate("WSQLDDE", "dba,sql,sademo")

   ' Connect to the database
   DDEPoke chan, "connect", ""

   ' Get the data using the clipboard
   DDEPoke chan, "clip_data", \
      "select emp_lname \
      from employee where emp_id = 102"
   EditPaste

   ' Insert the data directly into the document
   DDEPoke chan, "clear", \
      "select emp_lname \
      from employee where emp_id = 105"
   Insert DDERequest$(chan, "data")

   ' Disconnect from the database
   DDEPoke chan, "disconnect", ""

   ' Terminate the DDE connection
   DDETerminate(chan)
End Sub
```

Visual Basic and WSQL DDE Server

The following Visual Basic code fragment initializes a DDE conversation, connects to a database, requests and clips some data, executes a command and finally disconnects from the database.

LinkTimeout property
Visual Basic controls have a LinkTimeout property, which specifies the amount of time the application will wait for a DDE function to execute. Some database queries can be time-consuming, so you may wish to set the LinkTimeout property to -1 (wait forever). The problem with this is that if there are link errors it may seem as though your computer has `locked up'. If this happens, try pressing the ALT or ESC key to abort the operation.

```
' INITIALIZING A DDE CONVERSATION
MyTextBox.LinkTopic = "WSQLDDE|DBA,SQL,sademo"
    ' Set the service name to WSQLDDE
    ' and the topic name to DBA,SQL,sademo
MyTextBox.LinkMode = 2
    ' Two is cold link mode

' The DDE conversation has now been initiated,
' but the Database Engine has not been accessed.

' CONNECTING
MyTextBox.LinkItem = "Connect"
MyTextBox.LinkPoke
    ' The Connect command has just been poked.
    ' WSQL DDE Server tries to connect to
    ' the sample database using
    ' the user name dba and password sql.
    ' The contents of MyTextBox are ignored
    ' and unaltered.

'REQUESTING A QUERY
MyTextBox.LinkItem = "Clear"
MyTextBox.Text = "SELECT * FROM department"
MyTextBox.LinkPoke
    ' The query buffer is cleared and
    ' the SELECT statement is
    ' added to it.

MyTextBox.LinkItem = "Column_Names_And_Data"
MyTextBox.LinkRequest
    ' The LinkRequest sets MyTextBox to
    ' the results of the query
    ' that we just added to it.
    ' Had the LinkItem been "Column_Names"
    ' then only the column names
    ' would have appeared in MyTextBox.
```

```
    ' Similarly, if the LinkItem
    ' had been "Data" then just the data rows
    ' would have appeared.

'CLIPPING A QUERY
MyTextBox.LinkItem = "Clear"
MyTextBox.Text = " SELECT emp_lname, emp_id FROM
employee "
MyTextBox.LinkPoke
    ' The query buffer is cleared and
    ' the SELECT statement is added.

MyTextBox.LinkItem = "Add"
MyTextBox.Text = " WHERE emp_id < 150 "
MyTextBox.LinkPoke
    ' The query buffer now contains:
    ' SELECT emp_lname, emp_id
    ' FROM employee WHERE emp_id < 150

MyTextBox.LinkItem = "Clip_Data"
MyTextBox.Text = " ORDER BY emp_lname "
MyTextBox.LinkPoke
    ' The ORDER BY clause is appended to the
    ' query buffer, and the
    ' data from the query is placed on the
    ' clipboard. The LinkItem
    ' could also have been "Clip_Column_Names" or
    ' "Clip_Column_Names_And_Data"

MyTextBox.Text = ClipBoard.GetText( )
    ' This pastes the contents of the clipboard
    ' to the text box.

'EXECUTING A COMMAND
MyTextBox.LinkItem = "Clear"
MyTextBox.Text = "UPDATE employee SET dept_id=200 "
MyTextBox.LinkPoke
MyTextBox.LinkExecute " WHERE emp_lname='Chin' "
' The WHERE clause is appended to the query buffer
and the command
' UPDATE employee SET dept_id=200 WHERE
emp_lname='Chin'
' is sent to the database engine.

'DISCONNECTING
MyTextBox.LinkItem = "Disconnect"
MyTextBox.LinkPoke
    ' The connection to the database
    ' engine is terminated.
    ' The DDE conversation still exists,
    ' so another Connect
    ' could be done.
```

The test application

There is a Visual Basic test application (both source and executable) included with WSQL DDE Server (see directory C:\SQLANY50\ACCXMP\VB) which demonstrates the features of WSQL DDE Server. The filename is WSQLDDET.EXE. You can take a look at the source by loading the project WSQLDDET.MAK if you have the Visual Basic software.

Running the test application

Start the database engine (on the sample database), WSQL DDE Server, and the test application. A window containing a large text box, a small text box and a button marked Connect will appear. Press the Connect button. (If you want to use a different username or password than the default, enter them into the small text box before connecting. The contents of this box will become the Link Topic.)

Pressing the Connect button terminates the current DDE conversation (if one exists), starts a new DDE conversation with the given topic, and pokes a *Connect* to the DDE server. After a moment, the Test Application screen will change. There will now be a combobox and seven buttons: Disconnect, Query to Clipboard, Query to Memory, Add to Buffer, Clear Buffer, Clear Screen and Execute Command.

The combo box is used to select which portions of a result set you want. There are three selections, which allow you to see just column names, just data, or both column names and data.

The Disconnect button terminates the database connection, but not the DDE conversation. (The DDE conversation can be terminated by quitting the test application.) You will then be given the option to connect again. You should disconnect from the database engine before quitting the test application.

Pressing the Query to Clipboard button first adds the contents of the small text box to the query buffer, and then gets the query results from the database engine. The result set is placed on the clipboard, and the clipboard contents are placed in the large text box. Finally, the query buffer is cleared.

Pressing the Query to Memory button first adds the contents of the small text box to the query buffer, and then sends a DDE request transaction. The result set is placed directly in the large text box. The query buffer is then cleared.

Pressing the Execute button first adds the contents of the small text box to the query buffer, and then executes the contents of the query buffer. The execute command does not put any data in the large text box.

The execute command uses the DDE execute transaction.

Pressing the Add to Query Buffer button adds the contents of the small text box to the query buffer.

Pressing the Clear Query Buffer button erases the whole query buffer.

Pressing the Clear Screen button clears the contents of the large text box.

CHAPTER 36

The WSQL HLI Interface

About this chapter

WSQL HLI is a high level callable interface which allows you to access data in SQL Anywhere databases from a variety of applications and programming environments. This chapter documents the WSQL HLI programming interface.

WSQL HLI is available as a DLL for Windows, OS/2 and Windows NT.

The WSQL HLI call interface allows you to access information in SQL Anywhere databases from a programming environment such as Visual Basic in Windows or REXX in OS/2. It allows programmers to use a record-at-a-time cursored database interface similar to Embedded SQL or a set-at-a-time interface similar to the WSQL DDE Server . WSQL HLI can process queries and database commands using host variables, return query results from both individual fields and entire query result sets, and provide error information. Most of the functionality is provided by the **wsqlexec** function which takes a string as a parameter. This string is parsed by WSQL HLI, and the appropriate Embedded SQL commands are executed.

Contents

Topic	Page
DLL concepts	780
Using WSQL HLI	781
Host variables with WSQL HLI	782
WSQL HLI functions	783
wsqlexec command strings	790
WSQL HLI and Visual Basic	798
WSQL HLI and REXX	802

DLL concepts

A **Dynamic Link Library** (**DLL**) is a collection of functions which are available to any program. DLL's are different from regular applications in that they are unable to execute on their own. They are always called from another application or DLL.

The main benefits of DLL's are:

- their ability to be shared by many independent applications at the same time (DLL's are a **global** resource).
- they provide a high level of abstraction. An application does not need to know how the DLL works to use it.
- their high upgradeability. If a new version of a DLL is released you should be able to use the new one immediately, without modifying, recompiling or relinking your software.
- their ease of use. In some programming environments all you have to do is declare which DLL you are using. Calling DLL functions is then identical to calling normal language functions. (DLL's are sometimes called **plug-and-play** for this reason.)

The WSQL HLI DLL is a shared system resource—it can be used by multiple applications. However, internally, all resources such as database connections, statements and cursors are managed on a per task basis. Thus, there will not be any name conflicts between various applications using the DLL.

Using WSQL HLI

The WSQL HLI DLL has the file name WSQLCALL.DLL (Windows 3.x), WSQLCAL2.DLL (OS/2), or WSQLCALT.DLL (Windows NT and Windows 95).

Host variables with WSQL HLI

A **host variable** is a variable in an application which can be used (referenced) by WSQL HLI. Host variable support is an optional feature and requires the application to provide callback functions for host variable access. The Visual Basic interface does not provide host variable support. OS/2 REXX host variable support is provided by WSQL HLI (see "WSQL HLI and REXX" on page 802).

Host variables provide a convenient way to use program variables in SQL commands that are sent to WSQL HLI. You can also use host variables to FETCH values directly into program variables rather than using the **wsqlgetfield** function.

For example, if you had an application variable **myquery** containing a SELECT statement that you wanted to execute, then you could prepare it like this:

```
return_code = wsqlexec("PREPARE Stmt1 FROM
:myquery");
```

The following rules should be noted:

1 Host variable names must be under 30 characters long.

2 To distinguish host variables from regular text, they must be prefixed with a colon. (See the example above.)

WSQL HLI needs some way to reference host variables, to both access and change their contents. To do this, a pair of **callback functions** must be supplied to WSQL HLI. When WSQL HLI encounters a host variable, it calls the appropriate callback function to either retrieve or set the variable's value. For this reason, it is necessary to call the function **wsqlregisterfuncs** before using any host variables. **wsqlregisterfuncs** takes two parameters, both pointers to callback functions. The first function, **PutHostVar** specifies a host variable name and the string to put in it. The second function, **GetHostVar** specifies a host variable name and the address of a string pointer to fill in with its value. See "wsqlregisterfuncs" on page 788 for more information.

For example, the statement:

```
wsqlexec( "FETCH Cursor1 INTO :host1" );
```

would call the function registered as **PutHostVar** with the name *host1* and the value of the item fetched (converted to a string). Your application's callback function would then identify the host variable and save the new value.

WSQL HLI functions

WSQL HLI has six functions:

Ordinal	Function Name	Comments
3	wsqlexec	execute a command
4	wsqlgetfield	get a result column value
5	wsqlgetcolumnname	get a result column name
6	wsqlquerytomem	retrieve results in one piece of memory
7	wsqllasterror	get error string for last error
8	wsqlregisterfuncs	register host variable callback functions

The Windows DLL entry points are all FAR PASCAL functions and all pointers are far pointers. The OS/2 DLL entry points are all _System functions. The Windows NT entry points are all __stdcall functions.

All functions return a long integer containing a SQL error code indicating the success or failure of the function. The **wsqllasterror** function can be called to get a meaningful message about the last error. See "SQL Anywhere Database Error Messages" for a complete list of error codes. The following sections describe these functions in detail.

wsqlexec

The **wsqlexec** function is used to process all SQL requests, such as PREPAREing statements or OPENing cursors. In OS/2, the **wsqlexecrexx** function can be used from a REXX program (see "WSQL HLI and REXX" on page 802).

The C prototype for this function is:

```
extern long _entry wsqlexec(
   char *command
);
```

The command must be one of the following:

- CLIP
- CLOSE

- COMMIT
- DECLARE
- DROP
- EXECUTE
- FETCH
- OPEN
- PREPARE
- ROLLBACK
- SET CONNECTION
- START USING DATABASE
- STOP USING DATABASE

These commands are described fully in "wsqlexec command strings" on page 790.

wsqlgetfield

The **wsqlgetfield** function is used to retrieve the data value from one column of the most recently FETCHed row of a query result set.

The C prototype for this function is:

```
extern long _entry wsqlgetfield(
    char  * cname,
    int     offset,
    short * ind,
    char  * result,
    int     len
);
```

cname name of a cursor which is opened and positioned at the row of the query result set that you want to examine.

offset the column number which you wish to examine. Columns are numbered from left to right starting at 0.

ind indicator variable. If the **result** buffer is too small, then the required length is put in the indicator variable. If the requested item is null, then the indicator is set to -1, and if there is a conversion error, then the indicator is set to -2. Otherwise, it is set to zero.

result string buffer which will be filled in with the requested column name. The result buffer should be allocated and freed by the calling application.

len maximum number of characters that can be placed in the **result** buffer.

wsqlgetcolumnname

The **wsqlgetcolumnname** function is used to get one column name of a query result set.

The C prototype for this function is:

```
extern long _entry wsqlgetcolumnname(
   char  * cname,
   int     offset,
   short * ind,
   char  * result,
   int     len
);
```

cname name of a cursor that has been opened.

offset the column number whose name you want. Columns are numbered from left to right starting at 0.

ind indicator variable. If the **result** buffer is too small, then the required length is put in the indicator variable. Otherwise, it is set to zero.

result string buffer that will be filled in with the requested column name. The result buffer should be allocated and freed by the calling application.

len maximum number of characters that can be placed in the **result** buffer.

wsqlquerytomemdelim

The **wsqlquerytomemdelim** function is used to place a portion of a query result set into a memory buffer.

If column names are requested, then the resulting buffer will start with a list of column names.

If data is requested, then the buffer will contain all of the rows of the query result set from the current cursor position to the end of the result set. If the supplied buffer is not large enough, then as many rows as will fit will be placed in the result buffer, and the cursor will be positioned on the row following the last one placed in the buffer. If this occurs, then the return code will be SQLE_HLI_MORE_DATA_AVAILABLE.

Each row will end with the specified *rowdelim* and the data in the row will be separated by *coldelim*.

The C prototype for this function is:

```
extern long _entry wsqlquerytomemdelim(
   char * cursor,
   char * result,
   int len,
   bool data,
   bool names,
   char * coldelim,
   char * rowdelim
);
```

cursor name of a cursor that is opened and (if you want data) positioned at the first row of data to be returned.

result character buffer, allocated by the calling application. It will be filled in with as many rows as possible from the result set, starting at the current cursor position.

len maximum number of characters that should be placed in the **result** buffer.

data flag indicating that you want the buffer to contain query results. If **data** is zero, then no query results will be returned. Otherwise, query results will be returned, starting from the current cursor position.

names flag indicating that you want a list of column names. If it is zero, then column names will not be included. Otherwise, a list of column names will be placed at the beginning of the returned buffer.

coldelim A string that will be placed between each column name and each column value.

coldelim A string that will be placed at the end of the column names and at the end of each data row.

wsqlquerytomem

The **wsqlquerytomem** is similar to the **wsqlquerytomemdelim** function except the column delimiter is fixed as the tab character and the row delimiter is a carriage return - linefeed pair.

The C definition for this function is:

```
extern long _entry wsqlquerytomem(
   char * cursor,
   char * result,
   int len,
   bool data,
   bool names,
)
{
   return( wsqlquerytomemdelim( cursor, result,
   len, data, names, ".t", ".r.n" ) );
}
```

The parameters are the same as those for **wsqlquerytomemdelim**.

wsqllasterror

The **wsqllasterror** function is used to provide a meaningful message explaining the last error condition. :note.The return code will be the number of the last error condition.

The C prototype for this function is:

```
extern long _entry wsqllasterror(
   char * buffer,
   int len
);
```

buffer string buffer which will be filled in with an error message. It should be allocated and freed by the calling application.

len maximum number of characters that can be placed in **buffer**. For a complete description of SQL Anywhere errors, see "SQL Anywhere Database Error Messages".

wsqlregisterfuncs

The **wsqlregisterfuncs** function is used to register callback functions for host variables (see "**Host variables with WSQL HLI**" on page 782). Host variable use is optional. It is best used when your application system has the concept of variables and you can provide a callback function to access them. It is not necessary to register callback functions in order to use host variables from REXX.

The C prototype for this function is:

```
extern long _entry wsqlregisterfuncs(
   p_callback PutHostVar,
   p_callback GetHostVar
);
```

The Windows callbacks are FAR PASCAL functions that are exported and all pointers are far pointers. The OS/2 callbacks are _System functions. The Windows NT callbacks are __stdcall functions.

PutHostVar pointer to a callback function with the following prototype:

```
extern long _callback PutHostVar(
   char * name,
   short ind,
   char * value,
   int len
);
```

name name of the host variable whose value WSQL HLI wishes to change.

ind indicator variable for the value of the host variable. See "Indicator variables" in the chapter "The Embedded SQL Interface" for details.

value new value for the host variable. It is always a null terminated string.

len number of characters in **value**.

return value a SQL error code indicating the success or failure of the operation. If the host variable or its value are invalid, then the return value should be the SQLE_HLI_BAD_HOST_VAR_VALUE or SQLE_HLI_BAD_HOST_VAR_NAME codes.

GetHostVar pointer to a callback function with the following prototype:

```
extern long _callback GetHostVar(
   char * name,
   short * ind,
   char * * value,
   int * len
);
```

name name of the host variable whose value is needed by WSQL HLI.

ind place to put an indicator value, if one exists, for the host variable.

value pointer to a string buffer pointer. The **GetHostVar** function should fill in value with a pointer to a buffer containing the value of the host variable.

len number of characters placed in **value** by the function.

return value a SQL error code indicating the success or failure of the operation. If the host variable is invalid, then the return value should be the SQLE_HLI_BAD_HOST_VAR_NAME code.

wsqlexec command strings

This section contains a detailed description of each command that can be passed as a string to the **wsqlexec** function. The syntax conventions used in this chapter are the same as those used in "Watcom-SQL Language Reference".

The commands are listed one per page with a syntax summary box at the top of the page.

CLIP HLI Statement

Syntax

```
CLIP
        NAMES
      | DATA
      | BOTH
    ... cursor-name
```

cursor-name: identifier, or host variable

Purpose

To place query results on the Windows or Windows NT clipboard.

See also

wsqlquerytomem.

Description

This command places a query result set on the system clipboard. The **cursor-name** must identify an open cursor.

If NAMES is specified, then the clipboard contents will be a single row containing the column names of the result set of the indicated cursor. The names will be separated by tab characters and terminated with a carriage return / linefeed.

If DATA is specified, then the clipboard contents will be the entire query result set starting from the current cursor position. Each row will be terminated with a carriage return / linefeed and the values within a row will be separated by tab characters.

If BOTH is specified then the row of names will be placed on the clipboard first, followed by the rest of the result set.

If the BOTH or DATA keyword is specified then the cursor will be positioned just after the last row of the query result set.

CLOSE HLI Statement

Syntax

CLOSE cursor-name

cursor-name: identifier, or host variable

Purpose

Close the named cursor.

See also

OPEN, PREPARE, DECLARE.

Description

This command closes the indicated cursor. The cursor must have been previously OPENed.

This command is functionally similar to the Embedded SQL command CLOSE. See "CLOSE statement" in the chapter "Watcom-SQL Statements".

COMMIT HLI Statement

Syntax

COMMIT

Purpose

To make changes permanent to the database.

See also

ROLLBACK.

Description

COMMIT makes any changes you have made to the database permanent.

This command is functionally similar to the SQL command COMMIT. See "COMMIT statement" in the chapter "Watcom-SQL Statements" for details.

DECLARE HLI Statement

Syntax

DECLARE cursor-name CURSOR FOR statement-name

cursor-name: identifier, or host variable

statement-name: identifier, or host variable

Purpose

To declare a cursor. Cursors are used to retrieve query results a record at a time.

See also

DROP, FETCH, OPEN, PREPARE.

Description

This command associates a cursor with a particular statement. The statement name must have been used in a previous PREPARE command. Once declared, a cursor can be opened and fetched.

All cursors are scrollable in both directions and will remain open across COMMIT and ROLLBACK commands.

This command is functionally similar to the Embedded SQL command DECLARE. See "DECLARE CURSOR statement" in the chapter "Watcom-SQL Statements".

DROP HLI Statement

Syntax

DROP statement-name

statement-name: identifier, or host variable

Purpose

To free resources used by a statement.

See also

PREPARE, DECLARE.

Description

The DROP command frees resources used by a prepared statement, and unreserves the statement name (and any associated cursors). If there is a cursor associated with this statement, then it will be CLOSEd and unreserved.

This command is functionally similar to the Embedded SQL command DROP STATEMENT.

EXECUTE HLI Statement

Syntax

1. EXECUTE statement-name [USING hostvariablelist]

statement-name: identifier, or host variable

2. EXECUTE IMMEDIATE statement

statement: string or host variable

Purpose

To execute a SQL statement that is not a query (SELECT).

See also

PREPARE, DECLARE, DROP.

Description

This command executes a SQL statement. Format 1 executes a named statement which has been prepared. The prepared statement may contain host variable parameters. If it does, and no :keyword.USING clause is specified, then the values of the host variables named in the statement will be requested. If a USING clause is specified, then the parameters will be set to the values of the host variables in the list. The number of host variables in the list must match the number of host variable parameters specified in the statement.

Format 2 will take a SQL statement and PREPARE, EXECUTE and DROP it. The **statement** in Format 2 is specified as a string with single quotes, or as a host variable.

Any SQL statement except a SELECT statement may be executed.

This command is functionally similar to the Embedded SQL command EXECUTE. See "EXECUTE statement" in the chapter "Watcom-SQL Statements".

FETCH HLI Statement

Syntax

FETCH [ABSOLUTE offset | RELATIVE offset]
... cursor [INTO hostvariablelist]

offset: number

cursor-name: identifier, or host variable

hostvariablelist: list of host variables

Purpose

To retrieve a row from a cursor.

See also

DECLARE, PREPARE, OPEN.

Description

This command repositions a cursor and optionally puts the data from the new row into host variables (see "Host variables with WSQL HLI"). If a host variable list is not used, then the data from the new row can be obtained using the **wsqlgetfield** function.

The named cursor must be open.

The ABSOLUTE clause places the cursor on row number **offset**. The RELATIVE clause places the cursor **offset** rows forward or backward (if negative) in the result set. If both clauses are omitted, then the default behavior is to move the cursor forward one row.

This command is functionally similar to the Embedded SQL command FETCH. See "FETCH statement" in the chapter "Watcom-SQL Statements".

OPEN HLI Statement

Syntax

OPEN cursor-name

cursor-name: identifier, or host variable

Purpose

To open a cursor.

See also

DECLARE, PREPARE, FETCH, CLOSE, DROP.

Description

This command opens a previously declared cursor to access information from a database. The cursor must have been DECLAREd, and the associated statement must have been prepared.

All cursors are opened WITH HOLD, that is, they are left open and at the same row if a COMMIT or ROLLBACK is performed.

When opened, the cursor is positioned just before the first row of the result set.

This command is functionally similar to the Embedded SQL command OPEN. See "OPEN statement" in the chapter "Watcom-SQL Statements".

PREPARE HLI Statement

Syntax

PREPARE statement-name FROM statement

statement-name: identifier, or host variable

statement: string or host variable

Purpose

To prepare and name a SQL statement.

See also

DECLARE, DROP, OPEN, EXECUTE.

Description

This command prepares a statement to be executed or used for a cursor. Any statement previously prepared with the same name is overwritten.

To free resources associated with a prepared statement, use the DROP command.

This command is functionally similar to the Embedded SQL command PREPARE. See "PREPARE statement" in the chapter "Watcom-SQL Statements".

ROLLBACK HLI Statement

Syntax ROLLBACK

Purpose To undo changes made to the database since the last COMMIT or ROLLBACK.

See also COMMIT.

Description This command undoes any changes made to the database since the last COMMIT or ROLLBACK.

This command is functionally similar to the SQL command ROLLBACK. See "ROLLBACK statement" in the chapter "Watcom-SQL Statements" for details.

SET CONNECTION HLI Statement

Syntax SET CONNECTION [connection-name]

connection-name: identifier, or host variable

Purpose To change active connections.

See also START USING DATABASE, STOP USING DATABASE.

Description This command changes the current database connection. You can use the START USING DATABASE command several times naming the connection each time. Each time, the new connection becomes the current connection. You can use the SET CONNECTION statement to make the named connection current.

This command is functionally identical to the Embedded SQL command of the same name. See "SET CONNECTION statement" in the chapter "Watcom-SQL Statements".

START USING DATABASE HLI Statement

Syntax

1. START USING DATABASE [engine-name:database-name]
   ... [AS connection-name] [USER] userid
   ... IDENTIFIED BY password

2. START USING DATABASE connect-string

engine-name: identifier, or host variable

database-name: identifier, or host variable

connection-name: identifier, or host variable

userid: identifier, or host variable

password: identifier, or host variable

Purpose

Form 1: to establish a connection to a database engine or server. Form 2: to start a new database on an existing engine or server.

See also

STOP USING DATABASE, SET CONNECTION.

Description

Form 1 of the START USING DATABASE statement establishes a connection to a database engine or server. If **database** is omitted then WSQL HLI connects to the default database environment. If there is no such database environment then this command fails.

You can optionally name the connection. This is useful if you wish to simultaneously use more than one connection to the same or different database engines or servers.

This command is functionally similar to the Embedded SQL command CONNECT. See "CONNECT statement" in the chapter "Watcom-SQL Statements".

Form 2 of the START USING DATABASE statement with a connection string does not try to connect to this starting database; it merely starts it. The connection string specifies the database. For a description of connection strings, see "Database connection parameters" in the chapter "Connecting to a Database".

To connect to the database, once it is started, use the START USING DATABASE statement a second time, with form 1.

Examples

The following statement starts the sample database on the currently running server or engine **server_name**:

```
START USING DATABASE
"eng=server_name;dbf=c:\sqlany50\sademo.db"
```

The following two statements start and then connect to the sample database on the currently running server or engine **server_name**:

```
START USING DATABASE
"eng=server_name;dbf=c:\sqlany50\sademo.db"

START USING DATABASE server_name:sademo

AS conn1 USER dba IDENTIFIED BY sql
```

STOP USING DATABASE HLI Statement

Syntax

1. STOP USING DATABASE [connection-name]
2. STOP USING DATABASE connect-string

connection-name: identifier, or host variable

Purpose

Form 1: to terminate a connection to a database engine or server. Form 2: to stop a database.

See also

START USING DATABASE, SET CONNECTION.

Description

Form 1 of this statement terminates a connection to the database.

This statement is functionally similar to the Embedded SQL command DISCONNECT. See "DISCONNECT statement" in the chapter "Watcom-SQL Statements". Form 2 of this statement stops, or unloads, a database from a named server.

Examples

The following statement uses form 1 to disconnect from the current database.

```
STOP USING DATABASE
```

The following statement uses form 2 to stop the sample database on the engine **server_name**:

```
STOP USING DATABASE "eng=server_name;dbn=sademo"
```

WSQL HLI and Visual Basic

Visual Basic 3.0 introduced ODBC support. By using ODBC, you can develop a Visual Basic application that uses SQL Anywhere that can easily use another ODBC data source sometime in the future. WSQL HLI is still a good alternative when using ODBC does not make sense for your application.

It is very easy to use DLL's from the Visual Basic environment. All that is necessary is for the functions to be declared in the global code area or a general-declarations area. Since Visual Basic does not support callback functions, the **wsqlregisterfuncs** function cannot be used. Host variables are not used in Visual Basic.

To declare WSQL HLI functions, put the following statements in your project's global code area

```
Declare Function wsqlexec Lib "wsqlcall.dll" (ByVal
cmd$) As Long

Declare Function wsqlgetfield Lib "wsqlcall.dll"
(ByVal cur$, ByVal i As Integer, ind As Integer,
ByVal buf$, ByVal l As Integer) As Long

Declare Function wsqlgetcolumnname Lib
"wsqlcall.dll" (ByVal cur$, ByVal i As Integer, ind
As Integer, ByVal buf$, ByVal l As Integer) As Long

Declare Function wsqllasterror Lib "wsqlcall.dll"
(ByVal buf$, ByVal i As Integer) As Long

Declare Function wsqlquerytomem Lib "wsqlcall.dll"
(ByVal buf$, ByVal cur$, ByVal dta As Integer, ByVal
names As Integer) As Long
```

These declarations can also be found in the Test Application's global area. The filename is WSQLDLLT.GLB.

It is important that all string buffers that are to be filled in by WSQL HLI be **fixed-length strings**. WSQL HLI will not work properly with variable-length strings. Also, all returned character string buffers will be terminated with a null (zero) character.

Visual Basic example

The following code fragments are part of a Visual Basic application that uses WSQL HLI.

In the global module or form, declare the DLL or QuickLibrary functions which are going to be used. Now the five functions can be used in other subroutines, such as the following code fragment:

```
ret = wsqlexec("START USING DATABASE sademo AS Con1
USER dba IDENTIFIED BY sql")
ret = wsqlexec("SET CONNECTION Con1")
    ' We are now connected to the
    ' sample database engine
ret = wsqlexec("PREPARE stmt1 FROM 'SELECT * FROM
department' ")
    ' The single quotes are optional.
ret = wsqlexec("DECLARE cur1 CURSOR FOR stmt1")
    ' The cursor name 'cur1' and the statement name
    'stmt1' are now
    ' reserved.
ret = wsqlexec("OPEN cur1")
    ' The cursor is now open and positioned before
the    ' first row of the query.

ret = wsqlexec( "CLIP BOTH cur1" )

MyTextBox.Text = ClipBoard.GetText( )
    ' MyTextBox now contains the query results.
    ' cur1 is positioned just after the last
    ' row of the query.

' ---------------------------------------------
Dim ret As Long
Dim ind%
Dim buf As String * 100

ret = wsqlexec("FETCH ABSOLUTE 1 cur1")
    ' cur1 is now positioned on the
    ' first row of the query.
    '
    ' buf is a 100 byte string, ind% is an integer:

ret = wsqlgetfield("cur1", 1, ind%, buf, 100)
    ' translation - put the first 100 characters of
the
    ' second column of the current row of cursor
"cur1"   into
    ' the string buf, and if there are more than 100
    ' characters, put how many characters
    ' there are in    ind%.
    ' buf now contains "R & D", ind contains zero.

ret = wsqlgetcolumnname("cur1", 0, ind%, buf, 100)
    ' translation - put the first
    '  100 characters of the name
    ' of the first column of the result
    ' set of cursor "cur1"
    ' into the string buf$, and if there
   are more than 100  characters, put how many
```

```
    ' characters there are in ind%.
    ' buf$ now contains "dept_id", ind% contains
zero.

ret = wsqlexec("CLOSE cur1")
ret = wsqlexec("DROP STATEMENT stmt1")
    ' the cursor and statement are no longer valid.
    '
    ' two ways of executing a statement:
    ' method one:
ret = wsqlexec("EXECUTE IMMEDIATE INSERT INTO
department VALUES ('220','Eastern Sales', 502)")

    ' method two:
ret = wsqlexec("PREPARE stmt2 FROM INSERT INTO
department VALUES ('230','Western Sales', 502)")
ret = wsqlexec("EXECUTE stmt2")
ret = wsqlexec("DROP stmt2")

' handling errors:
ret = wsqlexec("EXECUTE IMMEDIATE DROP TABLE
BogusTable ")
    ' There is no such table - ret is set to -141.
    ' Get more information:
ret = wsqllasterror(buf, 100)
    ' buf now contains "table 'BogusTable' not
found",
    ' a more meaningful error message

ret = wsqlexec("STOP USING DATABASE Con1")
    ' database connection is terminated.
```

The sample application

There is a Visual Basic sample application (both source and executable) included with WSQL HLI that demonstrates some features of WSQL HLI. The filename is WSQLDLLT.EXE, and you can take a look at the source by loading the project WSQLDLLT.MAK from Visual Basic. (Visual Basic is not included with the SQL Anywhere software.)

This example is found in the ACCXMP\VB subdirectory of the SQL Anywhere installation directory (usually C:\SQLANY50).

Running the sample application

The sample application is a simple display utility which shows the contents of the **employee** table, one record at a time. Start the sample database and the test application. Once the database engine and the test application have started, press the **Connect** button. After a moment, a slider bar will appear, the **Connect** button will have become the **Disconnect** button, and the text boxes will be displaying the first record in the **employee** table.

Use the slider bar to examine any of the one hundred entries in the **employee** table. When you are done, press the **Disconnect** button and close the application.

WSQL HLI and REXX

There is a special entry point in the WSQL HLI for OS/2 specifically for REXX that can be called directly from REXX and allows REXX variables to be used in SQL commands as host variables (see "Host variables with WSQL HLI" on page 782). This function must be registered by any REXX program that uses WSQL HLI. Put the following statement in your REXX program:

```
call RXFUNCADD 'SQLEXEC', 'WSQLCAL2', 'WSQLEXECREXX'
```

WSQL HLI is very similar to the IBM DB/2 REXX programming interface. START USING DATABASE and STOP USING DATABASE are supported by the **wsqlexecrexx** entry point. You may wish to register the function a second time to make your REXX program more portable:

```
all RXFUNCADD 'SQLDBS', 'WSQLCAL2', 'WSQLEXECREXX'
```

WSQL HLI only contains a subset of the IBM DB/2 REXX programming interface.

Error codes and messages

The REXX interface function returns an error code. The error code is also assigned to the REXX stem variable **SQLCA.SQLCODE**. See "SQL Anywhere Database Error Messages" for a complete list of error codes. An error message is assigned to the REXX variable **SQLMSG**.

REXX examples

The following short REXX program fetches 10 employees from the sample database using WSQL HLI.

```
* WSQL HLI from REXX */
call RXFUNCADD 'SQLDBS', 'WSQLCAL2', 'WSQLEXECREXX'
call RXFUNCADD 'SQLEXEC', 'WSQLCAL2', 'WSQLEXECREXX'
call sqldbs 'START DATABASE MANAGER'
call sqldbs 'START USING DATABASE SADEMO USER DBA
IDENTIFIED BY SQL'
stmt = "select emp_lname from employee"
call sqlexec "PREPARE s1 FROM :stmt"
call CheckSQLError
call sqlexec "DECLARE c1 CURSOR FOR s1"
call CheckSQLError
call sqlexec "OPEN c1"
call CheckSQLError
do 10
```

```
/* fetch a row */
call sqlexec "FETCH c1 INTO :lname:Indlname"
call CheckSQLError
say lname Indlname
end
call sqlexec "CLOSE c1"
call CheckSQLError
call sqldbs "STOP USING DATABASE ALL"
exit
CheckSQLError:
if SQLCA.SQLCODE = 0 then do
   say SQLCA.SQLCODE SQLMSG
end
return
```

There is a more substantial example in the ACCXMP\REXX subdirectory of the SQL Anywhere installation directory (usually C:\SQLANY50).

Index

Special Characters

@@char_convert 557
@@client_csid 557
@@client_csname 557
@@connections 557
@@cpu_busy 557
@@error 557, 904
@@identity 557, 904
@@idle 557
@@io_busy 557
@@isolation 557, 904
@@langid 557
@@language 557
@@max_connections 557
@@maxcharlen 557
@@ncharsize 557
@@nestlevel 557
@@pack_received 557
@@pack_sent 557
@@packet_errors 557
@@procid 557, 904
@@rowcount 557, 904
@@servername 557, 904
@@spid 557
@@sqlstatus 557, 904
@@textsize 557
@@thresh_hysteresis 557
@@timeticks 557
@@total_errors 557
@@total_read 557
@@total_write 557
@@tranchained 557
@@trancount 557, 904
@@transtate 557
@@version 557, 904

2

2000 930

A

abort a command 96
ABS function 938
ACOS function 938
ActiveReq 959
administering SQL Remote 427, 455
administrator role
  SQL Anywhere compatibility 540
ADS
  support for 660
aggregate functions 565, 936
  AVG 122
  COUNT 122
  LIST 122
  MAX 122
  MIN 122
  SUM 122
Alias property 963
aliases
  correlation name for tables 115
  for columns 1142
  in the DELETE statement 1045
ALL
  conditions 147, 578, 912
  keyword in SELECT statement 1142
ALL permissions 382
alloc_sqlda 709
alloc_sqlda_noind 709
allocating disk space 972
ALLOW_NULLS_BY_DEFAULT option 546, 1157
alphabetical order 103
ALTER DBSPACE statement
  syntax 972
  TRANSLOG Clause 972
ALTER permissions 382
ALTER PROCEDURE statement
  syntax 974
ALTER PUBLICATION statement
  syntax 975
ALTER REMOTE MESSAGE TYPE statement
  syntax 976

ALTER TABLE statement
and concurrency 260
and foreign keys 241
and integrity 229
CHECK conditions 235
column defaults 230, 231
examples 212
syntax 976
ALTER TRIGGER statement
syntax 982
ALTER VIEW statement
syntax 983
altering
dbspaces 972
publications 975
tables 976
AND conditions 913
AND keyword 107
ANSI
COMMIT behavior 1152, 1154
conformance 893, 1162
delete permissions 1159
integer overflow behavior 1158
NULL behavior 1159
update permissions 1159
variable behavior 1158
ANSI_BLANKS option 1157, 1158
ANSI_INTEGER_OVERFLOW option 1158
ANSI_PERMISSIONS option 1159
ANSINULL option 1159
ANY
conditions 147, 578, 912
Any outgoing messages will not be identified with a 1135
apostrophes
in SQL 215
using 105
application development systems 26
architecture
about 23, 25, 29, 32
basic 25
mixed operating systems 32
multiuser 29
network server 29
single-user 25
standalone 25
ARGN function 965
arithmetic expressions 899
arithmetic operators 563
articles
system table for 1331, 1332
ASCII 843
ASCII file format 1165, 1167
ASCII function 941
ASIN function 938
assertion failed error 1296
assessing permissions 396
Async2Read property 955, 959
AsyncRead property 955, 959
AsyncWrite property 955, 959
ATAN function 939
ATAN2 function 939
atomic compound statements 285
and COMMIT statement 307
and EXECUTE IMMEDIATE statement 307
attributes 182
audit trail 404
authorization 663, 893
auto_commit
ISQL option 1165
AutoCAD
support for 660
autoincrement 554
about 1021
default 228, 230, 232
maximum value 232
primary key values 1063
when to use 259
automatic joins
and foreign keys 1311
SELECT statement 1142
AUTOMATIC_TIMESTAMP option 546, 1157
Autostop connection parameter 162, 163
average 936
AVG function 122, 936

B

B+ tree
and indexes 327
BACKGROUND_PRIORITY option 1150, 1151
backing up databases 63, 588
backup functions
about 711
backup utilities
options 826
backups

about 399, 411, 826
and SQL Remote 498
connection parameters 826
database only 827
DUMP DATABASE Transact-SQL statement 588
DUMP TRANSACTION Transact-SQL statement 588
Embedded SQL functions 711
for remote databases 504
for replication 503, 504
full 411
live 415, 827
LOAD DATABASE Transact-SQL statement 588
LOAD TRANSACTION Transact-SQL statement 588
online 826
options 826
rename and start new transaction log 827
running database 63
with Sybase Central 63
balanced indexes 327
base tables 184
batches
about 265, 281
and control statements 281, 283
and data definition statements 281
statements allowed in 312
Transact-SQL overview 613
writing 613
beep 1165
BEGIN keyword 283, 614
syntax 995
BEGIN TRANSACTION statement 582
bell
ISQL option 1165
BETWEEN conditions 110, 577, 909
binary data
GET DATA statement 1081
BINARY data type 668, 926
Transact-SQL 551
binary data types 926
Transact-SQL 551
binary large objects
about 1082
bind variables
about 683
DESCRIBE statement 1049
EXECUTE statement 1062
OPEN statement 1113
BIT data type
Transact-SQL 551
bitmaps 200
bitwise operators 564
blanks
ANSI behavior 1158
BLOBS 200
about 1082
GET DATA statement 1082
GET DATE statement 1081
replication 492
block fetches 1070
OPEN statement 1113
BlockedOn property 955
blocking
and deadlock 254
database option 1151
transaction 254
BLOCKING option 1150
Borland C++
support for 656
break key 96
BREAK statement
Transact-SQL 622
browsing databases
and isolation levels 256
B-tree 1338, 1347
bulk operations 373, 817
BYTE_LENGTH function 941
BYTE_SUBSTR function 942

C

C
data types 668
interfaces for 653
programs 656
C Set ++ 656
cache 814, 817
and low memory 320
and performance 319
size 817
cache buffers 817
CacheHits property 959
CacheRead property 955, 959
CacheReadIndInt property 955, 959

CacheReadIndLeaf property 955, 959
CacheReadTable property 955, 959
CacheWrite property 955, 959
CALL statement
  about 265
  and parameters 288
  and Transact-SQL 617
  examples 269
  syntax 984
callback function 782
  in Windows 3.x 716
capabilities
  supported 648
CASCADE 243, 1025
cascading
  deletes 243
  updates 243
case sensitivity 101, 105, 360, 841, 896, 909, 910
  and pattern matching 910
  in the catalog 1339
  Transact-SQL compatibility 547
CASE statement 283
  syntax 986
CAST function 570, 929, 951
catalog
  diagram 1330
  SQL Server compatibility 540
  system tables 1329
  system tables list 1331
catalog procedures
  sp_column_privileges 602
  sp_columns 602
  sp_fkeys 602
  sp_pkeys 602
  sp_special_columns 602
  sp_sproc_columns 602
  sp_stored_procedures 602
  sp_tables 602
  Transact-SQL 601
CD-ROM 42
CEILING function 939
century issues 930
CHAINED option 1160
chained transaction mode 582, 1160
CHAR data type 918
  and host variables 1158
  Transact-SQL 550
CHAR function 942
CHARACTER data type 918
  Transact-SQL 550
character data types 157
  Transact-SQL 550
character sets
  about 347
  and ODBC 349
  Chinese 352
  choosing 354
  Japanese 352
  Korean 352
  multibyte 352
  Shift-JIS 352
  storage 918
  Taiwanese 352
  unicode 352
  UTF8 352
character strings 894
CHARACTER VARYING data type 918
CHECK conditions 235
  about 225, 227, 1025
  deleting 238
  modifying 238
  on columns 228, 235
  on tables 228, 237
  Transact-SQL 585
  user-defined data types 236
CHECK ON COMMIT clause
  and referential integrity 1025
CHECKPOINT
  checkpoint log 988
checkpoint log 401
CHECKPOINT statement
  and recovery 401
  syntax 988
CHECKPOINT_TIME option 1150, 1152
checkpoints
  CHECKPOINT_TIME option 1152
  scheduling of 402
  urgency of 402
Chinese character set 352
Chkpt property 959
ChkptFlush property 959
ChkptPage property 959
CLEAR ISQL command 848
Client
  defined 31
  SQL Anywhere 29
client applications

about 25, 30
and databases 166
SQL Anywhere programs 36, 40
Sybase Central 39
using 26
CLOSE statement
about 678
in procedures 295
syntax 989
CLOSE_ON_ENDTRANS option 1160
COALESCE function 965
code editor 57
code pages 354, 843
and data storage 918
col_length SQL Server function 573
col_name SQL Server function 573
cold links 767
collation file format 357
collation sequences 830, 843
collations 354
about 347
and pattern matching 910
collation file format 357
custom 357, 358
file format 358
multibyte 352
column defaults
about 231
column names
long 1051
columns 259
about 101, 182
adding 213
adding using Sybase Central 52
aliases 1142
altering constraints in Sybase Central 237
altering defaults in Sybase Central 231
altering definition 978
and user-defined data types 236, 928
changing 213
changing heading name 1142
CHECK conditions 228, 235, 585
choosing data types for 199
choosing names 199
constraints 200, 227, 228, 235, 928, 1023
creating 52
creating constraints in Sybase Central 237
creating defaults in Sybase Central 231
defaults 227, 230, 231, 259, 585
deleting 213
deleting constraints in Sybase Central 237
designing 199
drag and drop 54
dropping defaults in Sybase Central 231
identity 554
in publications 471
in the system tables 1334
list of 78, 95
making a primary key with Sybase Central 53
name 691
naming 215, 897, 901
ordering 104
permissions on 1333
renaming 213, 980
rules 585
selecting from a table 104
shortening 213
SYSCOLUMNS system view 1357
timestamp 552
comma delimited files 1165, 1167
command delimiter
setting 309, 1165
command echo 1165
command files
about 1126
and ISQL 205
building 152
Command window 152
overview 152
parameters 154, 1126
command line summary 808
command line utilities and Unix 809
command recall 94
Command window 87
ISQL 72
command-line switches
about 816
and services 520
for server 816
in configuration file 816
commands
editing in ISQL 76
executing in ISQL 72
getting 152
getting in ISQL 72
interrupting in ISQL 79
loading 152
loading in ISQL 72

previous 76
recalling in ISQL 76
saving 153
saving in ISQL 72
stopping in ISQL 79
COMMENT statement
syntax 991
comments
about 153, 915
comment indicators 915
indicators 580
Commit property 955
COMMIT statement
about 131
and procedures 308
and transactions 246
in compound statements 285
syntax 993
Transact-SQL 584
COMMIT_ON_EXIT
ISQL option 1165
CommitFile property 959
commits
COOPERATIVE_COMMIT_TIMEOUT option 1152
COOPERATIVE_COMMITS option 1152
CREATE TABLE statement 212
DELAYED_COMMIT_TIMEOUT option 1154
DELAYED_COMMITS option 1154
DROP TABLE statement 214
CommLink property 955
CompanyName property 959
comparing dates and times 925
comparison conditions
about 908
Transact-SQL 576
comparison operators 577, 908
comparisons
about 105, 106, 907
using subqueries 145
compatibility
of SQL Anywhere and SQL Server databases 544
QUERY_PLAN_ON_OPEN 1161
compatibility of SQL 606
compile and link process 657, 658
compilers supported 656
components 808
compound search conditions 107
compound statements 283, 614
about 995
and COMMIT statement 285
and ROLLBACK statement 285
atomic statements 285
declarations in 284
SQL statements allowed in 286
using 283
compressed databases 874
COMPUTE clause
Transact-SQL 593
concatenating strings 563, 905
concurrency 248
about 247, 248, 259
and data definition 259
and data definition statements 260
and locks 248
and performance 249
and primary keys 259
consistency 248
inconsistency 248
conditions 108
about 907
and GROUP BY clause 125
search 105, 106, 110
configuration
file, for database engine 816
CONFIGURE statement
syntax 998
configuring an Open Server 642
conflict resolution in SQL Remote 507
conflicts
cyclical blocking 255
locking 254
reporting in SQL Remote 507
resolution of 263
resolving in SQL Remote 507, 509, 510
VERIFY_ALL_COLUMNS option 510
ConnCount property 963
connect
granting 394
permission 380
CONNECT statement 733
and Embedded SQL 662
example 140
syntax 999
connecting 84
to a database 70, 161

Connecting to a database
using Sybase Central 48
using the default password 48
using the default user ID 48
connection properties
displaying 1321
connection_property function 953
connections
about 162
dropping 820
parameters 162, 163, 166, 173
properties of 955
same machine 26
string 162
consistency 248
during transactions 248
consolidate permissions 1089, 1134
granting 466
revoking 466
consolidated database 1134
publishing 1135
consolidated databases 422
setting up 433, 442
constant expressions
defaults 234
constants
in expressions 900
Transact-SQL 561
constraints 1023
about 225, 227, 978
containers 49
CONTINUE statement
Transact-SQL 622
ContReq property 959
control statements 606, 614
about 265, 995
BEGIN keyword 283
CALL statement 984
CASE statement 283, 986
compound statements 283
END keyword 283
IF statement 283, 1095
LEAVE statement 1104
LOOP statement 283, 1108
Transact-SQL BREAK statement 622
Transact-SQL CONTINUE statement 622
Transact-SQL GOTO statement 618
Transact-SQL IF statement 618
Transact-SQL RETURN statement 621
Transact-SQL WHILE statement 622
WHILE statement 283, 1108
conventions
documentation 9
conversion errors 372
CONVERSION_ERROR option 1157
CONVERT function 570, 951
converting ambiguous strings 932
converting data types 951
COOPERATIVE_COMMIT_TIMEOUT option
1150, 1152
COOPERATIVE_COMMITS option 1150, 1152
copyright
retrieving 1327
correlated subqueries
defined 148
correlation names
about 115
defined 148
in the DELETE statement 1045
COS function 939
COT function 939
COUNT function 121, 122, 123, 218, 936
CREATE DATABASE statement
Transact-SQL 588
CREATE DATATYPE statement
syntax 1002
CREATE DBSPACE statement
about 207
preallocating space 209
syntax 1004
CREATE DEFAULT statement
Transact-SQL 585
CREATE DOMAIN statement
syntax 1002
CREATE FUNCTION statement
about 273
syntax 1005
CREATE INDEX statement
and concurrency 260
and table use 1008
example 326
syntax 1007
Transact-SQL 585
CREATE MESSAGE statement
Transact-SQL 586
CREATE PROCEDURE statement
examples 268
parameters 287

syntax 1009
Transact-SQL 615
CREATE PUBLICATION 470, 471
CREATE PUBLICATION statement
syntax 1013
CREATE REMOTE MESSAGE TYPE statement
syntax 1015
CREATE RULE statement
Transact-SQL 585
CREATE SCHEMA statement
in batches 1017
syntax 1017
Transact-SQL 587
CREATE SUBSCRIPTION 483, 487
CREATE SUBSCRIPTION statement
syntax 1019
CREATE TABLE statement
and command files 205
and concurrency 260
and foreign keys 241
and integrity 229
column defaults 230
examples 1329
primary and foreign keys 214
syntax 1020
Transact-SQL 587
CREATE TRIGGER statement
about 277
and integrity 229
syntax 1028
Transact-SQL 616
CREATE VARIABLE statement
syntax 1031
CREATE VIEW statement
about 138
examples 1355
syntax 1033
WITH CHECK OPTION clause 219
creating 470
data types 927, 1002
databases 588, 841
functions 1005
groups 60
indexes in Sybase Central 224
publications 1013
stored procedures 1009
subscriptions 1019
users 61
users in Sybase Central 380
views in Sybase Central 218
creating publications 436, 444, 470
creating subscriptions 436, 444, 483
creator 897
critical device errors
processing 719
cross product
and joins 114
CS_CSR_ABS 648
CS_CSR_FIRST 648
CS_CSR_LAST 648
CS_CSR_PREV 648
CS_CSR_REL 648
CS_DATA_BOUNDARY 648
CS_DATA_SENSITIVITY 648
CS_PROTO_DYNPROC 648
CS_REG_NOTIF 648
CS_REQ_BCP 648
Ctrl+Break 96
in Embedded SQL 715
processing in DOS 719
current date
default 232
CURRENT DATE special constant 900
CURRENT PUBLISHER 435, 443, 900, 1091
current query 978, 1115
CURRENT REMOTE USER 510
CURRENT TIME special constant 900
current timestamp
default 232
CURRENT TIMESTAMP special constant 900
CURRENT USER
special constant 900
CurrIO property 959
CurrRead property 959
CurrTaskSwitch property 955
CurrWrite property 959
CursorOpen property 955
cursors
and FOR statement 299
and LOOP statement 297
and passthrough mode 515
and transactions 1160
closing 989, 1160
cursor stability 251
cursor types 1039
declaring 284, 1039
DESCRIBE 1049
example C code 730

EXPLAIN statement 1066
fetching 1068
in Embedded SQL 678
in ODBC 756
in procedures 295, 297
in triggers 295
looping over 1073
on SELECT statements 297
OPEN statement 1112
opening 1161
WITH HOLD clause 1112
Cursors property 955
curunreservedpgs SQL Server function 573
custom collations 358

D

data
and locks 248
consistency 248
deleting 227
duplicated 226, 227
exporting 368
invalid 226, 227
loading large amounts 320
modifying 227
updating 227
data definition
and concurrency 259
data definition statements
and concurrency 260
data entry
and isolation levels 256
data integrity 244
data normalization 191
data sources 169
about 173, 174, 179
adding 174
connection keywords 163
connection overview 162
modifying 179
removing 179
working with 173
data type conversion functions 570
data types 684, 691
about 182, 917
and OmniCONNECT support 650
binary 551, 926
bit 551
C data types 668
character 550, 918
choosing 199
converting 570, 929, 951
creating 1002
data and time 552
date 922
date and time 922
decimal 550
defined 897
dropping user-defined 1053
in the system tables 1335, 1353
integer 549
money 551
numeric 920
time 922
timestamp 552
Transact-SQL 549, 550, 551, 552
user-defined 555, 927, 1002, 1353
data_pgs SQL Server function 573
database containers
expanding with Sybase Central 49
database design
about 191
and performance 320
normalizing data 191
verifying 197
verifying a design 197
database engines
about 26, 36, 42, 263, 814
and portable computers 263
and updating databases 263
command line 814
connection 27
deployment 263
operating systems 30
runtime edition 37
starting 1173
stopping 42, 1176
unlimited runtime license 263
-v command line switch 263
Database extraction utilities 859
database files
adding 207
allocating space for 209
creating 1004
dbspaces 1004
erasing 835

storing indexes in 1007
uncompressing 874
database information 838
database management
using Sybase Central 39, 45
database objects
commenting 991
database options
ALLOW_NULLS_BY_DEFAULT 546
AUTOMATIC_TIMESTAMP 546
DATE_ORDER 923
QUOTED_IDENTIFIER 546, 562
set by Open Server Gateway 644
database pages
caching 320
displaying size of 837
database properties
changing 45
displaying 1321
database schema
about 1329
viewing with Sybase Central 49
database security 524
database servers
about 36
operating systems 30
database threads
blocked 255
database utilities
and database connections 168
DBBACKUP 42
DBERASE 42
DBINFO 42
DBINIT 42
DBLOG 42
DBSHRINK 42
DBSTOP 42
DBUNLOAD 42
DBUPGRAD 42
DBVALID 42
DBWRITE 42
ISQL 40
log translation 404
REBUILD 42
Wizards 63
DatabaseFile connection parameter 162, 163
DatabaseName connection parameter 162, 163
databases
about 42
administering 45
administrator, defined 377
alias 31
and portable computers 262
and write files 263
backing up 42, 63
building 203
cache size 814
checking validity of 42
compacting 42
compressed 42, 874
compressing 263
configuring for Transact-SQL 544
conflict resolution 263
connecting to 48, 161, 162, 166
consolidated 422
copying 262
creating 42, 203, 206, 841
creating Transact-SQL-compatible 544
data integrity 225, 226
designing 181, 185, 187
devices 27
erasing 42, 209, 835
files 27, 31, 1336
ignoring trailing blanks 544
information 837, 838
initializing 42, 206, 841
initializing for Transact-SQL 544
integrity constraints 227
large 263
library memory 718
loading data into 1105
maintaining 42
managing 206
managing 45
managing using Sybase Central 39
memory 719
multiple 27, 30
multiple-file 28, 30, 207, 209
name 31
objects 184
page usage 837
permissions 376
planning 185
problems updating 262
properties of 963
read-only 42
rebuilding 42, 857
relational 182

remote 422, 426, 438, 445
schema 1329
SQL Anywhere 27
starting 814, 816, 1172
stopping 867, 1175
storage on disk 209
structure 1329
synchronizing 445, 446
system procedures 1319
system tables 1329
Transact-SQL compatibility 544
Transact-SQL CREATE DATABASE 588
Transact-SQL DROP DATABASE 588
Transact-SQL DUMP DATABASE 588
Transact-SQL LOAD DATABASE 588
unloading 42, 878
unloading tables from 1182
updates 263
updating 262
updating from a transaction log 262, 263
upgrading 7, 42, 882, 883
using ISQL to manage 205
using Sybase Central 204
validating 411, 886
working with compressed 263
write files 890
DATALENGTH function 954
datalength SQL Server function 573
DataSourcename connection parameter 162, 163
date and time data types
Transact-SQL 552
date and time functions 568
Transact-SQL 568
DATE data type 922, 946
Transact-SQL 552
date format
database option 1152
DATE function 946
date order
database option 1153
date to string conversions 933
DATE_FORMAT option 1150
DATE_ORDER option 923, 1150
and ODBC 923
dateadd Transact-SQL function 568
DATEFORMAT function 946
DATENAME function 947
datename Transact-SQL function 568
datepart Transact-SQL function 568
dates 106, 110, 157, 922, 946, 947, 948, 949, 950
ambiguous string conversions 932, 933
combining 107
comparing 925
compound 107
conversion problems 933
formatting 946
interpreting strings as dates 923
unambiguous specification of 923
year 2000 930
DATETIME data type 668
Transact-SQL 552
DATETIME function 946
DAY function 946
DAYNAME function 947
DAYS function 947
db_abort_request 715
db_backup 711
db_break_handler 721
db_build_parms 708
db_cancel_request 715
db_catch_break 721
db_catch_critical 721
db_delete_file 714
db_destroy_parms 708
db_find_engine 707
db_fini 702
db_finished_request 720
db_free_parms 708
db_get_sqlca 718
DB_ID function 954
db_id SQL Server function 573
db_init 702
db_is_working 715
DB_NAME function 954
db_name SQL Server function 573
db_parms_connect 708
db_parms_disconnect 708
db_process_a_message 717
db_property function 954
db_register_a_callback 716
db_release_break 721
db_release_critical 721
db_sending_request 720
db_set_sqlca 718
db_start 708
db_start_database 706
db_start_engine 705

db_stop 709
db_stop_database 706
db_stop_engine 707
db_string_connect 704
db_string_disconnect 704
db_working 715
DBA
  defined 377
DBA authority
  about 376
  and permissions 386
  granting 381
  in the system tables 1352
  not inheritable 386
DBAlloc 718
dBASE file format 1165, 1167
DBBACKUP 42, 411, 826
  command line 826
DBCLIENT 29, 32, 37
DBCLIENW 29
DBCOLLAT 357, 358, 830
DBENG50S 37
DBENG50W 37
DBERASE 42, 209, 835
DBEXPAND 874
DBFree 719
DBINFO 42, 320, 838
DBINIT 42, 206, 320, 360, 841
  ignoring trailing blanks 544
DBL50T.DLL 172
DBL50W.DLL 172
DBLOG
  command line 871
DBNumber property 955
dbo user 545, 598
  system objects 858, 876
DBOS50 643, 853
DBOSINFO 855
DBOSSTOP 856
DBRealloc 719
dbremote 447
  about 496, 863
  and security 497
  command line 863
DBSHRINK 42, 263, 833
DBSPACE 1336
dbspaces
  altering 972
  creating 207, 1004
  dropping 1053
  managing 589
  preallocating space 209
DBSTOP 42, 867
DBTOOL statement
  creating databases 207
  syntax 1035
DBTRAN 262, 263, 404, 418
  command line 850
DBUNLOAD 42, 369, 878
  and replication 505
DBUPGRAD 7, 42, 883
DBVALID 42, 411, 886
DBWATCH
  and services 531
DBWRITE 42, 263, 890
DBXTRACT 445, 484
  about 859
  command line 859
DDE
  support for 654
deadlock
  and transaction blocking 254
deadlocks
  reasons for 255
DECIMAL data type 668, 920
  Transact-SQL 550
decimal data types
  Transact-SQL 550
decimal precision
  database option 1155
decision support
  and isolation levels 256
DECL_BINARY 668
DECL_DECIMAL 668
DECL_FIXCHAR 668
DECL_VARCHAR 668
declaration section 1038
  about 667
declarations 284
DECLARE CURSOR statement
  syntax 1039
DECLARE statement
  about 678
  and compound statements 614
  in procedures 295, 301
  syntax 995
  Transact-SQL compatibility 614
DECLARE TEMPORARY TABLE statement

syntax 1043
defaults
about 227
and foreign keys 242
autoincrement 228, 232, 1021
column 230
column, and Sybase Central 231
constant expressions 234
creating 230
current date 227, 231
current time 231
current timestamp 231
NULL 233
string and number 233
Transact-SQL 585
user 1021
user ID 232
with transactions and locks 259
DEGREES function 939
DELAYED_COMMIT_TIMEOUT option 1150, 1154
DELAYED_COMMITS option 1154
DELETE permissions 382
DELETE positioned statement
syntax 1047
DELETE statement
about 132
and errors 134
and integrity 227, 243
examples 134
syntax 1045
Transact-SQL 589
delete tables 56
DELETE_OLD_LOGS
replication option 1163
DELETE_OLD_LOGS option 504
resetting truncation offset 871
transaction log options 871
deletes
ANSI behavior 1159
Transact-SQL permissions 1159
deleting
services 521
deleting all rows from a table 1180
dependencies
of services 528, 529
deployment 170, 262
on laptop computers 262, 263
SQL Remote 494
DESCRIBE statement 685
long column names 1051
syntax 1049
descriptor
EXECUTE statement 1062
FETCH statement 1068
OPEN statement 1112
PREPARE statement 1120
PUT statement 1124
UPDATE (positioned) statement 1186
designing publications 475, 476, 477, 478, 479
device errors 719
devices
managing 589
recovery from failure of 416
DIF file format 1167
DIFFERENCE function 942
digits
maximum number 1155
dirty reads 248, 251
DISCONNECT statement
syntax 1052
disk full
error writing to transaction log 406
DiskRead property 955, 959
DiskReadIndInt property 955, 959
DiskReadIndLeaf property 955, 959
DiskReadTable property 955, 959
disks
access 817
fragmentation and performance 320
managing 209, 320, 589
recovery from failure of 416
DiskSyncRead property 955, 959
DiskSyncWrite property 955, 959
DiskWaitRead property 955, 959
DiskWaitWrite property 955, 959
DiskWrite property 955, 959
DISTINCT keyword 1142
distributed database systems 260
divide by zero
error 1160
DIVIDE_BY_ZERO_ERROR option 1160
DLL entry points 701
DLLs
calling from functions 314
calling from procedures 314
documentation
about 9

conventions 9
formats 9
domains *See* user-defined data types
DOS
and SQL Remote 457
ISQL 81
ISQL Statistics window 323
memory allocatoin 718
DOUBLE data type 920
Transact-SQL 550
DOW function 947
DOWN ISQL command 848
drag and drop
columns 54
DROP CONNECTION statement
syntax 1055
DROP DATABASE
Transact-SQL 588
DROP DATATYPE statement
syntax 1053
DROP DBSPACE statement
syntax 1053
DROP DOMAIN statement
syntax 1053
DROP FUNCTION statement
syntax 1053
DROP INDEX statement
syntax 1053
DROP OPTIMIZER STATISTICS statement
about 324
syntax 1056
DROP PROCEDURE statement
syntax 1053
DROP PUBLICATION 473
DROP PUBLICATION statement
syntax 1057
DROP REMOTE MESSAGE TYPE statement
syntax 1058
DROP statement
and concurrency 260
dropping indexes 224
syntax 1053
DROP STATEMENT statement
syntax 1059
DROP SUBSCRIPTION statement
syntax 1061
DROP TABLE statement
example 214
syntax 1053
DROP TRIGGER statement
about 279
syntax 1053
DROP VARIABLE statement
syntax 1060
DROP VIEW statement
about 139
example 221
syntax 1053
dropping
databases 588
indexes in Sybase Central 224
publications 1057
subscriptions 1061
users 1133
views in Sybase Central 222
dropping publications 473
dropping users
consolidate 1134
DT_STRING 710
DUMMY table 1331
DUMP DATABASE statement
Transact-SQL 588
DUMP TRANSACTION statement
Transact-SQL 588
duplicated data
as invalid data 226
dynamic cursors
example 738
DYNAMIC SCROLL cursors 1039
dynamic SQL 682

E

EBCDIC 843
echo
ISQL option 1165
editing commands 94, 105
editor 91
ELSE
IF expression 906
e-mail 423
MAPI link 429
SMTP link 429
system procedures 1323, 1324, 1325
VIM link 429
Embedded SQL
about 653, 656

authorization 893
character strings 894
command summary 727
compile and link process 657, 658
cursors 678, 730
dynamic 682
dynamic cursors 738
fetching data 677
functions 701, 718
host variables 667
import libraries 660
interface 34
line numbers 893
memory usage 718
static 682
encryption 841
END keyword 283, 614
syntax 995
ENDIF
IF expression 906
engine name
defined 31
engine properties
displaying 1321
EngineName connection parameter 162
engines
properties of 959
entities
about 185
entity integrity 1310
about 228
enforcing 239
Entity-Relationship design
overview 185
environment variables 168, 808, 810
Data Sources 169
SQLANY 810
SQLCONNECT 168, 810
SQLREMOTE 811
SQLSTART 811
TMP 812
erasing databases 209, 835
error
buffer 674
messages
Embedded SQL function 714
strings 674
errors
codes 1191, 1201
conversion 372
divide by zero 1160
error messages 1210
handling in ISQL 1167
in procedures 300
in Transact-SQL 620
in triggers 300
messages 1191
RAISERROR statement 620
reporting in SQL Remote 507
SIGNAL statement 1171
SQLSTATE values 1201
user-defined messages 1351
escape character
INPUT statement 1098
OUTPUT statement 1115
ESTIMATE function 965
ESTIMATE_SOURCE function 966
estimates
for search conditions 579
in queries 335
providing 334
row-count estimates 1155
exception handlers
declaring 284
using in procedures and triggers 304
EXCEPTION statement
syntax 995
exceptions
declaring 301
exclusive locks 250
EXEC SQL
Embedded SQL 662
executable files 808
EXECUTE IMMEDIATE statement
and atomic compound statements 307
in procedures 307
syntax 1064
EXECUTE statement 683
syntax 1062
Transact-SQL 617
executing commands 92
execution plan
ISQL statistics window 324
EXISTS conditions 578, 912
EXIT statement
syntax 1065
EXP function 939

expanding
viewing 49
EXPERIENCE_ESTIMATE function 966
EXPLAIN statement
syntax 1066
exporting data 364, 366, 1115, 1167
performance 373
SELECT statement 1141
expressions 899
Transact-SQL 561
ExtendDBWrite property 959
ExtendTempWrite property 959
external procedures 314
extraction utility 484

F

failure
recovery from 399
FALSE condition 913
far pointers
and Embedded SQL 701
Feb 29 932
FETCH
multi-row 686
wide 686
FETCH statement 685
about 678
in procedures 295
syntax 1068
fetching data
in Embedded SQL 677
file management 320
FILE message type
about 976
FILE messages
about 1015, 1058
File property 963
file system 817
files
dbspaces 1004
FileVersion property 963
fill_s_sqlda 710
fill_sqlda 709
FIPS
conformance 893, 1162
FIRE_TRIGGERS option 1157, 1160
FIXCHAR data type 668
FIXED file format 1165, 1167
FLOAT data type 920
Transact-SQL 550
FLOOR function 939
FOR BROWSE clause
Transact-SQL 593
FOR READ ONLY clause
Transact-SQL 593
FOR statement
in procedures 299
syntax 1073
FOR UPDATE clause 1070
Transact-SQL 593
foreign keys
about 183
and inserts 133
and integrity 241, 1310
and performance 325
and query execution 328
and referential integrity 241
and tables 117
as invalid data 226
choosing 201
defined 183
in the system tables 1336, 1337, 1338
integrity constraints 1024
mandatory 241
optional 241
role names 1024
system views 1357
unnamed 1024
using to improve performance 322
foreign table
in the system tables 1337
format 1165
forward log 403
FoxPro file format 1165, 1167
fragmentation
and peformance 319
free_filled_sqlda 710
free_sqlda 710
free_sqlda_noind 710
FreeWriteCurr property 959
FreeWritePush property 959
frequency
of sending messages 1089, 1092
FROM clause
and joins 115
SELECT statement 1142

syntax 1074
full backups
about 826
defined 411
performing 411
FullCompare property 955, 959
full-text search 66
functions 935
ABS function 938
ACOS function 938
aggregate 121, 565, 936
ARGN function 965
ASCII function 941
ASIN function 938
ATAN function 939
ATAN2 function 939
AVG function 936
BYTE_LENGTH function 941
BYTE_SUBSTR function 942
CAST function 951
CEILING function 939
CHAR function 942
COALESCE function 965
col_length SQL Server function 573
col_name SQL Server function 573
connection_property function 953
CONVERT function 951
COS function 939
COT function 939
COUNT 218
COUNT function 936
creating 1005
curunreservedpgs SQL Server function 573
data type conversion 570, 951
data_pgs SQL Server function 573
DATALENGTH function 954
datalength SQL Server function 573
date and time 568
date and time Transact-SQL 568
DATE function 946
dateadd Transact-SQL function 568
DATEFORMAT function 946
datename Transact-SQL function 568
datepart Transact-SQL function 568
DATETIME function 946
DAY function 946
DAYNAME function 947
DAYS function 947
DB_ID function 954
db_id SQL Server function 573
DB_NAME function 954
db_name SQL Server function 573
db_property function 954
DEGREES function 939
DIFFERENCE function 942
DOW function 947
dropping 1053
Embedded SQL 701
ESTIMATE function 965
ESTIMATE_SOURCE function 966
EXP function 939
EXPERIENCE_ESTIMATE function 966
external 314
FLOOR function 939
getdate Transact-SQL function 568
host_id SQL Server function 573
host_name SQL Server function 573
HOUR function 948
HOURS function 948
IFNULL function 966
index_col SQL Server function 573
INDEX_ESTIMATE function 966
INSERTSR function 942
ISNULL function 966
LCASE function 942
lct_admin SQL Server function 573
LEFT function 942
LENGTH function 942
LIST function 937
LOCATE function 942
LOG function 939
LOG10 function 939
LTRIM function 942
MAX function 937
MIN function 937
MINUTE function 948
MINUTES function 948
MOD function 939
MONTH function 948
MONTHNAME function 948
MONTHS function 948
next_connection function 954
next_database function 954
NOW function 949
NUMBER function 966
numeric 565, 938
object_id SQL Server function 573
object_name SQL Server function 573

PATINDEX function 943
PI function 939
PLAN function 967
POWER function 939
proc_role SQL Server function 573
property function 954
property_description function 954
property_name function 954
property_number function 954
QUARTER function 949
RADIANS function 939
RAND function 939
REMAINDER function 939
REPEAT function 943
reserved_pgs SQL Server function 573
RIGHT function 943
ROUND function 939
rowcnt SQL Server function 573
RTRIM function 943
SECOND function 949
SECONDS function 949
show_role SQL Server function 573
SIGN function 939
SIMILAR function 943
SIN function 940
SOUNDEX function 109, 943
SQRT function 940
string 566
STRING function 943
SUBSTR function 944
SUM function 937
suser_id SQL Server function 573
suser_name SQL Server function 573
system 336, 573
TAN function 940
text and image 572, 573
today 1331
TODAY function 949
TRACEBACK function 302, 967
Transact-SQL 565, 566, 568, 570, 572, 573
TRIM function 944
TRUNCATE function 940
tsequal 552
tsequal SQL Server function 573
UCASE function 944
used_pgs SQL Server function 573
user_id SQL Server function 573
user_name SQL Server function 573
user-defined 1005, 1130
valid_name SQL Server function 573
valid_user SQL Server function 573
WEEKS function 949
YEAR function 950
YEARS function 950
YMD function 950

G

GET DATA statement
  syntax 1081
GET OPTION statement
  syntax 1084
getdate Transact-SQL function 568
getting commands 92
global variables
  about 556, 903
  in procedures 903
  selecting 903
GOTO statement
  Transact-SQL statement 618
GRANT CONSOLIDATE 466
GRANT CONSOLIDATE statement
  syntax 1089
GRANT PUBLISH 434, 442, 464
GRANT PUBLISH statement
  syntax 1091
GRANT REMOTE 434, 442, 466
GRANT REMOTE statement
  syntax 1092
GRANT statement
  and concurrency 260
  CONNECT permissions 394
  creating groups 386
  DBA authority 381
  example 140
  group membership 387
  inheriting permissions 383
  new users 380
  passwords 381
  procedures 384
  resource authority 381
  syntax 1085
  table permissions 382
  Transact-SQL 589
  UPDATE permissions on views 394
  views permissions 382
  WITH GRANT OPTION 383

without password 389
granting consolidate permissions 466
granting permissions
about 379
on procedures 379
granting publish permissions 464
granting remote permissions 466
GROUP BY ALL clause
Transact-SQL 593
GROUP BY clause 123
and views 218
SELECT statement 1143
GROUP permissions
about 386
not inheritable 386
grouped data 121
groups
adding 60
adding users, using Sybase Central 61
creating 60, 386
creating in Sybase Central 387
granting membership in 387
in SQL Server and SQL Anywhere 541
managing 386
of services 528
permissions 379, 388
PUBLIC 390
special 389
SYS 390
without passwords 389

H

HAVING clause
and GROUP BY clause 125
SELECT statement 1143
header files
for Embedded SQL 659
heading
ISQL option 1165
heading name 1142
Help 86
accessing from ISQL 71
HELP statement
syntax 1094
hexadecimal constants
about 921
data type 1162
HintUsed property 955, 959
HLI
about 654
host variables 897
about 667
bind variables 692
Transact-SQL compatibility 1163
types 668
host_id SQL Server function 573
host_name SQL Server function 573
hot links 767
HOUR function 948
HOURS function 948
HP UX 809

I

I/O
estimates 334
idle 402
operations 817
IBM AIX 809
IBM C Set ++ 656
icon
for running services 524
icons
used in manuals 10
identifiers
uniqueness of 547
valid 897
identity column 554
Transact-SQL 550
idle I/O
task 402
IdleCheck property 959
IdleChkpt property 959
IdleChkTime property 959
IdleWrite property 959
IF expression 906
IF statement 283
and passthrough mode 515
syntax 1095
Transact-SQL statement 618
IF UPDATE clause
in triggers 616, 982, 1028
IFNULL function 966
IMAGE data type
Transact-SQL 551

import libraries
alternative to 723
for Embedded SQL 660
importing data 364, 370, 372
conversion errors 372
INPUT statement 1098
performance 373
IN conditions 110, 578, 912
INCLUDE statement
and the SQLCA 673
syntax 1097
inconsistency 248
incremental backups
about 826
defined 411
performing 413
IndAdd property 955, 959
index_col SQL Server function 573
INDEX_ESTIMATE function 966
indexes
about 1007
and foreign keys 1008
and primary keys 1008
and table use 1008
and views 1007
automatically created 1008
balanced indexes 327
clustering 585
creating 223, 326, 585, 1007
creating, in Sybase Central 224
dropping 223, 1053
dropping, in Sybase Central 224
how indexes work 327
in the system tables 1338, 1340
inspecting 224
naming 1008
on primary keys 223
owner 1007
system views 1358
Transact-SQL 585
unique 1007
unique names 1008
uniqueness of names 547
using 223
using to improve performance 326
using with views 223
indicator variables 897
about 671
IndLookup property 955, 959
inequality, testing for 106
initialization files 810
initialization functions
listing 702
initializing databases 841
inner joins
FROM clause 1077
in procedures 292
using 149
input format
ISQL option 1165
INPUT statement
about 371
syntax 1098
INSERT
multi-row 686, 1062, 1125
PUT statement 1124
wide 686, 1062, 1125
insert mode 91
INSERT permission 382
INSERT statement
about 371
and data import 372
and integrity 227
examples 128, 133
FROM SELECT clause 372
syntax 1102
Transact-SQL 590
truncation of strings 1162
INSERTSTR function 942
INT data type 921
Transact-SQL 549
INTEGER data type 921
Transact-SQL 549
integer data types 157
Transact-SQL 549
integer overflow
ANSI behavior 1158
integrity
about 225, 226
constraints 227, 229, 1022
reviewing 244
rules in the system tables 244
SQL statements for 229
interface library
dynamic loading 723
Internet
and SQL Remote 458
e-mail 458

interrupt 96
interrupts
  handling 719
INTO clause 291, 682
  SELECT statement 1142
invalid data 227
  types of 226
IS FALSE conditions 914
IS NULL conditions 578, 913
IS TRUE conditions 914
IS UNKNOWN conditions 914
ISNULL function 966
isolation levels 250
  about 251
  and locks 251
  and serializability 257
  changing within a transaction 252
  choosing 256
  cursors 1113
ISOLATION_LEVEL option 1150
ISQL
  about 40
  and Open Server Gateway 644
  character mode 81
  command line 846
  Command window 72
  connecting to a database 1000
  displaying a list of tables 112
  erasing databases 210
  error handling 1167
  executing commands 72, 92
  for DOS 81
  for Microsoft Windows 67
  for Microsoft Windows NT 67
  for OS/2 67
  for QNX 81
  getting commands 72, 92
  interrupting commands 79
  loading commands 72, 92
  options 1165
  saving commands 72, 92
  starting from Sybase Central 848
  statistics window 322
  using 67, 205
ISQL command delimiter 309
ISQL_LOG
  ISQL option 1167

J

Japanese character set 352
joining lines 91
joins
  about 111
  and deletes 1045
  and updates 1183
  automatic 1311
  cross product 114
  execution plans 329
  FROM clause 1075
  inner joins 149, 292
  key 1311
  natural 1311
  or subqueries 148
  outer joins 149, 291, 564
  SELECT statement 1142
  Transact-SQL 564, 591
  updates based on 1184

K

key joins
  FROM clause 1077
  using 118, 328
keyboard 84
keys
  about 183
  and performance 322
  assigning 192
  choosing 201
  creating 214
  foreign 183
  primary 183
keywords
  listing of 1316
  NON_KEYWORDS option 1161
  turning off 1161
Korean character set 352

L

labels
  for statements 618, 898
language support 354
  multibyte character sets 352
laptop computers 451

and replication 451
and SQL Remote 451
and transactions 262
large databases
index storage 1007
LAST USER
special constant 901
LastIdle property 955
LastReqTime property 955
LCASE function 942
lct_admin SQL Server function 573
leaf page
and indexes 327
leap years 932
LEAVE statement
syntax 1104
LEFT function 942
left, moving screen 93
LegalCopyright property 959
LegalTrademarks property 959
length 692
LENGTH function 942
libraries
for Embedded SQL 660
library functions
and Embedded SQL 701
LIKE conditions 577, 909
LIKE operator 108
line length
SQLPP output 893
line numbers 893
links 767
LIST function 122, 937
literal strings 898, 900
live backups 415
LOAD DATABASE statement
Transact-SQL 588
LOAD TABLE statement
about 370
syntax 1105
LOAD TRANSACTION statement
Transact-SQL 588
loading commands 92
loading tables 1105
local variables
about 556
and result sets 624
LocalSystem account 524
LOCATE function 942
locks 248, 250
exclusive 250
how locking works 250
nonexclusive 250
phantom 250
read 250
write 250
LockTablePages property 955, 959
log files 871
about 42
checkpoint 401
erasing 42
rollback 403
transaction 403
LOG function 939
log translation utility 404, 418
LOG10 function 939
LogFreeCommit property 955, 959
logging on 84
to a database 48, 70
logical operators 579
and three-valued logic 914
LogName property 963
logon options 524
LogRewrite property 955, 959
LogWrite property 955, 959
LONG BINARY
replication 492
LONG BINARY data type 926
Transact-SQL 551
long column names
retrieving 1051
long commands 105
LONG VARCHAR
replication 492
LONG VARCHAR data type 918
Transact-SQL 550
lookup
and indexes 327
LOOP statement 283
and passthrough mode 515
in procedures 297
syntax 1108
lost updates
and transactions 256
LOTUS file format 1165, 1167
Lotus Notes
and SQL Remote 429, 458, 461
LTM

transaction log options 871
LTRIM function 942

M

Macintosh
ODBC programming for 762
MainHeapPages property 959
mandatory foreign keys 241
MAPI
system procedures 1323, 1324, 1325
MAPI link 429
MAPI message type
about 976
MAPI messages
about 1015, 1058
MapPages property 959
MASTER service 643
mathematical expressions 899
MAX function 122, 937
maximum function 122, 937
MaxIO property 959
MaxRead property 959
MaxWrite property 959
media
recovery from failure of 416
media failure 403
recovery from 406
membership
of groups 387
memory 718
allocation
DOS 718
QNX 718
and caching 320
usage 719
menu 85
Message Agent 424, 447
about 496
and message tracking 500, 502
and recovery 498
and security 497
and transaction log management 503, 504
command line 863
delivering messages 500, 502
message links
parameters 459
MESSAGE statement
in procedures 301
syntax 1109
message-based replication 423
messages
and synchronizing databases 487
creating 586
delivering 500, 502
in SQL Remote 500, 502
receiving 447
sending 447
tracking 500, 502
types 976, 1015
dropping 1058
Microsoft C
support for 656
Microsoft Visual C++
support for 656
millenium issues 930
MIN function 122, 937
minimum function 122, 937
MINUTE function 948
MINUTES function 948
mirror
transaction log 403, 406, 408
mobile workforces 428, 451, 472
MOD function 939
MONEY data type
Transact-SQL 551
money data types
Transact-SQL 551
monitoring
services 531
monitoring performance 338
MONTH function 948
MONTHNAME function 948
MONTHS function 948
moving screen 93
multibyte character sets 352
multiple result sets from procedures 293
multiple row queries 295
and cursors 678
multi-row fetches 686, 1070
OPEN statement 1113
multi-row inserts 686, 1062
multi-row puts 686, 1125
multi-tier installations
passthrough statements 514

N

Name property 955, 959, 963
national language support 354
  multibyte character sets 352
natural joins
  FROM clause 1076
NCHAR data type
  and OmniCONNECT 650
  Transact-SQL 550
NEAREST_CENTURY option 1160
NetWare
  about 817
  and SQL Remote 457, 460
  cache buffers 817
  file system 817
  ISQL 81
network server
  connecting to 26
  same-machine connections 26
new databases 525
next_connection function 954
next_database function 954
NLMs
  calling from functions 314
  calling from procedures 314
NO SCROLL cursors 1039
NodeAddress property 955
NON_KEYWORDS database option 1161
nonexclusive locks 250
non-repeatable reads 248, 251
normal forms 191
NOT conditions 913
not found warning 295, 678
NOT NULL constraint 228
Notes
  and SQL Remote 458
NOW function 949
NT performance monitor 338
NT services
  example code 746, 760
  startup options 522
NULL value 691, 965
  about 1110
  allowed in columns 128
  and columns 200
  and indicator variables 671
  and output 369
  ANSI behavior 1159
  column default 546
  default 233
  integrity action 242
  NULLS option 1167
  Transact-SQL behavior 1159
NULLS option
  ISQL option 1167
NUMBER function
  about 966
  and updates 1184
  limitations 966
number of rows 1347
Number property 955
numbers 900
  defaults 233
NUMERIC data type 921
  Transact-SQL 550
numeric functions 565, 938
numeric precision
  database option 1155
NVARCHAR data type
  and OmniCONNECT 650
  Transact-SQL 550

O

object_id SQL Server function 573
object_name SQL Server function 573
objects
  name prefixes 391
  qualified names 391
ODBC
  about 26
  Administrator 170, 173, 174, 179
  and character sets 349
  conformance 169
  connecting from 169
  cursors 756
  data sources 163, 173
    about 162
  data sources, adding 174
  data sources, modifying 179
  DOS and 169, 180
  driver manager 170
  interface 34
  language dll 172
  Macintosh applications 762

ODBC.DLL 170
ODBC.INI 170, 173, 174, 179
ODBCINST.INI 170, 173
OS/2 and 169, 179
programming 752
QNX and 169, 180
SQL Anywhere driver 170
static cursors 1039
support for 653
unsupported functions 759
Windows and 170
Windows NT and 170
ODBC programming 751
offline backups
defined 411
OmniCONNECT
about 630, 649
and Open Server Gateway 630
data types 650
support for 649
support limitations 650
ON EXCEPTION RESUME 300
ON_ERROR
ISQL option 1167
online backups 711
about 826
defined 411
online Help
contents 65
documentation 9
for Sybase Central 65
full-text search 66
index 66
searching 66
Open Client data types
description limitations 640
mapping from SQL Anywhere data types 637
mapping to SQL Anywhere data types 636
problem free 634
unsupported in SQL Anywhere 641
value range limitations 638
Open Server Gateway 855, 856
adding 642
and database options 644
and OmniCONNECT 630
and Replication Server 630
and two-phase commit 648
configuring 642
connecting to 644
data type mappings 634
database requirements 633
information 855
limitations 648
master service for 643
performance 644
starting 643
stopping 856
verbose mode 644
Open Servers 853, 855, 856
connecting to 644
master service 643
query service 643
starting 643
OPEN statement 733
about 678
in procedures 295
QUERY_PLAN_ON_OPEN option 1161
syntax 1112
opening cursors 1161
operating systems
supported 5
operators
comparison operators 908
precedence of 564
Transact-SQL 563, 577
optimizer
about 333
cost based 333
estimates 333
optional foreign keys 241
options
about 1150
ALLOW_NULLS_BY_DEFAULT 546, 1157, 1158
ANSI_BLANKS 1157, 1158
ANSI_INTEGER_OVERFLOW 1158
ANSI_PERMISSIONS 1159
ANSINULL 1159
AUTOMATIC_TIMESTAMP 546, 1157, 1159
BACKGROUND_PRIORITY 1150, 1151
BLOCKING 1150
CHAINED 1160
CHECKPOINT_TIME 1150
CLOSE_ON_ENDTRANS 1160
CONVERSION_ERROR 1157
CONVERSION_ERROR option 1160
COOPERATIVE_COMMIT_TIMEOUT 1150

COOPERATIVE_COMMITS 1150
DATE_FORMAT 1150
DATE_ORDER 1150
DELAYED_COMMIT_TIMEOUT 1150
DIVIDE_BY_ZERO_ERROR 1160
FIRE_TRIGGERS 1160
GET OPTION statement 1084
in the system tables 1341
ISOLATION_LEVEL 1150
NEAREST_CENTURY option 1160
NON_KEYWORDS option 1161
PRECISION 1150
QUERY_PLAN_ON_OPEN 1157, 1161
QUOTED_IDENTIFIER 546, 562, 1157, 1161
RECOVERY_TIME 1150
retrieving 1084
RI_TRIGGER_TIME 1162
ROW_COUNTS 1150
SCALE 1150
set by Open Server Gateway 644
setting 1149
setting ISQL options 998
SQL_FLAGGER_ERROR_LEVEL 1162
SQL_FLAGGER_WARNING_LEVEL 1162
system views 1358, 1361
THREAD_COUNT 1150
TIME_FORMAT 1150
Transact-SQL 594
TSQL_HEX_CONSTANT 1162
TSQL_VARIABLES 1163
WAIT_FOR_COMMIT 1150
OR keyword 110, 913
ORDER BY clause 105, 1143
and performance 331
examples 103
OS/2
and ISQL 67
and SQL Remote 457
DOS client applications 32
Windows 3.x client applications 32
OS/2 DLL
for Embedded SQL 701
outer joins
and subqueries 905
FROM clause 1077, 1078
in procedures 291
operators 564
Transact-SQL 564, 591
using 149
outer references
defined 148
output length
ISQL option 1169
output redirection 368
OUTPUT statement
about 368
syntax 1115
owners
about 377
assigning ownership 389

P

page size
and performance 320
setting 841
PageRelocations property 959
pages
displaying usage in database files 837
PageSize property 963
parameters
command files 1126
to command files 154
to functions 122
PARAMETERS statement
syntax 1118
PASSTHROUGH 513
passthrough mode 513
starting 1119
stopping 1119
uses 514
PASSTHROUGH statement
syntax 1119
passthrough statements
and multi-tier installations 514
password 48, 84
passwords
and connecting 162
and permissions 140, 394
changing 381, 1086
defined 163
in the system tables 1352
using with ISQL 70
PATINDEX function 943
pattern matching 108, 577, 909, 943
and case-sensitivity 910

and collations 910
limits 910
maximum length of pattern 910
pausing
services 526
PendingReq property 959
performance
and cache size 320, 817
and foreign keys 325
and indexes 223
and multiple table queries 328
and page size 320, 327
and primary keys 324
disk fragmentation 209
improving 126, 319
monitoring 319, 336, 337, 338
of bulk operations 373
setting priorities 1151
temporary tables 332
transaction log 403
using ISQL to examine 322
performance monitor
NT 338
starting 338
permissions 158
about 376
assessing 396
changing 381
CONNECT authority 1086
consolidate 466, 1089
creating groups 386
DBA authority 376, 1086
execute 384, 1088
for triggers 377, 385
GRANT statement 1085
granting 378, 381, 382
granting connect permissions 380
granting group 379
granting group membership 387
granting individual 380
granting on procedures 384
granting passwords 380, 381
granting resource permissions 377, 381
group 379
GROUP authority 1086
in Sybase Central 383, 384
in the system tables 1333, 1348
individual 380
inheriting 383, 386
inspecting 397
managing 375
MEMBERSHIP 1086
of groups 388
on procedures 395
on tables 221, 378, 382
on triggers 279
on views 140, 221, 382
publish 434, 442, 464
remote 434, 442, 466
resource authority 377, 1086
revoking 385, 1132
revoking CONSOLIDATE 1134
revoking PUBLISH 1135
setting 58
SQL Server compatibility 541, 589
SYSCOLAUTH system view 1356
system views 1360
the right to grant 383
UPDATE, on views 394
users 375
WITH GRANT OPTION 383
phantom
locks 250
reads 251
rows 248
Pharlap
support for 660
PI function 939
place holders
and dynamic SQL 683
PLAN function 967
Platform property 959
platforms
supported operating systems 5
Polling frequency
for services 519
Port property 955
portable computers
and databases 262
and decision support applications 263
and multiuser databases 262
and SQL Anywhere 262
runtime applications 263
power failure
recovery from 416
POWER function 939
PowerBuilder
database connections 166

preallocating space for the transaction log 972
precedence of operators 564
PRECISION option 1150
predicates
about 907
PREPARE statement 683
syntax 1120
PREPARE TO COMMIT statement
syntax 1123
two-phase commit 260
PREPARE TRANSACTION statement
and Open Server Gateway 648
prepared statement 1120
prepared statements 1062
in the system tables 1348
PrepStmt property 955
previous commands 94
primary key errors 476
in SQL Remote 476
primary keys 183
about 966
adding 53
and concurrency 259
and entity integrity 239
and indexes 223
and integrity 1310
and performance 324
and replication 478, 479
and SQL Remote 478, 479
and transaction log 404
choosing 201
creating 53, 214
entity integrity 240
generation 259
identifying rows by 117
in the system tables 1334, 1337, 1347
integrity constraints 1023
using to improve performance 322
primary table
in the system tables 1337
PRINT statement 619
proc_role SQL Server function 573
ProcedurePages property 959
procedures 1010, 1051, 1121
about 266
allowed statements in 286
altering 974
and control statements 283
and data definition statements 286, 1064
and dynamic SQL statements 1064
and efficiency 267
and passthrough mode 515
and security 267, 393, 395
and SQL Remote 490, 491
and standardization 267
and Transact-SQL compatibility 608
and Transact-SQL support 606
benefits of 267
calling 269, 288, 617
catalog 601, 602
command delimiter and 309
control statements 614
creating 268, 1009
cursors in 295
default values for parameters 287, 288
dropping 270, 1053
dynamic statements in 307
Embedded SQL 696
error handing in Transact-SQL 620
error handling 300, 304
errors in 300
EXECUTE IMMEDIATE 286
EXECUTE IMMEDIATE statement 307
executing 269, 288, 617
external 314
FOR statement 299
in ODBC 757
invoking 269
multiple result sets from 293
OUT parameters 270
owner 377
parameters 287, 288
permissions for creating 377
permissions on 379, 384
RAISERROR statement 620
replicating 490, 491, 974
result sets from 292
returning results from 270, 290
returning values from 1130
setting permissions 58
SQL statements allowed in 286
system 601
table names in 309
testing 271, 309
Transact-SQL compatibility 615
Transact-SQL CREATE PROCEDURE statement 615
Transact-SQL overview 610

Transact-SQL RETURN statement 621
translation of 58, 608
translation of, using Sybase Central 608
using 265, 268
using cursors in 297
using ON EXCEPTION RESUME 300
variable result sets from 294
viewing 57
viewing Transact-SQL syntax 58
warnings in 300
writing 309
ProcessTime property 955
product name
retrieving 1327
ProductName property 959
ProductVersion property 959
program structure
Embedded SQL 662
programming interfaces
about 25, 34
DDE 35
embedded SQL 34
HLI 35
ODBC 34
supported by SQL Anywhere 34
projections
SELECT statement 1142
properties
connection 955, 1321
database 963, 1321
engine 959, 1321
property function 954
property_description function 954
property_name function 954
property_number function 954
PUBLIC group 390
in the system tables 1352
publications 424
and primary keys 478, 479
and referential integrity 478
and subqueries 480, 482
and updates 1185
creating 436, 444, 1013
designing 475, 476, 477, 478, 479, 1185
dropping 473, 1057
example 450
for many subscribers 472
notes on 474
selected columns 471
setting up 470
using a WHERE clause 472
using SUBSCRIBE BY clause 472
whole tables 470
publish permissions 434, 442, 1091
granting 464
managing 463
remote permissions 434, 442
revoking 464, 1135
publisher 435, 443, 464, 900
address 976, 1015, 1058
GRANT PUBLISH statement 1091
remote 1092
PURGE clause
FETCH statement 1070
PUT
multi-row 686
wide 686
PUT statement
and integrity 227
multi-row 1125
syntax 1124
wide 1125

Q

QNX 809
and SQL Remote 457
ISQL 81
ISQL Statistics window 323
memory
allocation 718
SQL Anywhere Client 29, 37
qualified object names 391
QUARTER function 949
queries
joins 591
optimization 333
SELECT statement 1141
sorting results of 331
Transact-SQL 592
QUERY service 643
QUERY_PLAN_ON_OPEN option 1157, 1161
quotation marks
and identifiers 897
in SQL 215, 394
using 105

QUOTED_IDENTIFIER option 546, 562, 1157

R

RADIANS function 939
RAISERROR statement 620
RAND function 939
range 106
Rational DOS4G
  support for 660
read locks 250
READ statement
  syntax 1126
ReadHint property 955, 959
READTEXT statement
  Transact-SQL 591
REAL data type 921
  Transact-SQL 550
REBUILD 42, 857
rebuilding databases 857
recalling commands 94
receiving messages 447
recover
  rapid 415
recovery
  about 399
  and SQL Remote 498
  from media failure 406
  time 1155
  transaction log 403
  uncommitted changes 418
RECOVERY_TIME option 1150
redirection 368
REFERENCE permissions 382
referential integrity 1310
  about 228
  and replication 478
  and SQL Remote 478
  breached by client application 242
  checking 243
  enforcing 239, 240
  FROM clause 1074
  losing 241
referential integrity actions
  and triggers 1162
registry
  and SQL Remote 459
registry entries 810
relational databases
  about 182
  concepts 182
  terminology 182
relations
  and entities 182
relationships
  between entities 185
  in the system tables 1337
  resolving 195
RELEASE SAVEPOINT statement 258
  syntax 1127
RelocatableHeapPages property 959
REMAINDER function 939
remote databases 422
  and remote permissions 467
  setting up 438, 445, 446
remote permissions 1092
  granting 466
  managing 463
  revoking 466, 468, 1136
remote users
  REVOKE REMOTE statement 1136
removing
  services 521
renaming
  columns 980
  tables 980
REPEAT function 943
repeatable reads 248
REPLICATE_ALL
  replication option 1163
replicating procedures 490, 491
replicating triggers 490, 491
replication 859
  administering 427
  and mobile workforces 451
  and triggers 479
  backup procedures 503, 504
  by e-mail 423
  case studies 451, 453
  conflicts 475, 507
  data types 492
  dbcc 871
  DBREMOTE 863
  design 475
  errors 507
  Message Agent 863
  message-based 423

of blobs 492
of control statements 515
of cursor statements 515
of procedures 490, 491, 974
of SQL statements 513
of stored procedures 515
of trigger actions 1160
of triggers 490, 491
options 1163
passthrough mode 513
primary key errors 476, 478, 479
publications 424
referential integrity errors 478
replication server 871
server-to-laptop replication 451
server-to-server 453
setup examples 451
subscriptions 424
synchronization 859
transaction log and 427
transaction log management 503, 504
UPDATE conflicts 477
upgrading databases 505
replication options
REPLICATION_ERROR 508, 1163
VERIFY_THRESHOLD 1164
Replication Server
and Open Server Gateway 630
and SQL Remote 498
Open Servers 853, 855, 856
REPLICATION_ERROR
replication option 508, 1163
Req property 959
ReqType property 955
request processing
and Windows 3.x 716
requests
aborting 715
reserved words *See* keywords
reserved_pgs SQL Server function 573
resident program 814
RESIGNAL statement
in procedures 305
syntax 1128
RESOLVE UPDATE triggers 507, 509, 510
resource authority 381, 386
about 377
in the system tables 1352
not inheritable 386
resource management 589
restore *See* recovery
restoring
databases 399
RESTRICT 1025
restrict action 242
restrictions *See* WHERE clause
RESULT clause
about 1009
result sets
and local variables 624
and temporary tables 624
from procedures 271, 292
multiple 293
shape of 294, 1010, 1051, 1121
variable 294, 1010, 1051, 1121
RESUME statement
syntax 1129
return codes 808, 813
RETURN statement
syntax 1130
Transact-SQL 621
returning results from procedures 290
REVOKE CONSOLIDATE 466
REVOKE CONSOLIDATE statement
syntax 1134
REVOKE PUBLISH 464
REVOKE PUBLISH statement
syntax 1135
REVOKE REMOTE 466, 468
REVOKE REMOTE statement
syntax 1136
REVOKE statement
about 385
and concurrency 260
syntax 1132
Transact-SQL 589
revoking consolidate permissions 466
revoking publish permissions 464
revoking remote permissions 466, 468
REXX 802
RI_TRIGGER_TIME option 1162
RIGHT function 943
right, moving screen 93
Rlbk property 955
role name 898
role names
about 1024
roles

compatibility with SQL Server 540
roll forward 403
ROLLBACK
TO SAVEPOINT statement 258
rollback log 403
and recovery 403
ROLLBACK statement 258
about 130
and procedures 308
and transactions 246
examples 132
in compound statements 285
ROLLBACK TRIGGER 1139
syntax 1137
Transact-SQL 592
ROLLBACK TO SAVEPOINT statement
syntax 1138
ROLLBACK TRIGGER statement
syntax 1139
RollbackLogPages property 955, 959
root file
about 28
defined 31
ROUND function 939
rounding
SCALE option 1155
row counts 1155
row not found 295, 678
ROW_COUNTS option 1150
rowcnt SQL Server function 573
rows
about 101, 182
selecting 105
RTDSK50 37
RTDSK50S 37
RTDSK50W 37
RTRIM function 943
RTSQL
command line 846
rules
Transact-SQL 585
runtime database engine 37
runtime system 37

S

sa_conn_info system procedure 1321
sa_conn_properties system procedure 1321
sa_db_info system procedure 1321
sa_db_properties system procedure 1321
sa_eng_properties system procedure 1321
sample database
connecting to 70
SAVEPOINT statement
and transactions 258
syntax 1140
savepoints 308
about 898
and procedures 308
and triggers 308
RELEASE SAVEPOINT statement 1127
ROLLBACK TO SAVEPOINT statement 1138
within transactions 258
saving commands 92
saving statements 1167
SCALE option 1150
schema
and system tables 1329
creating 587, 1017
Transact-SQL 587
viewing 49
screen
moving left or right 75
SCROLL cursors 1039
sdefaultx
user ID 232
search conditions
about 907
ALL conditions 578, 912
and GROUP BY clause 125
AND keyword pattern matching 108
and logical operators 579
and three-valued logic 914
ANY conditions 578, 912
BETWEEN conditions 577, 909
comparison conditions 908
date comparisons 106
estimates 579
EXISTS conditions 578, 912
IN conditions 578, 912
introduction to 105
IS NULL conditions 578, 913
LIKE conditions 577, 909
NOT conditions 913
short cuts for 110
subqueries 141, 908

Transact-SQL 576, 577, 578, 579
truth value conditions 914
SECOND function 949
SECONDS function 949
security 84
about 375
and procedures 267
passwords 70
services 524
tailored 393
using views for 393
SELECT 89
SELECT LIST 685
DESCRIBE statement 1049
SELECT statement 1142
SELECT permissions 382
SELECT statement
about 99, 141
access plans 328
and ISQL 74
and performance 326
examples 1355
FROM clause 1074
INTO clause 291
keys and query access 322
syntax 1141
to view variable values 560
Transact-SQL 592
SEND AT clause 1089, 1092
publish 1091
SEND EVERY clause 1089, 1092
sending messages 447
sequential searching 324
serializable transactions 257
server
defined 31
name 31
ServerName connection parameter 163
server-to-server replication 453
services
about 518, 520, 525, 526
account 524
adding 520
adding new databases 525
command-line switches 523
configuring 522
deleting 521
dependencies 528, 529
example code 746, 760
executable file 524
failure to start 523
groups 528
icon on the desktop 524
load ordering groups 528
managing 519
monitoring 531
more than one server 528, 530
multiple 528, 530
options 523
parameters 522
pausing 526
removing 520, 521
security 524
service manager 532
setting login options 524
setting logon options 524
starting 526
starting order 529
startup options 522
stopping 526
Windows NT Control Panel 532
SET CONNECTION statement
syntax 1147
SET DEFAULT 242, 1025
SET NULL action 242, 1025
SET OPTION statement
syntax 1149
Transact-SQL 546, 594
SET SQLCA statement
syntax 1170
SET statement
in procedures 290
syntax 1145
Transact-SQL 594
setting up a remote database 438, 445
setting up publications 470
setting up subscriptions 483
setting up the consolidated database 433, 442
Shift-JIS collation 352
show_role SQL Server function 573
SIGN function 939
SIGNAL statement
in procedures 301
syntax 1171
SIMILAR function 943
SIN function 940
single row queries 290, 291
in Embedded SQL 677

SMALLDATETIME data type
Transact-SQL 552
SMALLINT data type 921
Transact-SQL 549
SMALLMONEY data type
Transact-SQL 551
SMTP
and SQL Remote 458
e-mail 458
SMTP link 429
software
components 808
DBBACKUP 826
DBCOLLAT 830
DBERASE 835
DBEXPAND 874
DBINFO 838
DBINIT 841
DBLOG 871
DBOS50 853
DBOSINFO 855
DBOSSTOP 856
DBREMOTE 863
DBSHRINK 833
DBSTOP 867
DBTRAN 850
DBUNLOAD 878
DBUPGRAD 883
DBVALID 886
DBWRITE 890
DBXTRACT 859
ISQL 846
REBUILD 857
return codes 813
RTSQL 846
SQLPP 892
SOME conditions 912
sort position
description 359
sorting
in the system tables 1332
query results 103, 331
SOUNDEX function 109, 943
sp_addgroup system procedure 601
sp_addlogin system procedure 601
sp_addmessage system procedure 586, 601
sp_addtype system procedure 601
sp_adduser system procedure 601
sp_changegroup system procedure 601
sp_column_privileges
catalog procedure 602
sp_columns catalog procedure 602
sp_dboption system procedure 601
sp_dropgroup system procedure 601
sp_droplogin system procedure 601
sp_dropmessage system procedure 601
sp_droptype system procedure 601
sp_dropuser system procedure 601
sp_fkeys catalog procedure 602
sp_getmessage system procedure 601
sp_helptext system procedure 601
sp_password system procedure 601
sp_pkeys catalog procedure 602
sp_special_columns catalog procedure 602
sp_sproc_columns catalog procedure 602
sp_stored_procedures catalog procedure 602
sp_tables catalog procedure 602
special tables 1329
speed
and performance 319
SQL 895
communicating with 26
dialects 606
SQL/92 syntax 1162
SQL/92 syntax 893
values 215
SQL Anywhere
multiplatform support 30
new features 15
operating systems 30
standalone database engine 37
statistics 338
SQL Anywhere Client 37
about 29
QNX 29, 37
SQL Anywhere Desktop Runtime System 37, 263
SQL Anywhere Open Server 853
query service for 643
SQL Anywhere server 518
for Windows NT 518
SQL Preprocessor
command-line switches 892
SQL Remote 419
about 455
administering 427, 455, 513, 514
ALTER REMOTE MESSAGE TYPE
statement 976
altering publications 975

and database recovery 498
and mobile workforces 428, 451
and procedures 490, 491
and triggers 490, 491
articles 1331, 1332
avoiding conflicts 475
avoiding primary key errors 476
avoiding referential integrity errors 478
backup procedures 503, 504
case studies 451, 453
concepts 422
conflict resolution 507, 510
consolidate permissions 1089, 1134
consolidated databases 422
CREATE REMOTE MESSAGE TYPE statement 1015
creating publications 436, 444, 1013
creating subscriptions 483, 1019
dbremote 447
DBXTRACT utilities 859
deployment 494
designing publications 475
designing triggers 479
DROP REMOTE MESSAGE TYPE statement 1058
dropping publications 473, 1057
dropping subscriptions 1061
error reporting 507
extracting a database 859
for DOS 457
for many subscribers 472
for NetWare 457
for OS/2 457
for QNX 457
for Windows 3.x 457
for Windows 95 457
for Windows NT 457
GRANT PUBLISH statement 1091
granting publish permissions 434, 442
granting remote permissions 434, 442
managing 494
Message Agent 424, 447
message delivery 500, 502
message systems 429
message tracking 500, 502
multi-tier installations 514
options 1163
passthrough mode 513, 1119
primary key errors 478, 479
publications 424
publish permissions 1135
publishing a subset of rows 472
publishing selected columns 471
publishing whole tables 470
remote databases 422
remote permissions 1092, 1136
server-to-laptop replication 451
server-to-server replication 453
setting up 430, 439
setting up a consolidated database 433, 442
setting up a remote database 438, 445
setup examples 451
starting subscriptions 1174
stopping subscriptions 1177
subqueries 480, 482
subscribers 428
subscriptions 424
synchronization 859
synchronizing databases 484, 487
synchronizing subscriptions 1178
system tables 1331, 1332
transaction log management 503, 504
unloading databases 505
UPDATE conflicts 477
upgrading databases 505
upgrading installations 494
SQL variables
creating 1031
dropping 1060
SET VARIABLE statement 1145
SQL/92
conformance 893, 1162
sql_needs_quotes 714
SQLANY environment variable 810
SQLANY.INI
and SQL Remote 459
SQLCA 662
about 673
and threads 699
changing 699
multiple 700
SET SQLCA statement 1170
sqlcabc 674
sqlcaid 674
SQLCancel
support for 759
sqlcode 674
special constant 900

values 1191
SQLCONNECT environment variable 810
SQLDA
about 683, 690
allocating 709
and ANSI_BLANKS 1158
filling 709
freeing 710
length field 692
strings 710
sqlda_storage 710
sqlda_string_length 710
sqldef.h 747
SQLEDIT utility 642
sqlerrd 674
sqlerrmc 674
sqlerrml 674
sqlerror_message 714
sqlerrp 674
sqllen 692
sqlname 691
SQLPATH environment variable 811
SQLPP
about 656
command line 892
running 659
SQLREMOTE environment variable 459, 811
SQLSetConnectOption
support for 759
SQLSetStmtOption
support for 759
SQLSTART environment variable 811
sqlstate 675
special constant 900
values 1201
sqlwarn 675
SQRT function 940
Start connection parameter 162, 163
START DATABASE statement
syntax 1172
START ENGINE statement
syntax 1173
START SUBSCRIPTION statement
syntax 1174
starting
database engines 1173
databases 1172
ISQL, from Sybase Central 848
services 523, 529
Sybase Central 47
starting subscriptions 1174
startup options
for services 522
statement labels 618, 898
statement-level triggers 611
statements *See also* individual statement names
control 283
SQL, in procedures and triggers 286
using compound 283
static cursors
declaring 1039
static SQL 682
static SQL authorization 663
statistics
available 338
displaying 338
ISQL option 1169
monitoring 336
monitoring in Sybase Central 337
Statistics window 143
STOP DATABASE statement
syntax 1175
STOP ENGINE statement
syntax 1176
STOP SUBSCRIPTION statement
syntax 1177
stopping databases 867, 1175
stopping subscriptions 1177
store-and-forward 423
stored procedures
and passthrough mode 515
creating 1009
STRING function 943
string truncation
database option 1162
string type 710
strings
and host variables 1158
concatenating 563, 905
concatenation operators 563, 905
constants 898, 900
defaults 233
delimiter 562
functions 566
literal strings 898
Transact-SQL 562
structure packing
about 657

subqueries
ALL conditions 578
and comparisons 145
and publications 480, 482
ANY conditions 578
correlated subqueries 148
EXISTS conditions 578
in expressions 905
in search conditions 908
in SQL Remote 480, 482
or joins 148
using 141
SUBSCRIBE BY clause 472, 479, 1013
subqueries 480, 482
subscriptions 424
and updates 1185
creating 436, 444, 483, 1019
dropping 1061
setting up 483
starting 1174
stopping 1177
synchronizing 484, 487, 1178
SUBSTR function 944
subtransactions
and precedures 308
and savepoints 258
SUM function 122, 937
Sun Solaris 809
suser_id SQL Server function 573
suser_name SQL Server function 573
Sybase Central
about 39
adding columns 211
adding users to groups 387
altering columns 213
altering tables 213
and column constraints 237
and column defaults 231
and database objects 204
and permissions 383, 384
connecting to a database 48
creating groups 387
creating indexes 224
creating tables 211
creating users 380
creating views 218
documentation 65
drag and drop columns 54
dropping indexes 224
dropping views 222
erasing databases 210
main window 47
managing services 518, 519
starting 47
Starting ISQL from 848
Transact-SQL-compatible databases 544
translating procedures 608
synchronization utility 445
SYNCHRONIZE SUBSCRIPTION statement
syntax 1178
synchronizing databases 445, 446, 484
using DBXTRACT 484
using the extraction utility 484
using the message system 487
synchronizing subscriptions 1178
SyncWriteChkpt property 955, 959
SyncWriteExtend property 955, 959
SyncWriteFreeCurr property 955, 959
SyncWriteFreePush property 955, 959
SyncWriteLog property 955, 959
SyncWriteRlbk property 955, 959
SyncWriteUnkn property 955, 959
syntax rules 895
SYS group 390
in the system tables 1352
SYSCOLUMNS system view
conflicting name 545
SYSINDEXES system view
conflicting name 545
SYSREMOTEUSER 500
system administrator
SQL Servercompatibility 540
system calls
from stored procedures 1326
xp_cmdshell system procedure 1326
system catalog 1356
about 1329
diagram 1330
list 1331
SQL Server compatibility 540
Transact-SQL 598
using 155
system failure
recovery from 414
system fucntions
xp_msver 1327
system functions 336, 573
tsequal 552

system objects
and the dbo user 858, 876
system procedures
about 1319
sa_conn_info 1321
sa_conn_properties 1321
sa_db_info 1321
sa_db_properties 1321
sa_eng_properties 1321
sp_addgroup 601
sp_addlogin 601
sp_addmessage 601
sp_addtype 601
sp_adduser 601
sp_changegroup 601
sp_dboption 601
sp_dropgroup 601
sp_droplogin 601
sp_dropmessage 601
sp_droptype 601
sp_dropuser 601
sp_getmessage 601
sp_helptext 601
sp_password 601
Transact-SQL 601
xp_cmdshell 1326
xp_scanf 1328
xp_sendmail 1324
xp_sprintf 1327
xp_startmail 1323
xp_stopmail 1325
SYSTEM statement
syntax 1179
system tables
about 216, 1329
and character sets 358
and indexes 224
and integrity 244
and national languages 358
and views 222
diagram 1330
list 1331
permissions 397
SQL Server compatibility 540
SYSARTICLE 1331
SYSARTICLECOL 1332
SYSCATALOG 155
SYSCOLLATE 1332
SYSCOLLATION 358
SYSCOLPERM 1333
SYSCOLUMN 1334
SYSCOLUMNS 157
SYSDOMAIN 1335
SYSFILE 1336
SYSFKCOL 244, 1336
SYSFOREIGNKEY 244, 1337
SYSFOREIGNKEYS 244
SYSGROUP 1338
SYSICOL 224
SYSINDEX 224, 1338
SYSINDEXES 224
SYSINFO 358, 1339
SYSIXCOL 1340
SYSOPTION 1341
SYSPROCEDURE 1341
SYSPROCPARM 1342
SYSPROCPERM 1343
SYSPUBLICATION 1344
SYSREMOTEUSER 1344
SYSSUBSCRIPTION 1346
SYSTABLE 158, 222, 244, 1346
SYSTABLEPERM 1077, 1348
system views 1355
SYSTRIGGER 244, 1349
SYSUSERMESSAGES 1351
SYSUSERPERM 1077, 1351
SYSUSERTYPE 1353
Transact-SQL 545, 598
Transact-SQL name conflicts 545
users and groups 397
using 155
system variables 903
system views
about 1355
and indexes 224
and integrity 244
and views 222
SYSCATALOG 1356
SYSCOLAUTH 1356
SYSCOLUMNS 1357
SYSFOREIGNKEYS 1357
SYSGROUPS 1358
SYSINDEXES 1358
SYSOPTIONS 1358
SYSPROCPARMS 1359
SYSREMOTEUSERS 1359
SYSTABAUTH 1360
SYSTRIGGERS 1360

SYSUSERAUTH 1361
SYSUSERLIST 1361
SYSUSEROPTIONS 1361
SYSUSERPERMS 1362
SYSVIEWS 222, 1362

T

table constraints 1022
table editor 52
table list 78, 95
  FROM clause 1074
table naming 215
table number 1347
tables 378
  about 182, 211, 976, 1020
  adding 52
  adding columns 52
  adding keys to 214
  alias 115
  altering 212, 231, 235, 976
  altering definition 978
  and foreign keys 117, 183
  and view permissions 221
  base tables 1021
  characteristics 182
  CHECK conditions 228, 235, 237
  checks 237
  columns 199
  constraints 200, 214, 227, 228, 235
  correlation name 115
  creating 52, 211, 229, 235, 587, 1020
  creating a primary key with Sybase Central 53
  creating, in Sybase Central 211
  deleting 56
  deleting all rows from 1180
  designing 185, 199
  dropping 56, 214, 1053
  editing existing 53
  editor 52
  GLOBAL temporary 1021
  group owners 388
  in publications 470
  list of 78, 95
  loading 370, 1105
  loading using ISQL 371
  looking up 388
  name prefixes 391
  naming 898
  naming, in SQL 215
  owner 377
  owners 391
  permissions 377, 378
  primary keys 117
  qualified names 388, 391
  renaming 213, 980
  special 155
  structure 212
  table lists 898
  temporary 587, 1021, 1043
  Transact-SQL 587
  truncating 1180
  unloading 366, 1182
  using qualified names 309
  validating 1187
  viewing 50
Taiwanese character set 352
TAN function 940
TaskSwitch property 955
TaskSwitchCheck property 955
temporary file
  location 812
temporary tables
  and importing data 372
  and query processing 332
  and result sets 624
  created 372
  creating 1020
  declared 372, 587
  declaring 284, 1043
  GLOBAL 1020
  in procedures 624
  LOCAL 1043
  Transact-SQL 587
text and image functions 572, 573
TEXT data type
  Transact-SQL 550
TEXT file format 1167
textptr function 572, 573
THEN
  IF expression 906
thread count
  database option 1155
THREAD_COUNT option 1150
threads
  and Embedded SQL 699
three valued logic

NULL value 1110
three-valued logic 914
time 922
TIME data type 922
Transact-SQL 552
time format
database option 1156
TIME_FORMAT option 1150
times
comparing 925
timestamp
adding a timestamp column to a table 553
and OmniCONNECT 650
timestamp column 552
data type 553
timestamp columns 1159
TIMESTAMP data type 668, 922
Transact-SQL 552
timestamp format
database option 1156
TINYINT data type 921
Transact-SQL 549
TMP environment variable 812
TODAY function 949, 1331
TRACEBACK function 302, 967
trademark information
retrieving 1327
trailing blanks in strings 544
transaction log
about 42, 403
allocating space for 209, 972
and database performance 320
and portable computers 262, 263
and primary keys 404
and replication 427
and SQL Remote 427, 863
and uncommitted changes 418
DBLOG 871
erasing 835
Message Agent 863
mirror 406, 408
mirroring 403
monitoring performance 319
options 413
preallocating space for 972
size 404
Transact-SQL DUMP DATABASE 588
Transact-SQL LOAD DATABASE 588
translation utilities 850
TRUNCATE TABLE statement 1180
updating databases from 262
utilities 871
transaction log mirror
about 406
and replication 503
and SQL Remote 498
purpose 406
requirements 406
transaction management 582, 584, 592
in Transact-SQL 582, 584, 592
transaction modes
chained 582, 1160
unchained 582, 1160
transaction processing 246, 249
and performance 249
transactions
and concurrency 247
and data recovery 247
and deadlock 254
and lost updates 256
and procedures 308
and triggers 308
blocking 254
closing cursors 1160
committing 993
coordinating with multiple servers 260
overview 246
ROLLBACK statement 1137
ROLLBACK TO SAVEPOINT statement 1138
SAVEPOINT statement 1140
serializable 257
Transact-SQL
about 536
aggregate functions 565
ALL conditions 578
ALLOW_NULLS_BY_DEFAULT option 546, 1158
and logical operators 579
and procedures 626
and Watcom-SQL 606
ANY conditions 578
arithmetic operators 563
assigning values to variables 626
AUTOMATIC_TIMESTAMP option 1159
batches 613
BETWEEN conditions 577

binary data type compatibility 551
bit data type compatibility 551
bitwise operators 564
case sensitivity
user-defined data types 547
catalog procedures 601, 602
character data types compatibility 550
column NULLs compatibility 1158, 1161
COMMIT 584
comparison conditions 576
comparison operators 577
configuring databases for 544
constants 561
CREATE INDEX statement 585
CREATE MESSAGE 586
CREATE PROCEDURE statement 615
CREATE SCHEMA statement 587, 1017
CREATE TABLE statement 587
data type compatibility 549
data type conversion functions 570
database options 594
date and time data type compatibility 552
date and time functions 568
dbo user 545
decimal data type compatibility 550
DECLARE section 614
delete permissions 1159
DELETE statement 589
error handling in 620
executing stored procedures 617
EXISTS conditions 578
expressions 561
functions 565
global variables 557
GRANT statement 589
hexadecimal constants 1162
identity column 554
IN conditions 578
INSERT statement 590
integer data type compatibility 549
IS NULL conditions 578
joins 591
LIKE conditions 577
local variables 556
money data type compatibility 551
NULL behavior 1159
numeric functions 565
operators 563
outer join operators 564
procedures 610, 615
QUOTED_IDENTIFIER option 546, 1161
READTEXT statement 591
REVOKE statement 589
ROLLBACK statement 592
search conditions 576, 577, 578, 579
SELECT statement 592
SET statement 594
string concatenation operator 563
string functions 566
strings 562
system catalog 598
system functions 573
system procedures 601
system tables 545
text and image functions 572, 573
timestamp column 552
triggers 611
update permissions 1159
UPDATE statement 596
user-defined data types 555
viewing procedures in 58
WRITETEXT statement 597
writing portable SQL 543
translating
procedures 608
tree
and indexes 327
TriggerPages property 959
triggers
about 266
allowed statements in 286
altering 982
and control statements 283
and permissions 385
and referential integrity actions 1162
and replication 479, 1160
and SQL Remote 479, 490, 491
and Transact-SQL compatibility 611
benefits of 267
command delimiter and 309
creating 277
cursors in 295
dropping 279, 1053
error handling 300, 304
errors in 300
executing 279
execution permissions 279, 385

permissions for creating 377
replicating 490, 491
RESOLVE UPDATE 507, 509, 510
ROLLBACK statement 1139
statement-level 611
Transact-SQL 616
TRUNCATE TABLE statement 1180
uniqueness of names 547
using 265
using ON EXCEPTION RESUME 300
viewing 57
warnings in 300
Watcom-SQL 982, 1028
TRIM function 944
TRUE condition 913
TRUNCATE function 940
TRUNCATE TABLE statement
syntax 1180
truncating a table 1180
truncation
of character strings 1162
on FETCH 672
truncation length
ISQL option 1169
tsequal function 552
tsequal SQL Server function 573
TSQL_HEX_CONSTANT option 1162
TSQL_VARIABLES option 1163
tuples 182
Tutorial
using Sybase Central 45
Tutorials
Sybase Central 45
two-phase commit 260
and Open Server Gateway 648
PREPARE TO COMMIT statement 1123
type conversions 929
types
about data types 917

U

UCASE function 944
unchained transaction mode 582, 1160
UncommitOp property 955
unicode collation 352
UNION operation 1181
unique
constraint 1022
unique indexes 1007
uniqueness of object names 547
Unix 809
Unix commands 809
UNKNOWN condition 906, 913
UNLOAD TABLE statement
about 366
syntax 1182
unloading databases
and replication 505
unloading tables 366, 1182
UnschReq property 959
UP ISQL command 848
UPDATE
IF UPDATE clause 616, 982, 1028
UPDATE (positioned) statement
syntax 1186
update column permission 1333
UPDATE permissions 382
UPDATE statement
and integrity 227, 243
and transaction logs 263
examples 129
syntax 1183
Transact-SQL 596
truncation of strings 1162
updates
and joins 1183, 1184
and views 1186
ANSI behavior 1159
applying 262
Transact-SQL permissions 1159
upgrade utility 7
upgrading
and replication 505
and SQL Remote 505
databases 882, 883
SQL Remote installations 494
upgrading a database 7
used_pgs SQL Server function 573
USER
default 1021
number 1351
special constant 900, 1331
user IDs
and connecting 162
and permissions 140
and system tables 156

benefits of different user IDs 376
changing passwords 1086
creating 380
creation 1086
default 232
granting permissions to 394
in the system tables 1347, 1351
system views 1361
user_id SQL Server function 573
user_name SQL Server function 573
user-defined data types
about 927
case-sensitivity 547
CHECK conditions 236, 238
CREATE DATATYPE statement 1002
dropping 1053
Transact-SQL 555
user-defined functions
calling 274
CREATE FUNCTION statement 1005
creating 273
dropping 274
RETURN statement 1130
using 273
userid 84
about 898
defined 163
using with ISQL 70
Userid property 955
users
adding 61
adding to groups, using Sybase Central 61, 387
creating 61, 380
creating individual 380
creating, in Sybase Central 380
deleting 385
dropping 1133
in SQL Server and SQL Anywhere 541, 589
inspecting 397
managing 60, 375
occasionally connected 262
remote 426
UTF8 collation 352
utilities *See* database utilities
backup 824
collation 829
compression 832
erase 834
information 837
initialization 839
ISQL 846
log translation 849
Open Server Gateway 853
Open Server information 855
Open Server stop 856
remote database extraction 858
remote message ageng 863
SQL preprocessor 892
stop 857, 867
transaction log 869
uncompression 873
unload 876
upgrade 882
validation 885
write file 888

V

valid_name SQL Server function 573
valid_user SQL Server function 573
VALIDATE TABLE statement
syntax 1187
validating databases 886
validating tables 1187
validity checking 133, 214
VARBINARY data type
Transact-SQL 551
VARCHAR data type 668, 918
Transact-SQL 550
variable result sets
from procedures 294, 1010, 1051, 1121
variable result sets from procedures 294
variables 901
about 898, 1031
assigning values to 594, 626
declaring 284
dropping 1060
global 556, 557, 903
in procedures 626
local 556, 594
obtaining values using SELECT 560
SELECT statement and 594, 626
SET statement 594, 626
SET VARIABLE statement 1145
setting values of 290
VERIFY_ALL_COLUMNS

replication option 1164
VERIFY_ALL_COLUMNS option 510
VERIFY_THRESHOLD
replication option 1164
version number
retrieving 1327
ViewPages property 959
views
about 137, 983, 1033
altering 983
and indexes 1007
and security 393
and table permissions 221
and tables 217, 219
and updates 1186
check option 219
creating 138, 217, 1033
creating, in Sybase Central 218
deleting 139, 221
dropping 1053
dropping, in Sybase Central 222
examples 1355
modifying 221
owner 377
permissions 140, 221, 377, 378
system views 1362
updatable 219
update permissions 394
updating 219
using 218
using indexes with 223
viewing 57
working with 217
VIM 458
VIM link 429
VIM message type
about 976
VIM messages
about 1015, 1058
Visual C++
support for 656
VoluntaryBlock property 955, 959
VX-Rexx 802

W

WAIT_FOR_COMMIT option 1150
WaitReadCmp property 955, 959
WaitReadOpt property 955, 959
WaitReadSys property 955, 959
WaitReadTemp property 955, 959
WaitReadUnkn property 955, 959
warm links 767
warnings
error messages 1210
in procedures 300
in triggers 300
SQLSTATE values 1201
WATCOM C/C++
support for 656
Watcom-SQL
and Transact-SQL 606
new features in 18
WATFILE file format 1165, 1167
WEEKS function 949
WHENEVER statement
syntax 1188
WHERE clause 132
and BETWEEN conditions 110
and date comparisons 106
and GROUP BY clause 125
and ORDER BY clause 105
and pattern matching 108
examples 105
in publications 472
SELECT statement 1143
UPDATE statement 129
WHILE statement
control statements 283
syntax 1108
Transact-SQL 622
wide fetches 686
wide inserts 686, 1062
wide puts 686, 1125
window
moving left or right 75
Windows 3.x
and SQL Remote 457
background processing 716
Windows 95
and ISQL 67
and SQL Remote 457
DOS client applications 32, 33
DOS client applications on 32
Windows 3.x client applications 32, 33
Windows 3.x client applications on 32
Windows DLL

for Embedded SQL 701
Windows NT
and ISQL 67
and SQL Remote 457
DOS client applications 32, 33
services 518
Windows 3.x client applications 32, 33
Windows NT DLL
for Embedded SQL 701
WITH GRANT OPTION clause 383
WITH HOLD clause
OPEN statement 1112
Wizards
in Sybase Central 63
WOD50T.DLL 170
WOD50W.DLL 170
write files 42, 890
and deployment 263
erasing 835
write locks 250
WRITETEXT statement
Transact-SQL 597
WSQL HLI
about 654

X

xp_cmdshell system procedure 1326
xp_msver system function 1327
xp_scanf system procedure 1328
xp_sendmail system procedure 1324
xp_sprintf system procedure 1327
xp_startmail system procedure 1323
xp_stopmail system procedure 1325

Y

Y2K 930
year 2000 930
NEAREST_CENTURY option 1160
YEAR function 950
YEARS function 950
YMD function 950