



# Watcom SQL

VERSION 4.0

Copyright © 1988-1994 by Watcom International Corporation.  
All rights reserved.  
First printed and distributed in the United States of America.

Information in this manual may change without notice and does not represent a commitment on the part of Powersoft Corporation.

The software described in this manual is provided by Powersoft Corporation under a Powersoft License agreement. The software may be used only in accordance with the terms of the agreement.

Powersoft Corporation ("Powersoft") claims copyright in this program and documentation as an unpublished work, revisions of which were first licensed on the date indicated in the foregoing notice. Claim of copyright does not imply waiver of Powersoft's other rights.

This program and documentation are confidential trade secrets and the property of Powersoft. Use, examination, reproduction, copying, decompilation, transfer, and/or disclosure to others are strictly prohibited except by express written agreement with Powersoft.

PowerBuilder, Powersoft, and SQL Smart are registered trademarks, and InfoMaker, Powersoft Enterprise Series, PowerMaker, PowerSQL, PowerViewer, and CODE are trademarks of Powersoft Corporation. DataWindow is a proprietary technology of Powersoft Corporation (U.S. patent pending).

1-2-3 is a registered trademark of Lotus Development Corporation. 386 is a trademark of Intel Corporation. ALLBASE/SQL and IMAGE/SQL are trademarks of Hewlett-Packard Company. AT&T Global Information Solutions and TOP END are registered trademarks of AT&T. CICS/MVS, DB2, DB2/2, DRDA, IMS, PC-DOS, and PL/1 are trademarks of International Business Machines Corporation. CompuServe is a registered trademark of CompuServe, Inc. DB-Library, Net-Gateway, SQL Server, and System 10 are trademarks of Sybase Corporation. dBASE is a registered trademark of Borland International, Inc. Graphics Server is a trademark of Bits Per Second Ltd. DEC and Rdb are trademarks of Digital Equipment Corporation. FoxPro, Microsoft, Microsoft Access, MS-DOS, and Multiplan are registered trademarks, and Windows and Windows NT are trademarks of Microsoft Corporation. INFORMIX is a registered trademark of Informix Software, Inc. INTERSOLV, PVCS, and Q+E are registered trademarks of INTERSOLV, Inc. ORACLE is a registered trademark of Oracle Corporation. PaintBrush is a trademark of Zsoft Corporation. PC/SQL-link is a registered trademark, and Database Gateway is a trademark of Micro Decisionware, Inc. Paradox is a registered trademark of Borland International, Inc. SQLBase is a registered trademark of Gupta Corporation. Watcom is a registered trademark of Watcom International Corporation. XDB is a registered trademark of XDB Systems.

December 1994

# About This Manual

## Subject

This manual introduces Watcom SQL and provides information about using Watcom SQL with PowerBuilder and InfoMaker.

It also contains:

- ◆ Information on backup and recovery
- ◆ Hints on improving database performance
- ◆ Description of Watcom SQL's procedures and triggers
- ◆ Program descriptions
- ◆ Syntax and description of the Watcom SQL commands
- ◆ Descriptions of the Watcom SQL system tables

## Audience

This manual is for users who will be accessing Watcom SQL databases from PowerBuilder or InfoMaker. It assumes the reader is familiar with Microsoft Windows 3.x and with basic SQL concepts and syntax.





# Contents

<b>About This Manual</b> .....	iii
<b>1 Introduction</b> .....	1
Overview .....	2
Features .....	3
Benefits .....	4
PowerBuilder and InfoMaker .....	4
<b>2 Database Design</b> .....	5
Planning the database .....	6
The design .....	6
Basic terminology .....	6
The design process .....	8
Step 1: identify tables and relationships .....	8
Step 2: identify the required data .....	11
Step 3: normalize the data .....	13
Step 4: resolve the relationships .....	19
Step 5: verify the design .....	21
Step 6: implement the design .....	22
<b>3 Watcom SQL Architecture</b> .....	25
Single-user .....	26
Multi-user .....	28
<b>4 Using ODBC</b> .....	31
Data sources .....	32
Adding a data source .....	33
Modifying an existing data source .....	35
Removing a data source .....	35
<b>5 Using ISQL</b> .....	37
Starting ISQL .....	38
The ISQL Interface .....	38
Help .....	39
Connecting to the database .....	40
Database structure .....	41

# Contents

Working with ISQL .....	42
Entering commands .....	42
Displaying data .....	42
Scrolling the data window .....	43
Command recall .....	44
Function keys .....	46
Aborting a command .....	46
Available commands for ISQL .....	47
Using the database tools .....	48
Leaving ISQL .....	49
<b>6 Watcom SQL Concepts .....</b>	<b>51</b>
Connecting to a database .....	52
Named connections .....	52
Connection parameters .....	53
Transactions .....	55
Connecting during development .....	56
Connections during execution .....	57
Savepoints .....	58
Two-phase commit .....	59
Understanding the NULL value .....	60
Ensuring database integrity .....	61
Entity integrity .....	61
Referential integrity .....	62
<b>7 Data Types, Functions, Expressions, and Conditions .....</b>	<b>65</b>
Data types .....	66
Functions .....	71
Expressions .....	84
Conditions .....	90
<b>8 SELECT Command Syntax .....</b>	<b>97</b>
Building SELECT statements .....	98
SELECT .....	99
<b>9 Locking and Concurrency .....</b>	<b>103</b>

# Contents

Consistency .....	104
Isolation levels .....	105
Types of locks .....	106
Concurrency .....	107
Transaction blocking .....	107
Deadlock .....	107
Choosing an isolation level .....	109
Primary key generation .....	110
Data definition commands .....	111
Portable computers .....	112
Applying updates .....	112
Large databases .....	112
<b>10 Backup and Recovery .....</b>	<b>115</b>
The need for backups .....	116
Logs .....	117
Checkpoint log .....	117
Rollback log .....	118
Transaction log .....	118
Backups .....	120
Recovery from system failure .....	122
Recovery from media failure .....	123
Media failure on the database file .....	123
Media failure on the transaction log .....	124
<b>11 Improving Performance .....</b>	<b>127</b>
Other factors affecting performance .....	128
Keys .....	129
Indexes .....	130
Optimizing joins .....	131
Sorting .....	132
How the optimizer works .....	133
Self tuning .....	133
Temporary tables .....	134
Using estimates to improve performance .....	135

# Contents

<b>12 User IDs and Permissions</b> .....	137
Granting and revoking user IDs and permissions .....	138
Granting new user IDs .....	138
Granting permissions on tables .....	138
Execute permission on procedures .....	139
DBA and resource authority .....	139
User groups .....	140
Creating user groups .....	140
Group permissions .....	140
Group tables .....	140
An example of user groups .....	141
<b>13 Views</b> .....	143
Defining a view .....	144
Using views for security .....	146
<b>14 Procedures and Triggers</b> .....	147
Overview of procedures and triggers .....	148
Advantages .....	149
Using procedures .....	150
Creating procedures .....	150
Dropping a procedure .....	151
Calling procedures .....	151
Permission to execute procedures .....	152
Statements .....	153
Compound statements in procedures and triggers .....	153
SQL statements in procedures and triggers .....	154
Atomic statements .....	154
Control statements .....	155
Triggers .....	158
Creating triggers .....	158
Dropping a trigger .....	160
Executing triggers .....	160
Warnings in procedures and triggers .....	161
Errors in procedures and triggers .....	162
Without exception handlers .....	162

# Contents

With exception handlers .....	163
Transactions and savepoints .....	165
Single row SELECT .....	166
Cursors in procedures and triggers .....	167
Result sets from procedures .....	170
Multiple result sets .....	171
<b>15 Database Collations</b> .....	<b>173</b>
Collations .....	174
Countries, languages, and code pages .....	175
Form of the custom collation file .....	177
<b>16 Command Syntax</b> .....	<b>181</b>
Conventions .....	182
Language elements .....	183
ALTER DBSPACE .....	186
ALTER TABLE .....	187
CALL .....	192
CASE .....	193
CHECKPOINT .....	194
CLOSE .....	195
COMMENT .....	196
COMMIT .....	197
Compound statements .....	199
CONFIGURE .....	201
CONNECT .....	202
CREATE DBSPACE .....	204
CREATE INDEX .....	205
CREATE PROCEDURE .....	207
CREATE TABLE .....	209
CREATE TRIGGER .....	217
CREATE VARIABLE .....	219
CREATE VIEW .....	221
DBTOOL .....	223
DECLARE CURSOR .....	227
DECLARE TEMPORARY TABLE .....	230

# Contents

DELETE .....	231
DELETE (positioned) .....	232
DISCONNECT .....	233
DROP .....	234
DROP OPTIMIZER STATISTICS .....	235
DROP VARIABLE .....	236
EXIT .....	237
FETCH .....	238
FOR .....	241
FROM .....	243
GRANT .....	249
HELP command .....	253
IF statement .....	254
INPUT .....	255
INSERT .....	259
LEAVE .....	261
LOOP .....	262
NULL value .....	263
OPEN .....	265
OUTPUT .....	267
PARAMETERS .....	270
PREPARE TO COMMIT .....	271
READ .....	272
RELEASE SAVEPOINT .....	274
RESIGNAL .....	275
RESUME .....	276
REVOKE .....	277
ROLLBACK .....	279
ROLLBACK TO SAVEPOINT .....	280
SAVEPOINT .....	281
SET CONNECTION .....	282
SET OPTION .....	283
SET variable .....	298
SIGNAL .....	299
UNION .....	300
UPDATE .....	302

# Contents

UPDATE (positioned) .....	304
VALIDATE TABLE .....	305
<b>17 Program Summary</b> .....	<b>307</b>
DBBACKUP .....	308
DBCOLLAT .....	311
DBERASE .....	313
DBEXPAND .....	314
DBINFO .....	315
DBINIT .....	317
DBLOG .....	322
DBSHRINK .....	323
DBSTART .....	324
DBSTOP .....	330
DBTRAN .....	331
DBUNLOAD .....	333
DBUPGRAD .....	336
DBVALID .....	338
DBWRITE .....	339
ISQL .....	341
Environment variables .....	346
Software component return codes .....	349
<b>A Watcom SQL Features</b> .....	<b>351</b>
Differences from other SQLs .....	352
<b>B Limitations</b> .....	<b>355</b>
<b>C Database Error Messages</b> .....	<b>357</b>
Alphabetic by error message .....	358
Alphabetic by SQLSTATE .....	363
Error message descriptions .....	368
Warnings .....	368
Environment errors .....	370
Connection errors .....	374
Creation errors .....	377

# Contents

Permission errors .....	381
Prepare errors .....	382
Semantic errors .....	383
Expression and function errors .....	388
Describe errors .....	389
Open errors .....	390
Fetch errors .....	390
Update and insert errors .....	392
Variable errors .....	396
Procedure errors .....	396
Option errors .....	399
Concurrency errors .....	400
Savepoint errors .....	401
Version checking errors .....	402
Backup errors .....	403
Miscellaneous errors .....	404
User interruption .....	405
Errors that cause a rollback .....	405
Errors specific to WSQL HLI .....	407
Internal errors (assertion failed) .....	409
<b>D Watcom SQL Keywords .....</b>	<b>411</b>
<b>E Watcom SQL System Tables .....</b>	<b>413</b>
SYS.SYSUSERPERM .....	415
SYS.SYSGROUP .....	417
SYS.SYSFILE .....	418
SYS.SYSTABLE .....	419
SYS.SYSDOMAIN .....	421
SYS.SYSCOLUMN .....	422
SYS.SYSINDEX .....	424
SYS.SYSIXCOL .....	426
SYS.SYSFOREIGNKEY .....	427
SYS.SYSFKCOL .....	429
SYS.SYSTABLEPERM .....	430
SYS.SYSCOLPERM .....	433



# Contents

SYS.SYSOPTION .....	434
SYS.SYSINFO .....	435
SYS.SYSCOLLATE .....	437
SYS.DUMMY .....	438
SYS.SYSPROCEDURE .....	439
SYS.SYSTRIGGER .....	440
SYS.SYSPROCPARM .....	442
SYS.SYSPROCPERM .....	444
<b>F Watcom SQL System Views .....</b>	<b>445</b>
SYS.SYSCATALOG .....	446
SYS.SYSCOLUMNS .....	447
SYS.SYSVIEWS .....	448
SYS.SYSINDEXES .....	449
SYS.SYSFOREIGNKEYS .....	450
SYS.SYSUSERAUTH .....	451
SYS.SYSUSERPERMS .....	452
SYS.SYSUSERLIST .....	453
SYS.SYSGROUPS .....	454
SYS.SYSTABAUTH .....	455
SYS.SYSCOLAUTH .....	456
SYS.SYSOPTIONS .....	457
SYS.SYSUSEROPTIONS .....	458
SYS.SYSTRIGGERS .....	459
SYS.SYSPROCPARMS .....	460
SYS.SYSPROCAUTH .....	461



## CHAPTER 1

# Introduction

About this chapter	This chapter introduces Watcom SQL, outlining its features and benefits.
Contents	<ul style="list-style-type: none"><li>◆ "Overview" on page 2</li><li>◆ "Features" on page 3</li><li>◆ "Benefits" on page 4</li></ul>

## Overview

Watcom SQL is a complete **relational** database system that runs under Windows on your PC. Watcom SQL conforms to the ANSI SQL89 standard but has many additional features defined in the IBM DB2 and SAA specification and in ANSI SQL92.

# Features

Among the Watcom SQL features are:

- ◆ Primary and foreign key support
- ◆ Security to restrict access to your data
- ◆ Backup and recovery using logs
- ◆ Integrity checking
- ◆ Automatic row-level locking
- ◆ Stored procedures and triggers
- ◆ Blob support
- ◆ 32-bit processing

In addition, Watcom SQL is easy to install and tunes itself.

## **Benefits**

Since Watcom SQL runs on your PC, you can easily work with the same data at home, on your laptop, and in the office. Watcom SQL is available as a single-user local database, as in this package, or as a network server. And the single-user applications you develop can be deployed without a deployment fee.

## **PowerBuilder and InfoMaker**

PowerBuilder and InfoMaker access and manipulate data in Watcom SQL databases through the ODBC interface. If you choose the default installation of PowerBuilder or InfoMaker you will be connected to a Watcom database.

From within PowerBuilder or InfoMaker applications, you can create and maintain databases; create, drop, and update tables; alter the table definition; and create, update, and drop views. In addition, you can retrieve data from the database, modify it, and then update the database with your changes.

To do all this from within PowerBuilder or InfoMaker, you just make selections in the painters. PowerBuilder and InfoMaker use your selections to generate the required SQL statements (commands) and then submit them to Watcom SQL for execution.

In PowerBuilder, you can also create scripts that have embedded SQL to access and manipulate data in the database. You can use the Database Administration painter (DBA notepad) to execute SQL commands immediately.

## CHAPTER 2

# Database Design

- About this chapter      Watcom SQL is a **relational** database system.
- Before you begin to design your database, you should have some understanding of relational database concepts. This chapter briefly describes these concepts.
- Before you begin      If you are not familiar with relational databases, you may wish to consult an introductory book such as *A Database Primer* by C. J. Date. If you are interested in database theory, C. J. Date's *An Introduction to Database Systems* (fourth edition) is an excellent textbook on the subject.
- Contents
- ◆ "Planning the database" on page 6
  - ◆ "The design process" on page 8

## Planning the database

The most important consideration in designing your database is how the information will be used. The various applications and procedures that will use the database introduce requirements upon the structure of the data.

In a relational database, you represent the data and data relationships as a collection of tables. Each table has one or more columns.

The first step in creating a database is designing it: you plan what tables you require and what data they will contain. You also determine how the tables are related. This is a very important step and deserves careful consideration.

You must determine what things you want to store information about (entities) and how these things are related (relationships). A useful technique in designing your database is to draw a picture of your tables as shown later in this chapter. 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.

## The design

The data you need in order to create a good database design comes from the:

- ◆ Business activities you will use the database to perform
- ◆ Business rules that apply to these activities
- ◆ Data you want to maintain in the database

When you complete your database design, you will have a diagram of your database. The diagram, along with the business activities and rules, provides all the information you need to implement the database.

## Basic terminology

This table lists some of the relational database terms and their equivalent in other nonrelational databases:



Formal relational term	Informal relational term	Equivalent nonrelational term
Relation	Table	File
Attribute	Column	Field
Tuple	Row	Record

**Tables** Database tables are sometimes called entities. They are the database equivalent of nouns:

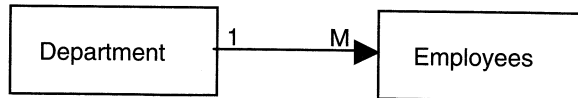
- ◆ People, places, things
- ◆ Events, activities

For example, in keeping track of information about employees, the subject is **employees**, and employees becomes a table.

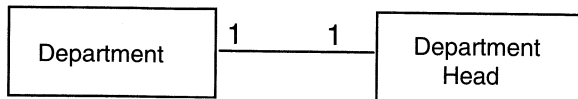
A table contains information on a particular topic and is made up of **columns** and **rows**. A column is named and contains related information. Each row in the table has one value in each column in the table. Rows are not named.

**Relationships** A relationship is the database equivalent of a verb. For example, an employee is **associated** with a department. Tables can be related to one another in three ways:

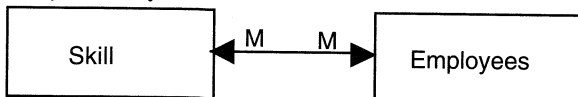
- ◆ One-to-many



- ◆ One-to-one



- ◆ Many-to-many



## The design process

There are six major steps in the design process. The first five steps are usually done on paper and then the final design is implemented.

- 1 Identify tables and relationships
- 2 Identify data that is needed for each table and relationship
- 3 Normalize the data
- 4 Resolve relationships
- 5 Verify the design
- 6 Implement the design using PowerBuilder or InfoMaker

### Step 1: identify tables and relationships

To identify the tables and their relationship to each other:

- 1 Define high-level activities  
Identify the general activities you will use this database for. For example, you want to keep track of information about employees.
- 2 Identify tables  
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 only one department head; the department head is unique.

Example

ACME Corporation is a small company with offices in five locations. Currently 71 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.

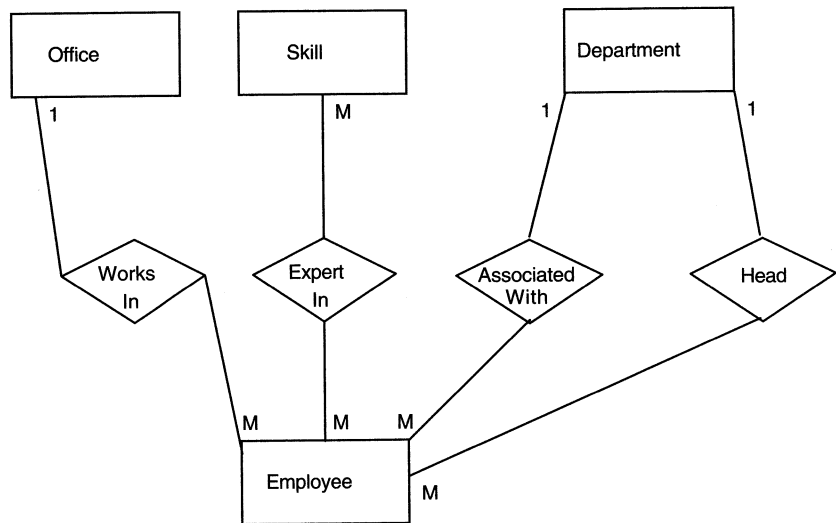
Some of the high-level activities for ACME Corporation are:

- ◆ 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

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. At this point we can also identify which relationships are one-to-many, one-to-one, and many-to-many.

Right now, this is a rough E-R diagram. It will be refined throughout the chapter.



The next step is to look at each high-level activity and see if it is really made up of one or more lower-level activities. For example, the lower-level activities below are based on the high-level activities listed earlier:

- ◆ 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

Use these lower-level activities to identify any new tables or relationships.

Examine the business rules to identify where these activities impact your database design. Business rules often identify one- to-many, one-to-one, and many-to-many relationships.

- ◆ 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.

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.

- 2 Set up data for each table.

List the available data for each table as it seems appropriate right now.

- 3 Set up data for each relationship.

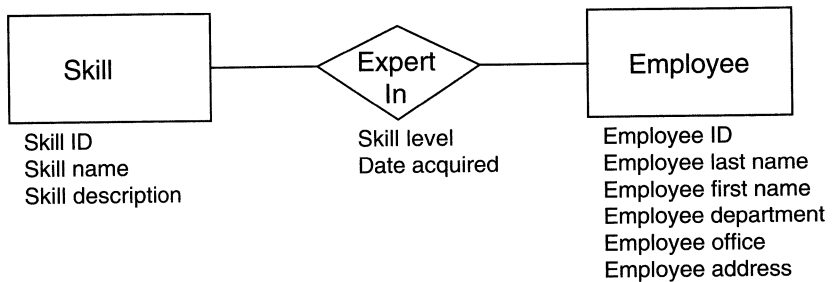
List the data that applies to each relationship (if any).

## 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 diagram this data, your diagram 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 against the data to eliminate redundancy 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 information about the normalization test, 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:
  - ◆ Identify at least one key for each table. Each table must have a primary key.
  - ◆ Identify keys for relationships. The keys for a relationship are the keys from the two tables it joins.
  - ◆ Check for calculated data in your supporting data list. Calculated data is not normally stored in the database.
- 2 Put data in first normal form:
  - ◆ Remove repeating data from tables and relationships.
  - ◆ Create one or more tables and relationships with the data you remove.
- 3 Put data in second normal form:
  - ◆ Identify tables and relationships with more than one key.

- ◆ Remove data that depends on only one part of the key.
  - ◆ Create one or more tables and relationships with the data you remove.
- 4 Put data in third normal form:
- ◆ Remove data that depends on other data in the table or relationship and not on the key.
  - ◆ 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:



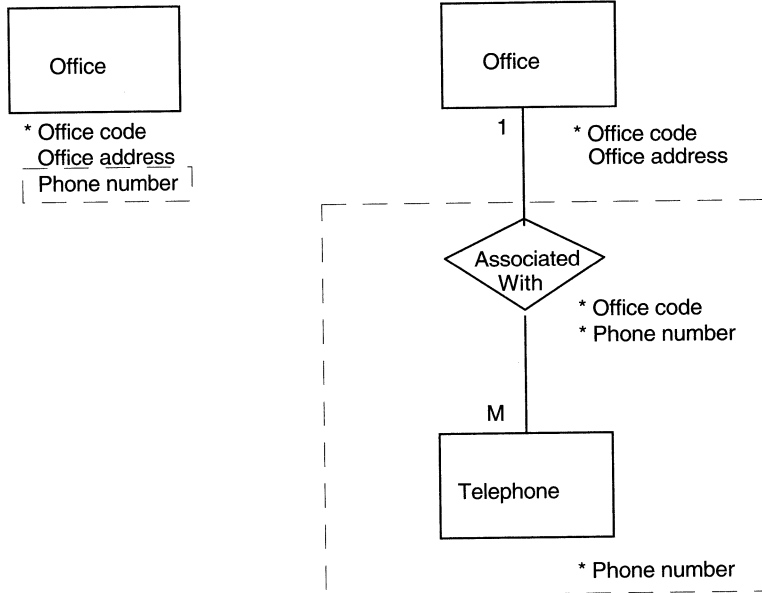
Relationship	Key
Office	*Office code Office address Phone number
Works in	*Office code *Employee ID
Department	*Department ID Department name
Heads	*Department ID *Employee ID
Assoc with	*Department ID *Employee ID
Skill	*Skill ID Skill name Skill description
Expert in	*Skill ID *Employee ID Skill level 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.

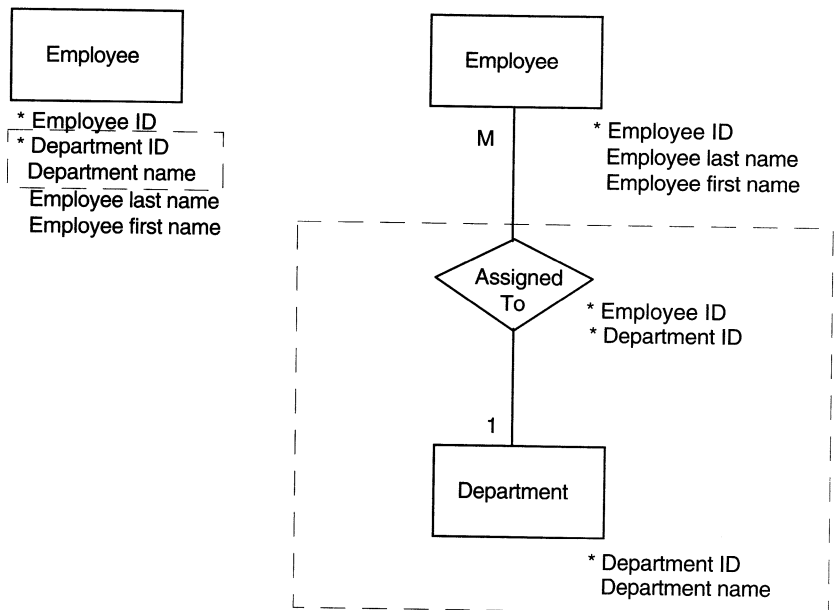


## Putting data in second normal form

- ◆ 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 two keys. The data, however, does not depend on the whole key; it depends only one of those keys (Employee ID). Therefore, the Department ID, which the 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.

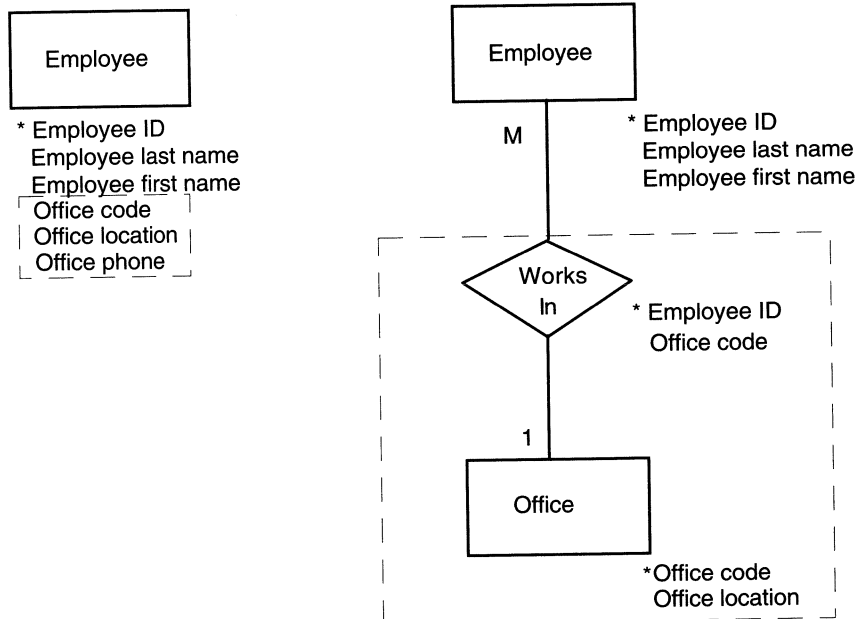


## Putting data in third normal form

- ◆ 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.

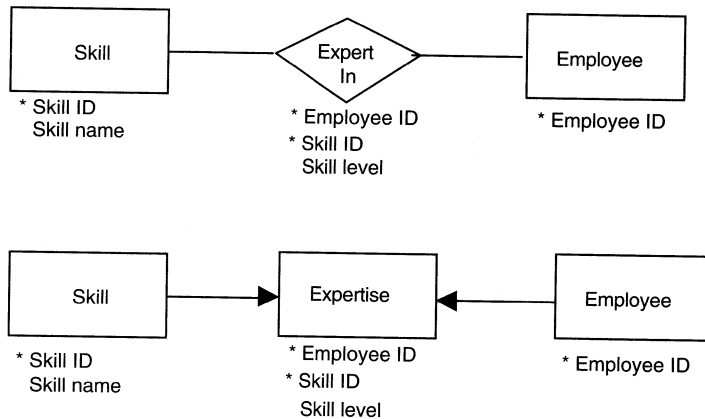


## Step 4: resolve the relationships

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.



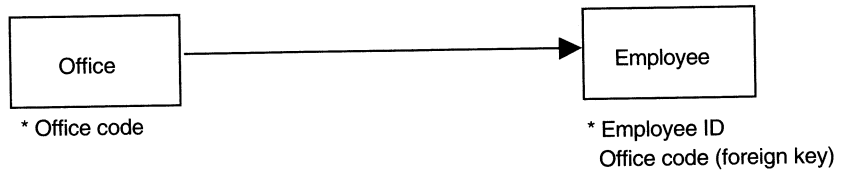
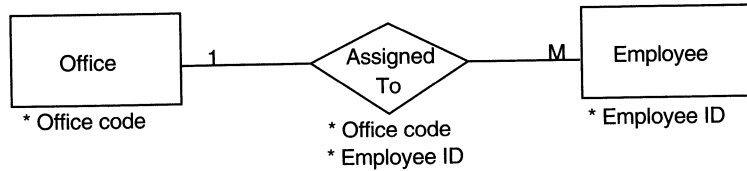
When this is the case, change the relationship to a table. The key to the new table remains the same as it was for the relationship.

### Resolving relationships that do not carry data

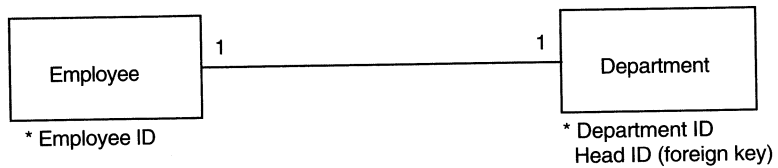
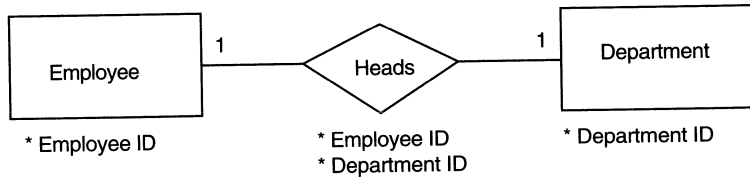
In order to implement relationships that do not carry data, you need to define **foreign keys**. A **foreign key** is a column or set of columns that contains primary key values from another table. The foreign key allows you to access data from more than one table at one time.

There are some basic rules that help you decide where to put the keys:

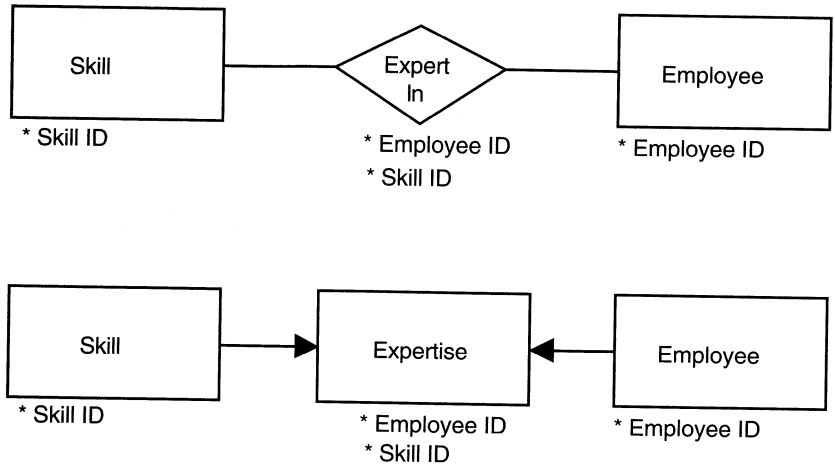
**One to many** In a one-to-many relationship, the primary key in the **one** is carried in the **many**. In this example, the foreign key goes into the Employee table.



**One to one** In a one-to-one relationship, the foreign key can go into either table. In this example, the foreign key (Head ID) is in the Department table.



**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

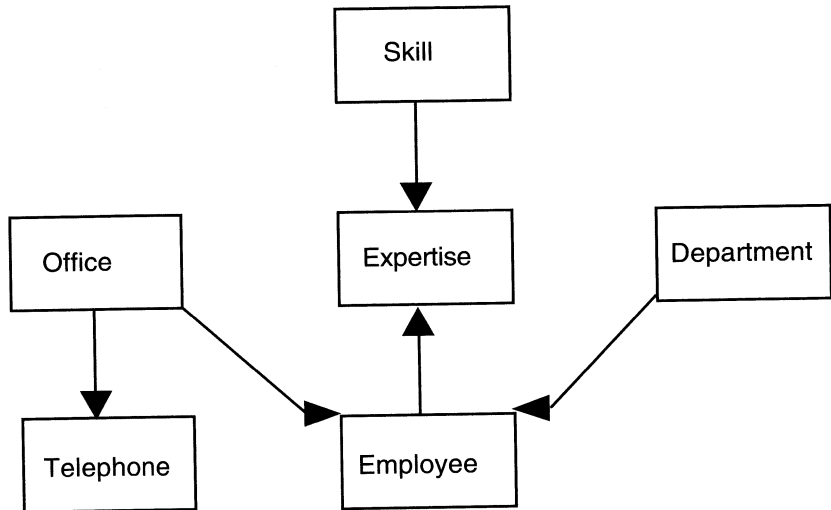
Before you implement your design, you need to make sure it will support your needs. Examine the activities you identified at the start of the design process and make sure you can access all the data the activities require:

- ◆ 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 will look like this:



## Step 6: implement the design

The final step is to implement this design using the PowerBuilder or InfoMaker database painter. You will name the tables and the columns containing the data to be stored in each table and specify data type and other information (such as display format) for each column.

During the design process, you decided what tables you needed and what data you wanted in each table. Now you need to select a column name for each column of data, specify the data type and size for the column, and decide whether you will allow NULL values and whether you want the database to restrict the values allowed in the column.

### Column name

A column name can be any set of letters, numbers, or symbols. However, if a column name contains characters other than letters, numbers, or underscores, or does not begin with a letter, or is a keyword (see "Watcom SQL Keywords" on page 411), it must be enclosed in double quotation



marks. Whenever you use such a column name, you must enclose it in double quotation marks.

## Data type and size

The valid types of data supported by Watcom SQL are:

- ◆ Integer (int, integer, smallint)
- ◆ Decimal (decimal, numeric)
- ◆ Floating point (float, double, real)
- ◆ Character (char, varchar, long varchar)
- ◆ Binary (binary, long binary)
- ◆ Date/time (date, time, and timestamp)

Int, decimal, and real are not available in PowerBuilder; you should use one of the other synonyms for these data types.

The type of column will affect its maximum size. 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. For a complete description of data types, see "Data types" on page 66.

## NULL versus NOT NULL

When the column value is mandatory for a row, you define the column as being NOT NULL. Otherwise, the column is allowed to contain the NULL value, which represents no value. The default in SQL 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 and its use in comparisons, see "Conditions" on page 90 and "NULL value" on page 263.

## Restrictions

Although the data type of a column restricts the values allowed in that column (for example, only numbers or only dates), you often want to further restrict the allowed values. You can restrict the values of any column by specifying a CHECK constraint or specifying valid extended attributes. You can use any valid condition that could appear in a WHERE clause to restrict the allowed values.

## **For more information**

For more information about creating a database in PowerBuilder or InfoMaker, see the PowerBuilder or InfoMaker *User's Guide*.

## CHAPTER 3

# Watcom SQL Architecture

### About this chapter

Your copy of PowerBuilder or InfoMaker contains a single-user version of Watcom SQL. Watcom SQL is also available in a Network Server package for a variety of operating systems.

This chapter describes the architecture of single-user Watcom SQL, as well as the architecture of the Watcom SQL Network Server. The latter may be of use as you migrate your applications to a network setting.

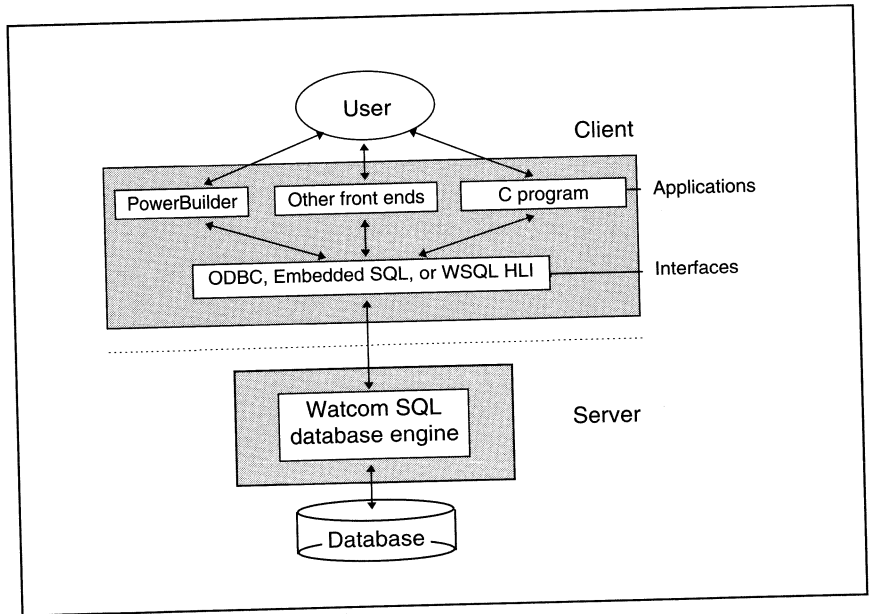
### Contents

- ◆ "Single-user" on page 26
- ◆ "Multi-user" on page 28

# Single-user

This is the version of Watcom SQL included in your package.

Watcom SQL is composed of several software components. There is a database engine that runs as a separate program on the computer. This engine does not interact directly with the user but manages and provides the access to databases stored on disk. The application system communicates with the database engine and presents those results to the user. Several applications can use the database engine at the same time.



There are three application systems shown in the figure:

**PowerBuilder** PowerBuilder uses the ODBC interface to communicate with Watcom SQL.

**Other front ends** Many other leading database front ends available today support Watcom SQL through the ODBC or embedded SQL interface. These include application development systems, report generators, and ad hoc query tools.

**C program** There are header files and libraries that allow you to program to the ODBC, embedded SQL or WSQL HLI interface.

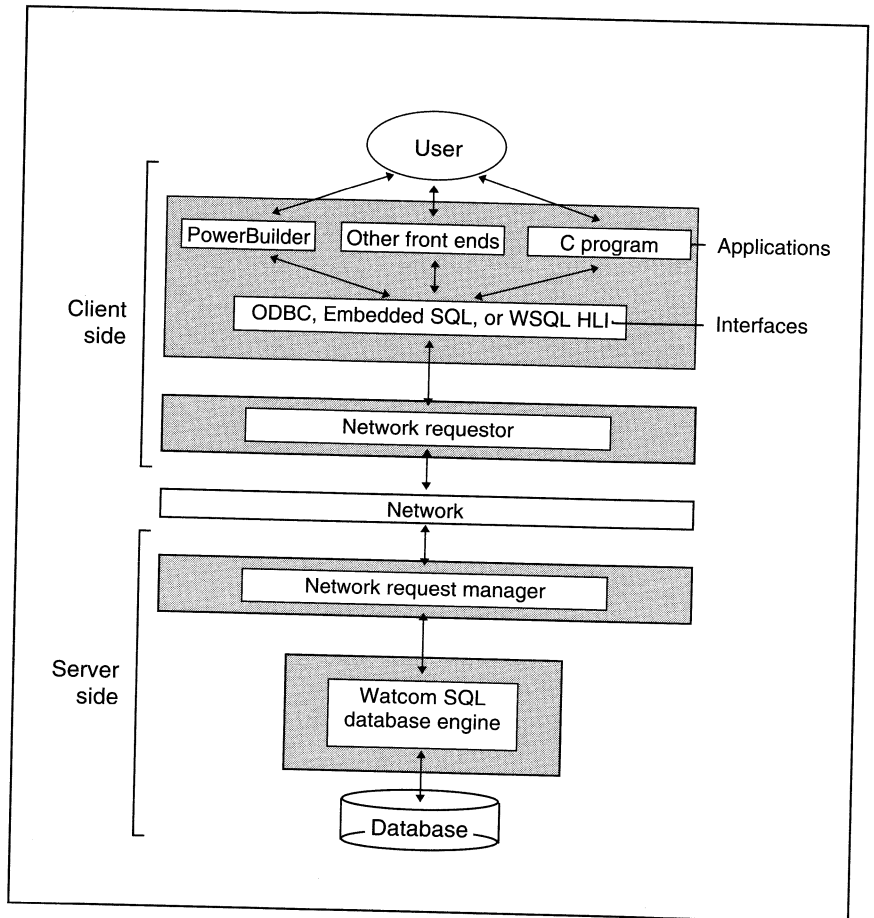
The lowest level interface to the database engine is Embedded SQL. The ODBC interface is an alternative to Embedded SQL which allows you to use the Microsoft standard ODBC API to access the database. The WSQL HLI (high level interface) is a simple programming interface for use in programming environments that will allow you to call dynamic link libraries.

## Multi-user

Watcom SQL can also be used by multiple users, over a **local area network** (LAN). In this case, the database engine runs on one computer and PowerBuilder, front ends, or custom C programs run on client computers. The client computers send SQL requests over the network to the database server and the database server sends the results of the requests back to the client. The requests are handled by two Watcom SQL components: the **Network Requestor** and the **Network Request Manager** (see the drawing below). For a complete description of all components in the Watcom SQL product, see "Program Summary" on page 307.

The following packages support multi-user network access to Watcom SQL:

- ◆ Watcom SQL Network Server for DOS
- ◆ Watcom SQL Network Server for OS/2
- ◆ Watcom SQL Network Server for NetWare
- ◆ Watcom SQL Network Server for Windows
- ◆ Watcom SQL Network Server for Windows NT
- ◆ Watcom SQL for QNX
- ◆ Watcom SQL Network Server for QNX







## CHAPTER 4

# Using ODBC

### About this chapter

The **Open Database Connectivity** (ODBC) interface, defined by Microsoft Corporation, is a standard interface to database management systems in the Windows and Windows NT environments. Applications such as PowerBuilder use the ODBC interface to access a wide range of database systems.

While PowerBuilder takes care of ODBC interface details for you, it is useful to have an understanding of ODBC concepts.

Watcom SQL provides an **ODBC Driver** which conforms to the ODBC interface, giving ODBC applications access to information in Watcom SQL databases. This chapter describes how to set up ODBC data sources so that ODBC-compliant applications will be able to access Watcom SQL.

### Contents

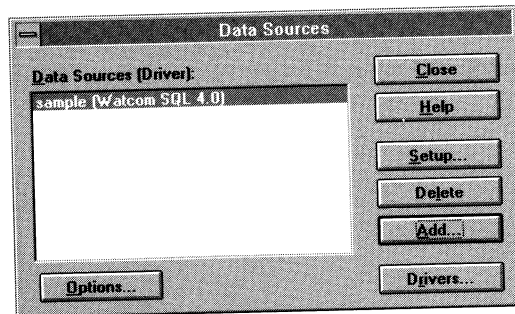
- ◆ "Data sources" on page 32

## Data sources

ODBC requires a description of every **data source** (that is, database) available. That description includes information about the Database Management System (DBMS), the location of the database files, and other DBMS-dependent information. When an application connects to a data source, ODBC uses these descriptions to load appropriate programs to access the database. These descriptions are contained in the ODBC.INI file in Windows or the Windows NT registry. ODBC includes an **Administrator** program which manages the **data sources** for the system. This enables users to configure ODBC drivers into their environment, allowing applications to connect to different data sources.

The Watcom SQL for Windows installation adds the description for the sample database to the ODBC.INI file. The Watcom SQL for Windows NT installation adds this information in the Windows NT registry.

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 installation process installs the ODBC Administrator as one of the icons in your Watcom SQL group. The ODBC Administrator can also be activated through the Control Panel.



The left side of the Administrator window lists the available data sources. 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 drivers, and allow you to install new drivers or remove drivers.

There are three actions available: adding a new data source, removing a data source, and modifying an existing data source.

## Adding a data source

The ODBC driver can access both Watcom SQL database files on your hard disk and Watcom SQL network database servers (if you have one of the Network Server packages).

### Database must exist

To add a data source for a database file, the database must already exist. You need to create a database before using the ODBC Administrator program to add a data source for the database.

If you want to add a new data source, press the Add button. You will be presented with a list of the available drivers. Select the Watcom SQL driver from the list and press the OK button. You will be presented with the following dialog box:

The Watcom SQL ODBC Configuration dialog box contains the following fields. These fields correspond to the connection parameters. See "Connection parameters" on page 53 for a description of the connection parameters and a description of the manner in which they are used to establish a connection with a database.

### 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 it is omitted, most ODBC applications will 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 will prompt you to enter your password when connecting to the data source.

### **Server Name**

The name of a Watcom SQL database engine or the name of a Watcom SQL network server. If not specified, the default local engine is used (the first database engine started). This field corresponds to the EngineName connection parameter.

### **Database Alias**

If specified, this corresponds to the name of a database already running on a Watcom SQL database engine or Watcom SQL 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:\WSQL\SAMPLE.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.

### **Prevent Driver not Capable Errors**

The Watcom SQL 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.

## Modifying an existing data source

To modify an existing data source, select the data source and press the **Setup** button. You can modify any of the attributes set when the data source was added.

## Removing a data source

To remove a data source, select the data source and press the **Delete** button. You will be 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.



## CHAPTER 5

# Using ISQL

### About this chapter

ISQL (Interactive SQL) is an interactive interface to Watcom SQL databases. With ISQL you can connect to a database, type SQL statements and send them directly to the database, and launch tools for Watcom SQL databases.

This chapter introduces you to running ISQL. SQL itself is not discussed here.

You will gain the most from this chapter if you run ISQL on your computer as you work through this chapter.

### Contents

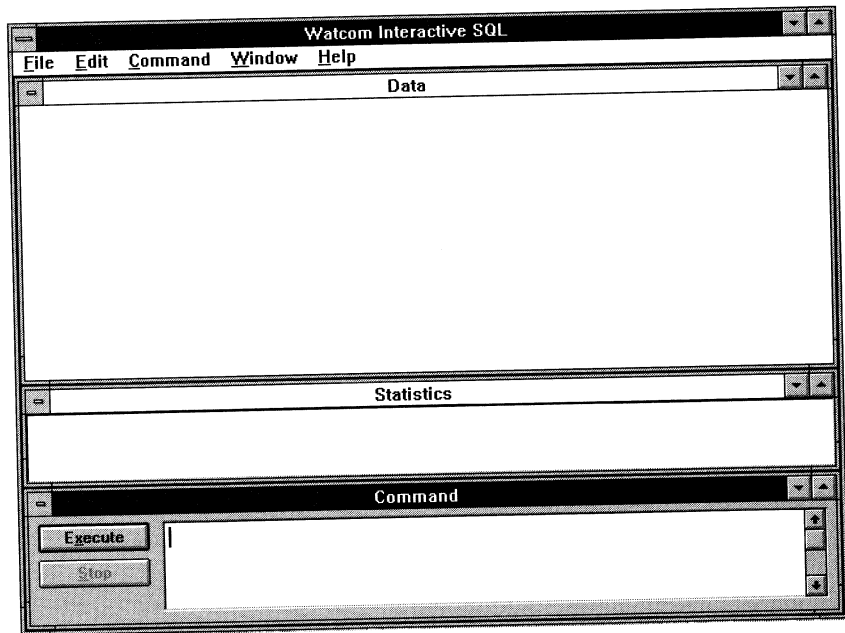
- ◆ "Starting ISQL" on the next page
- ◆ "Connecting to the database" on page 40
- ◆ "Working with ISQL" on page 42
- ◆ "Using the database tools" on page 48
- ◆ "Leaving ISQL" on page 49

## Starting ISQL

On installation, PowerBuilder and InfoMaker create a Powersoft program group in Program Manager. This group contains an icon to launch the Watcom SQL database engine on a sample database (PSDemoDB), and an icon to launch ISQL.

First launch the sample database by double clicking on the demo DB icon. A window appears, displaying some startup information. After a few seconds, the program 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 on your screen, as in the figure.



## The ISQL Interface

The ISQL main window contains three text windows.

- ◆ **The command window** This is the area where you type SQL commands and queries to send to the database, as well as instructions to ISQL itself.



It is a standard Windows edit control. If more lines are typed than will fit in this window, the window automatically scrolls.

- ◆ **The statistics window** This window displays information such as the number of rows returned on a query.
- ◆ **The data window** This window displays the rows of a database that are returned from a query.

You can scroll each of these windows using the cursor keys or the scroll bar on the right side of the window. These windows can also be made larger and maximized to full screen size in the standard Windows fashion. See the Microsoft Windows User's Guide for more information on controlling windows and working with text.

## Help

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. For more information on using help, choose Using Help from the Help menu.

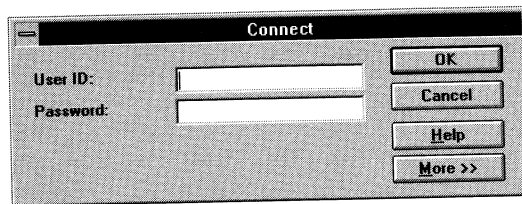
## Connecting to the database

ISQL is not yet connected to the database. To connect to the PSDemoDB database that you have already started running, type the command

```
connect
```

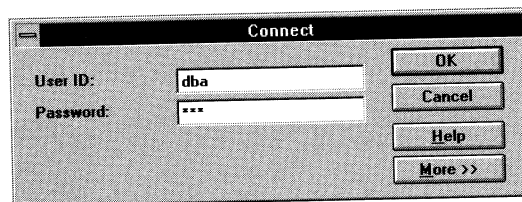
in the command window, and click on the button labelled **Execute**.

ISQL will ask you to enter a user identification (**user ID**), and **password** as shown in the figure.

A screenshot of a 'Connect' dialog box. The title bar reads 'Connect'. On the left, there are two input fields: 'User ID:' and 'Password:'. On the right, there are four buttons: 'OK', 'Cancel', 'Help', and 'More >>'. The dialog box has a standard Windows-style border with a close button in the top-left corner.

Type in the three-letter user ID DBA, and press the **Tab** key to move to the next field.

Type in the three-letter password for the DBA user ID: SQL. The password does not appear when you type it. This prevents anyone else from seeing your password.

A screenshot of the 'Connect' dialog box, similar to the one above. The 'User ID:' field now contains the text 'dba'. The 'Password:' field contains three asterisks '\*\*\*'. The buttons and layout are the same as in the previous screenshot.

### The DBA user ID

Watcom SQL databases are always created with user ID "DBA" and password "SQL." The PSDemoDB database was created under the user ID DBA.

The connect dialog contains a button labelled **More>>**. Pressing this button will reveal a larger dialog box which contains more options for connecting to the database engine. For the purposes of this tutorial, pressing this button is

not necessary. For more information about the Connect command, see the description in "Command Syntax" on page 181.

Press Enter (or click the OK button) to connect to the database.

If you have made typing mistakes or if the demo 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 successfully connect to the database, the statistics window should display the message "Connected to database".

## Database structure

Now that you have connected, you can explore the structure of the active database. The demo database concerns a fictional small company. It contains internal information about employees and departments, as well as information about contacts and orders. All of this information is organized into a number of **tables**. Select Insert Table... from the Edit menu, or press the F7 key to display a dialog box listing the tables in the current database. The table names are of the form DBA.table\_name. The DBA prefix indicates that this database is owned by user ID DBA.

In addition to the tables that are part of the database, there are several **system tables** listed, which maintain information about the state of the database itself.

Select the DBA.department table with the mouse, and press the Details button. This produces a listing of the columns in that table.

These tables can be inserted into commands in the command window. For now, however, simply press the Cancel button on each dialog box to clear them, without taking any actions.

## Working with ISQL

### Entering commands

Commands are typed into the command window, and executed by pressing the execute key (F9) or by clicking the Execute button.

Multiple commands can be entered at one time by separating them with a semicolon. Commands can also be stored in a text file or loaded from a text file, by choosing Save or Open from the File menu, respectively.

### Displaying data

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

For example, to look at the entire contents of the table called DBA.employee, type

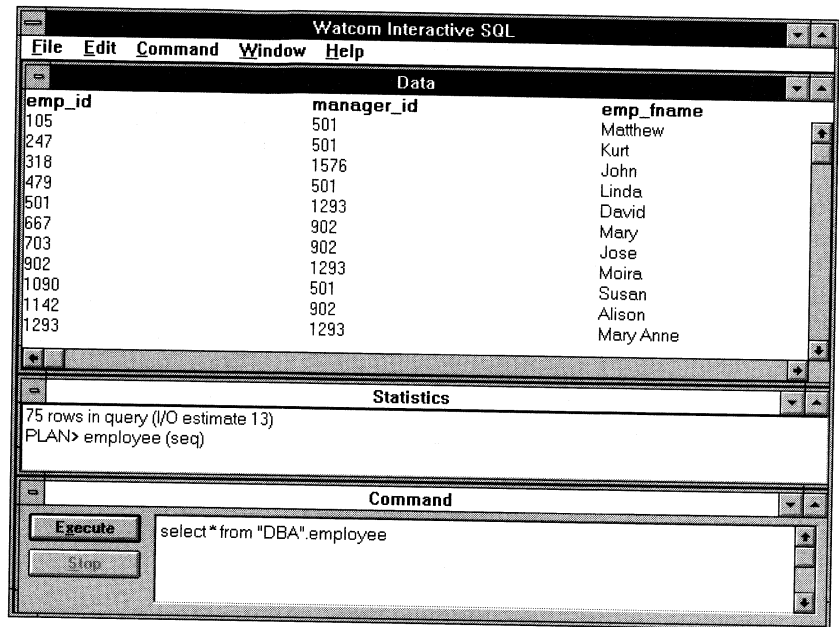
```
select * from "DBA".employee
```

and press the Execute key (F9) or click the Execute button.

#### **Keywords**

The double quotes around DBA in the above command are necessary because DBA is a SQL **keyword**.

The query is sent to the Watcom SQL database engine, which executes it, and returns the results of the query to the ISQL data window, as shown below. (Your screen may be somewhat different, depending on your video configuration.)



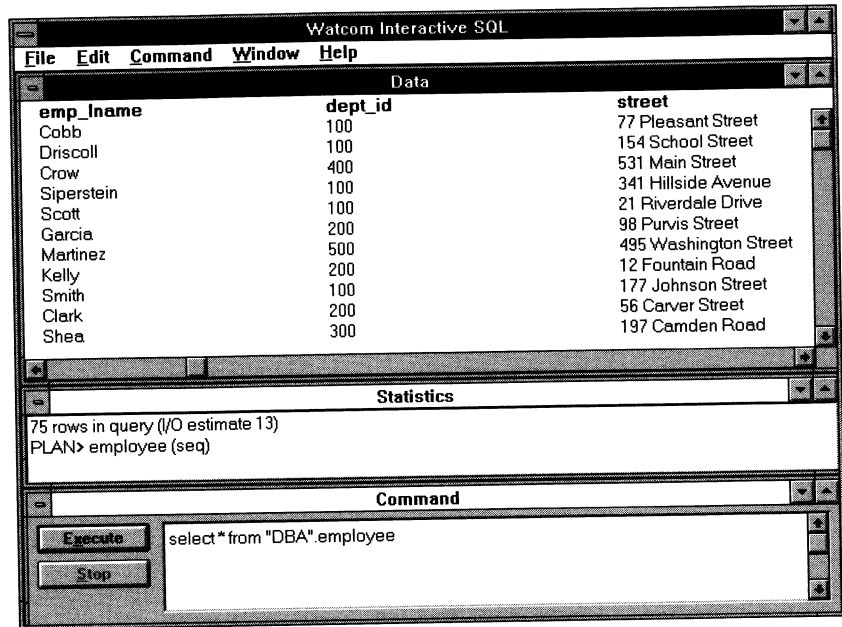
If you make a spelling mistake, use the standard edit keys for correcting the problem.

## Scrolling the data window

Not all of the DBA.employee table fits into the data window. There is more information about each of these employees, and there are more employees than fit on the screen.

Use the scroll bar at the bottom of the data window to pan left and right across the information about each employee.

Scroll the window to the right to see the rest of the information that is held about each employee visible in the list. Scroll back to the left by scrolling the window in the opposite direction until the emp\_id column appears again.



To see information on other employees, use the scroll bar to the right of the data window. Scroll the window down to see more employees in the table. Continue scrolling the window until the data no longer scrolls.

The vertical scroll bar is slightly differently to a standard scroll bar. If the number of rows in the result of a command or query is unknown, a guess as to the number of rows is used to determine how far to scroll. If ISQL determines that its guess is wrong, the guess will be adjusted and the slider will "jump".

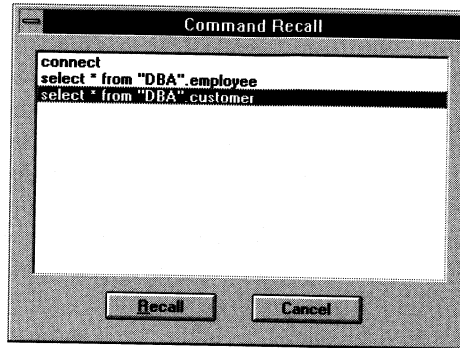
## Command recall

Now list the contents of one of the other tables in the database; the DBA.customer table, Type the following and then press **F9**:

```
select * from "DBA".customer
```

The contents of the customer database table are displayed in the data window, and the results of the previous query are cleared.

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 shown below.



The command recall window displays the first line of the last fifteen commands that you have executed. Use the cursor up and down keys ( $\uparrow$  and  $\downarrow$ ) to scroll through the commands.

If you position the cursor on the second command that you executed, which was:

```
select * from "DBA".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 you can modify it to make a new command.

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

- |               |   |
|---------------|---|
| <b>Ctrl+R</b> | bring up the command recall window.   |
| <b>Ctrl+P</b> | cycles backwards through previously executed commands. Retrieved commands are placed into the command window. |
| <b>Ctrl+N</b> | cycles forward through previously executed commands.  |

## Function keys

ISQL uses some function keys and special keys as follows:

**F1** Help.

**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 on the **Execute** button.

## Aborting a command

The **Stop** button is used to abort a command.

An abort operation will stop current processing and prompt for the next command. If a command file was being processed, you will be prompted for an action to take (Stop command file, Continue, or Exit ISQL). These actions can be controlled with the `ON_ERROR` ISQL option (see "SET OPTION" on page 283).

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 will be displayed:

```
ISQL command terminated by user
```

ISQL will stop 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 will be 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.

This message will also occur 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 will cancel not only the current command, but everything that has been done since the last COMMIT. In some cases, it will take a considerable amount of time for the database engine to perform the automatic ROLLBACK.

- 3 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 will be displayed.

`Statement interrupted by user`

The effects of the current command will be undone.

## Available commands for ISQL

You can enter any SQL statement supported by Watcom SQL in ISQL, as well as several other commands which are available only within ISQL. Two such commands are INPUT and OUTPUT, which allow you to put data into a &dbdame database from other formats, and write Watcom SQL data out into other formats for use elsewhere.

For more information about the commands supported by ISQL, see "Command Syntax" on page 181

## **Using the database tools**

ISQL provides a convenient platform for using the set of database tools that are provided with Watcom SQL.

Select Database Tools from the Window menu to display the DBTOOLS dialog box. From this dialog box you can run the utility programs described in "Program Summary" on page 307.

## Leaving ISQL

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



## CHAPTER 6

# Watcom SQL Concepts

**About this chapter** This chapter explains how PowerBuilder and InfoMaker connect to a Watcom SQL database and how you can use PowerBuilder or InfoMaker to access and manipulate Watcom SQL databases.

**Contents**

- ◆ "Connecting to a database" on page 52
- ◆ "Savepoints" on page 58
- ◆ "Two-phase commit" on page 59
- ◆ "Understanding the NULL value" on page 60
- ◆ "Ensuring database integrity" on page 61

### **PowerBuilder users**

For more information about connecting to the database and storing and manipulating extended table and column (catalog) information, see *Connecting to Your Database*.

## Connecting to a database

Any program that uses a database must first establish a **connection** with the database engine. A user ID and password must be specified when the connection is established. The connection is usually established transparently to users by the program they are running. The user may be prompted to enter a user ID, password, and optionally other database parameters, but the program will take care of establishing the connection by using the SQL CONNECT statement.

### Multiple connections

Multiple simultaneous connections to the database engine are supported. In this case, you must use the PowerBuilder transaction object.

## Named connections

When a connection is established with a Watcom SQL database through ODBC, the ODBC driver generates a unique **name** to identify the connection internally. Therefore, if you are using ODBC to connect to Watcom SQL databases, you do not have to be concerned with named connections. PowerBuilder and InfoMaker connect to Watcom SQL through ODBC. Some programs using the database engine may have more than one connection established with the database engine. In this case, each connection after the first must be a **named** connection. The CONNECT command is used to establish a named connection (see "CONNECT" on page 202). In particular, ISQL employs named connections.

From one program, only one of the connections is active at any time. Any commands to the database engine are associated with the active connection. The SET CONNECTION command is used to switch the active connection.

Applications can use a second connection for automatically generating primary key values without inhibiting concurrency (see "Primary key generation" on page 110).

## Connection parameters

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

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

A connection parameter string may look like the following:

```
sqlca.DbParm="Connectstring='DSN=PowerBuilderDemoDb;UID=dba;-
PWD=sql' "
```

The minimum attributes required to connect to Watcom SQL are the dbms and the DbParm. The following example uses the default transaction object, which is sqlca.

```
sqlca.dbms="odbc"
sqlca.DbParm
="Connectstring='DSN=PowerBuilderDemoDb;UID=dba;PWD=sql' "
```

DBBACKUP, DBUNLOAD, and DBVALID will take the following steps for connecting to the database. These parameter settings are used as default values for unspecified parameters to connect:

- ◆ 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 network requestor.
- ◆ 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:\WSQL40\SAMPLE.DB, then look for a local engine named sample.
- ◆ If no matching local engine is found and the network requestor (DBCLIENT) is not running, start a database engine or network requestor using the Start parameter (or the default start command). The StayConnected attribute of the transaction object will determine if the engine automatically stops when the last database is shut down or if the network requestor automatically stops when its last connection is gone.
- ◆ 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.

For example, consider the following parameters:

```
DBF=c:\pb4\pbdemodb.db;UID=dba;PWD=sql
```

- 1 If there is a local engine with the name PSDemoDB, then you will connect to it.
- 2 If the client is running and there is a server with the name PSDemoDB, then you will connect to it.
- 3 Otherwise, a local engine will be started on the file C:\PB4\PSDEMODB.DB. If the database exists, and the engine starts without errors, then you will be connected to that engine.

The **Userid** and **Password** specify the authorization information to the database engine. The **ConnectionName** is optional and allows you to name



the connection established with the database engine. The ODBC interface will prompt for missing information that is required.

The **Start** parameter allows you to specify a command line for starting the database engine or network requestor (DBCLIENT). The default start command is DBSTART -q ( DBSTARTW -q for Windows). Use DB32W (under NT), or DB32W (under Windows) to take advantage of the 32-bit engine. Before Watcom SQL Version 4.0, the start string required a %d somewhere in the command as a placeholder for the database name. This is no longer necessary because the information is passed as **EngineName** or as **DatabaseFile**.

## Transactions

SQL statements (commands) are grouped into **transactions**. Each transaction is a **logical unit of work**, meaning that the commands within a transaction should be somewhat related. In addition, the database engine guarantees that either each transaction is completed in its entirety, or not at all.

### InfoMaker

InfoMaker automatically takes control of the transaction so you do not have to.

A transaction begins when a connection is established. The transaction ends with a COMMIT or a ROLLBACK. A new transaction starts with the next SQL statement.

The COMMIT statement ends a transaction and commits the changes to the database. If the command is successful, the transaction changes are guaranteed to be in the database. The ROLLBACK statement aborts all changes made during the current transaction (since the last COMMIT statement). Additionally, on any power failure or abnormal termination of the database software, a ROLLBACK is performed automatically.

### Example

Suppose a customer becomes inactive and all records for that customer are to be removed from the database. SQL commands are required to remove rows related to the customer from the customer table and all other tables. The database would be in an inconsistent state if only the row from the customer table was deleted, since other information related to the deleted customer in

other tables would remain. Thus, DELETE commands are used to remove all the customer information from a logical unit of work.

For a complete description of consistency checking in the database, see "Ensuring database integrity" on page 61.

**AutoCommit** Use care when you change the setting of the boolean attribute AutoCommit in the transaction object. The setting of AutoCommit determines whether normal recoverable transaction handling takes place. If AutoCommit is TRUE, ROLLBACK cannot be issued. AutoCommit is set to be TRUE by a statement of the form:

```
sqlca.autocommit = true
```

When AutoCommit is FALSE (the default), normal transaction processing takes place: a BEGIN TRANSACTION is internally issued on a successful connect and this transaction is terminated by a COMMIT TRANSACTION or ROLLBACK TRANSACTION. AutoCommit is set to be FALSE by a statement of the form:

```
sqlca.autocommit = false
```

## Connecting during development

In the development environment, a database connection is established whenever you require access to the database—for example, when you:

- ◆ Invoke a painter that requires access to the database (for example, the Database painter or the Query painter)
- ◆ Save a form or query that accesses the database

A user ID and password must be available to PowerBuilder when the connection is established.

## Closing the connection

The database connection is opened the first time you need to connect to the database and closed when you close the painter that needed the connection.

**In PowerBuilder** The setting of the Database variable StayConnected controls when PowerBuilder closes the connection. The default is to close the connection when you close the painter.

## Connections during execution

When a user is running an application, the database connection is usually transparent to the user. The user may be prompted to enter a user ID, password, and optionally a database name, but the program takes care of establishing the connection by populating the transaction object with the correct attributes and then issuing a connect, using sqlca.

## Savepoints

Within a transaction, Watcom SQL supports **savepoints**. Before Watcom SQL Version 4.0, savepoints were referred to as **subtransactions**. Savepoints only apply when Autocommit is set to false. A `SAVEPOINT` command defines a point in a transaction where all changes after the point can be undone by a `ROLLBACK TO SAVEPOINT` command. Once a `RELEASE SAVEPOINT` command has been executed, the savepoint can no longer be used.

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.

## Two-phase commit

Two-phase commit is a mechanism to coordinate 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` command. The second phase actually performs the commit operation via the `COMMIT` command.

If you want to coordinate transactions with multiple servers, you can issue the `PREPARE TO COMMIT` command 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).

## Understanding the NULL value

The NULL value is a special value which is different from any valid value for any data type. However, the NULL value is a legal value in any data type. The NULL value is used to represent missing or inapplicable information. Note that there are two separate and distinct cases where NULL is used:

- ◆ **missing** There may be a valid value for this field, but that value is unknown.
- ◆ **inapplicable** The field does not apply for this particular row.

Many common errors in formulating SQL queries are caused by the behavior of NULL. See "NULL value" on page 263 for a complete description of the behavior of NULL. See also the "Empty String is NULL" attribute of an Edit Style in the DataWindow painter, in the PowerBuilder or InfoMaker *User's Guide*.

## Ensuring database integrity

Watcom SQL supports **entity integrity** and **referential integrity**.

### Entity integrity

**Entity integrity** ensures that every row of a given table can be uniquely identified by a **primary key** that does not contain any NULL values.

#### Example

A table of employees might use an employee ID as a primary key. This would imply that employee IDs cannot be the NULL value, and each row in the employee table has a unique employee ID.

Although only the employee ID column is part of the primary key in this example, a primary key can consist of more than one column.

### Creating primary keys

When you use the Database painter to create or alter a table, you can create a primary key. PowerBuilder or InfoMaker will generate the Watcom SQL CREATE TABLE or ALTER TABLE statement syntax to create the key and submit it to the Watcom SQL database.

#### Using primary keys

The use of primary keys is optional. However, in the PowerBuilder and InfoMaker DataWindows, you can update data only if the table has a unique index. A unique index is created automatically in Watcom SQL databases when you create the primary key.

**Primary key searches** Watcom SQL has built-in facilities to make searches on primary keys go quickly. User-defined indexes on the primary key are not required and not recommended.

For more information about creating and maintaining primary keys in PowerBuilder or InfoMaker, see the PowerBuilder or InfoMaker *User's Guide*.

## Referential integrity

Two separate tables in a database are often related in some way. Watcom SQL supports **referential integrity**, which 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.

### Example 1

Assume you have a table of employees and a table of departments. The table of employees is related to the department table, because each employee must be assigned to a department in the department table. Each row in the department table contains a department ID. In the employee table, the department ID is called a **foreign key** for the department table; each department ID in the employee table must correspond exactly to a department ID in the department table. This is a **mandatory** foreign key since it is not allowed to be NULL.

### Example 2

Assume there is also a 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. However, office locations are often not assigned when 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.

## Creating foreign keys

When you use the Database painter to create or alter a table, you can create foreign keys. PowerBuilder or InfoMaker will generate the Watcom SQL CREATE TABLE or ALTER TABLE statement syntax to create the key and submit it to the Watcom SQL database.

Once a foreign key has been created, Watcom SQL ensures that the columns will contain only values that are present as primary key values in the table associated with the foreign key.

### Searches and joins

Watcom SQL has built-in facilities to make searches and joins on foreign keys efficient. User-defined indexes on the foreign key are not required and not recommended.



## Referential integrity actions

Referential integrity actions can be used by a database designer to maintain foreign key relationships in the database. Whenever a primary key value is changed or deleted from a database table, there may be corresponding foreign key values in other tables that should be modified in some way. Through the use of the CREATE TABLE or ALTER TABLE statements, you can define the actions taken in these situations.

### Example

A foreign key in the Employee table might be defined with the following clause:

```
FOREIGN KEY REFERENCES Department
ON UPDATE CASCADE
ON DELETE CASCADE
```

The ON UPDATE CASCADE tells the database engine that, whenever it updates a primary key value in the Department table, it should automatically update any corresponding foreign keys in the Employee table. ON UPDATE CASCADE must be entered through the Database Administrator.

The ON DELETE CASCADE tells the database engine that whenever it deletes a row from the Department table, it should automatically delete any rows in the Employee table whose foreign keys correspond to the primary key value in the row deleted from the Department table. You can specify the action on DELETE through the foreign key definition in the Database painter.

You can specify either an ON UPDATE clause, an ON DELETE clause, or both, followed by one of the following actions:

**CASCADE** When used with ON UPDATE, update the corresponding foreign keys to match the new primary key value. When used with ON DELETE, deletes the rows from the table that match the deleted primary key.

**SET NULL** Sets to NULL all the foreign key values that correspond to the updated or deleted primary key.

**SET DEFAULT** Sets to the value specified by the column(s) DEFAULT clause, all the foreign key values that match the updated or deleted primary key value. This is set by the "Initial" dropdown.

**RESTRICT** Generates an error if an attempt is made to update or delete a primary key value while there are corresponding foreign keys elsewhere in

the database. This was the only form of referential integrity prior to Watcom SQL Version 4.0 and is the default action if no action is specified.

If an error occurs during the processing of a referential action, the statement that caused the trigger will fail.

**Referential action permissions**

A referential action executes with the permissions of the creator of the foreign table. A referential action can update or delete rows from a table that a user could not update or delete directly.

Watcom SQL also supports triggers (see "Triggers" on page 158), which allow you to define actions when rows are inserted, updated, or deleted.

## CHAPTER 7

# Data Types, Functions, Expressions, and Conditions

**About this chapter** This chapter describes the data types, functions, and expressions you can use with Watcom SQL databases. It also describes the conditional statements you can use in functions and expressions.

You can use expressions in commands including the SELECT command. The SELECT command is described in "SELECT Command Syntax" on page 97. For information about other Watcom SQL commands, see "Command Syntax" on page 181.

**Contents**

- ◆ "Data types" on page 66
- ◆ "Functions" on page 71
- ◆ "Expressions" on page 84
- ◆ "Conditions" on page 90

### **PowerBuilder users**

In PowerBuilder, you can also use expressions in commands in embedded SQL and in the Database Administration painter (DBA notepad). For more information about embedded SQL, see *PowerBuilder PowerScript Language*; for more information about the Database Administration painter, see the *PowerBuilder User's Guide*.

## Data types

### Syntax

data-type:

```
| BINARY [ ( max-length ) ] |
| CHAR [ ( max-length ) ] |
| CHARACTER [ ( max-length ) ] |
| CHARACTER VARYING [ ( max-length ) ] |
| DATE |
| DECIMAL [ ( precision [,scale] ) ] |
| DOUBLE |
| FLOAT |
| INT |
| INTEGER |
| LONG BINARY |
| LONG VARCHAR |
| NUMERIC [ ( precision [,scale] ) ] |
| REAL |
| SMALLINT |
| TIME |
| TIMESTAMP |
| VARCHAR [ ( max-length ) ] |
```

### Purpose

To specify a data type.

### Usage

PowerScript embedded SQL.

#### **Data type display**

When you are connected to a Watcom SQL database, the Watcom SQL data types display in the data type lists in the Database painter and the Select painter.

### Authorization

Must be connected to the database.

### Side effects

None.

### See also

CREATE TABLE, ALTER TABLE, Compound Statements, Expressions.

### Description

All information is stored in one of these data types:

#### **CHAR [(size)]**

Character data of maximum length **size**. If **size** is omitted, the default is 1.

The maximum size allowed is 32,767. See "Character data" and "Long strings" below.

**CHARACTER [(size)]**

Same as CHAR[(size)].

**VARCHAR [(size)]**

Same as CHAR[(size)].

**CHARACTER VARYING[(size)]**

Same as CHAR[(size)].

**LONG VARCHAR**

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

**BINARY [(size)]**

Binary data of maximum length **size** (in bytes). If **size** is omitted, the default is 1. The maximum size allowed is 32,767. The **BINARY** data type is identical to the **CHAR** data type except when used in comparisons.

**BINARY** values will be compared exactly while **CHAR** values are compared without respect to upper/lower case (depending on the case-sensitivity of the database) or accented characters.

**LONG BINARY**

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

**INT**

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

**INTEGER**

Same as **INT**.

**SMALLINT**

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

**DECIMAL [(precision[,scale])]**

A decimal number with **precision** total digits and with **scale** of the digits after the decimal point. The defaults are **scale = 6** and **precision = 30**. You can change the defaults in the Database painter.

For more information about the Database painter, see the PowerBuilder *User's Guide*.

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

$$2 + \text{int}( (\text{before}+1) / 2 ) + \text{int}( (\text{after}+1)/2 )$$

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

### **NUMERIC [(precision[,scale])]**

Same as DECIMAL.

### **FLOAT**

A double precision floating-point number stored in 8 bytes. The range of values is 2.22507385850720160e-308 to 1.79769313486231560e+308.

### **DOUBLE**

Same as FLOAT.

### **REAL**

A single precision floating-point number stored in 4 bytes. The range of values is 1.175494351e-38 to 3.402823466e+38.

### **DATE**

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

### **TIMESTAMP**

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

### **TIME**

A time of day, containing hour, minute, second and fraction of a second. The fraction is stored to 6 decimal places. A **TIME** value requires 8 bytes of storage. (ODBC standards restrict **TIME** data type to an accuracy of seconds. **Do not** use Time data types in **WHERE** clause comparisons.)

## Character data

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

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

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

This problem also shows up if a database is used across platforms. PowerBuilder and many other Windows applications insert data into the database in the standard ANSI character set. If non-Windows applications attempt to use this data, they will not properly display or update the extended characters. This problem is quite complex. If any of your applications use the extended characters in the upper half of the code page, make sure that all clients and all machines using the database use the same or a compatible code page.

## Long strings

Watcom SQL treats CHAR, VARCHAR, and LONG VARCHAR columns all as the same type. Values up to 254 characters are stored as short strings, which are stored with a preceding length byte. Any values that are longer than 255 bytes are considered long strings. Characters after the 255th are stored separate from the row containing the long string value.

There are several functions (see "Functions" on page 71) that will ignore the part of any string past the 255th character. They are soundex, similar, and all of the date functions. Also, any arithmetic involving the conversion of a long string to a number will work on only the first 255 characters. It would be extremely unusual to run in to one of these limitations.

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

### Dates and times

Date, time and timestamp constants are represented as strings. They are fetched from the database engine as a string, and they are sent to the database engine as a string. Internally, they are stored as numbers. When a string is compared to a date, the string will automatically be converted to a date (see "Date format" on page 88). If you wish to compare a date to a string *as a string*, you must use the `dateformat` function or `CAST` operator to convert the date.

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

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

If a database column requires only a date, the application should make sure that the times are not specified. This way, comparisons with date-only strings will work as expected.



# Functions

## Syntax

normal-function:

```
| ABS ( numeric-expr )  
| ACOS ( numeric-expr )  
| ARGN ( integer-expr, expression [, ...] )  
| ASCII ( string-expr )  
| ASIN ( numeric-expr )  
| ATAN ( numeric-expr )  
| CEILING ( numeric-expr )  
| CHAR ( string-expr )  
| COALESCE ( expression, expression [, ...] )  
| COS ( numeric-expr )  
| COT ( numeric-expr )  
| DATE ( expression )  
| DATEFORMAT ( datetime-expr, string-expr )  
| DATETIME ( expression )  
| DAY ( date-expr )  
| DAYS ( date-expr )  
| DAYS ( date-expr, date-expr )  
| DAYS ( date-expr, integer-expr )  
| DOW ( date-expr )  
| EXP ( numeric-expr )  
| FLOOR ( numeric-expr )  
| HOUR ( datetime-expr )  
| HOURS ( datetime-expr )  
| HOURS ( datetime-expr, datetime-expr )  
| HOURS ( datetime-expr, integer-expr )  
| IFNULL ( expression, expression [, expression] )  
| ISNULL ( expression, expression [, ...] )  
| LCASE ( string-expr )  
| LEFT ( string-expr, numeric-expr )  
| LENGTH ( string-expr )  
| LOCATE ( string-expr, string-expr [, numeric-expr ] )  
| LOG ( numeric-expr )  
| LOG10 ( numeric-expr )  
| LTRIM ( string-expr )  
| MINUTE ( datetime-expr )  
| MINUTES ( datetime-expr )  
| MINUTES ( datetime-expr, datetime-expr )  
| MINUTES ( datetime-expr, integer-expr )  
| MOD ( numeric-expr, numeric-expr )  
| MONTH ( date-expr )  
| MONTHS ( date-expr )  
| MONTHS ( date-expr, date-expr )  
| MONTHS ( date-expr, integer-expr )
```

```
| NOW ( * )
| NUMBER ( * )
| PI ( * )
| PLAN ( string-expr )
| RCASE ( string-expr )
| REMAINDER ( numeric-expr, numeric-expr )
| REPEAT ( string-expr, numeric-expr )
| RIGHT ( string-expr, numeric-expr )
| RTRIM ( string-expr )
| SECOND ( expression )
| SECONDS ( datetime-expr )
| SECONDS ( datetime-expr, datetime-expr )
| SECONDS ( datetime-expr, integer-expr )
| SIGN ( numeric-expr )
| SIMILAR ( string-expr, string-expr )
| SIN ( numeric-expr )
| SOUNDEX ( string-expr )
| SQRT ( numeric-expr )
| STRING ( string-expr [, ...] )
| SUBSTR ( string-expr, integer-expr [, integer-expr] )
| TAN ( numeric-expr )
| TODATE ( expression )
| TODAY ( * )
| TRACEBACK ( * )
| TRIM ( string-expr )
| WEEKS ( date-expr )
| WEEKS ( date-expr, date-expr )
| WEEKS ( date-expr, integer-expr )
| YEAR ( date-expr )
| YEARS ( date-expr )
| YEARS ( date-expr, date-expr )
| YEARS ( date-expr, integer-expr )
| YMD ( integer-expr, integer-expr, integer-expr )
```

## Syntax

aggregate-function:

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

where aggregate-parm is one of:  
DISTINCT column-name  
or expression

<b>Purpose</b>	To use a built-in function.
<b>Usage</b>	Embedded SQL, DataWindow, and Database painter expressions.
<b>Authorization</b>	Must be connected to the database.
<b>Side effects</b>	None.
<b>See also</b>	Expressions.
<b>Description</b>	<p>Watcom SQL has two types of functions. <b>Normal functions</b> take parameters and return results based on those parameters. <b>Aggregate functions</b> summarize data over a group of rows from the database. The groups are formed using the GROUP BY clause of the SELECT statement. Aggregate functions are only allowed in the select list and in the HAVING and ORDER BY clauses of a SELECT statement.</p> <p>For many of the functions, the types for function parameters have been specified. If you provide data of some other type, Watcom SQL will convert the data automatically. Alternative, you can use the CAST operator to do explicit conversions.</p>

**NULL value**

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

**Aggregate functions**

**COUNT( \* )**

Returns the number of rows in each group.

**COUNT( expression )**

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

**COUNT( DISTINCT column-name )**

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

**LIST( string-expr )**

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

**LIST( DISTINCT column-name )**

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

**AVG( numeric-expr )**

Computes the average of **numeric-expr** for each group of rows. This average does not include rows where the **expression** is the NULL value. Returns the NULL value for a group containing no rows.

**AVG( DISTINCT column-name )**

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

**MAX( expression )**

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

**MAX( DISTINCT column-name )**

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

**MIN( expression )**

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

**MIN( DISTINCT column-name )**

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

**SUM( expression )**

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

**SUM( DISTINCT column-name )**

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

**Normal functions**

Normal functions have been broken up into the following subcategories:

- ◆ Miscellaneous functions
- ◆ Numeric functions
- ◆ String functions
- ◆ Date and time arithmetic functions
- ◆ Other date and time functions

**Miscellaneous**

**ARGN( integer-expr, expression [, ...] )**

Using the value of **integer-expr** as **n**, return the **n**'th argument (starting at 1) from the remaining list of arguments.

**COALESCE( expression, expression [ ... , expression] )**

Returns the value of the first expression that is not NULL.

**IFNULL( expression1, expression2 [, expression3] )**

If the first expression is the NULL value, then the second expression is returned. Otherwise, the value of the third expression is returned if it was specified. If there was no third expression and the first expression is not NULL then the NULL value is returned.

**ISNULL( expression, expression [ ... , expression] )**

Same as the COALESCE function.

**NUMBER( \* )**

Generates numbers starting at 1 for each successive row in the results of the query. This is extremely useful for generating primary keys when using the INSERT from SELECT command (see "INSERT" on page 259).

In Embedded SQL, care should be exercised when seeking a cursor that references a query containing a NUMBER(\*) function. In particular, this function will return negative numbers when a database cursor is positioned relative to the end of the cursor (an absolute seek with a negative offset).

## Numeric

### **ABS( numeric-expr )**

Compute the absolute value of **numeric-expr**.

### **ACOS( numeric-expr )**

Compute the arc-cosine of **numeric-expr** in radians.

### **ASIN( numeric-expr )**

Compute the arc-sine of **numeric-expr** in radians.

### **ATAN( numeric-expr )**

Compute the arc-tangent of **numeric-expr** in radians.

### **CEILING( numeric-expr )**

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

### **COS( numeric-expr )**

Compute the cosine of **numeric-expr**, expressed in radians.

### **COT( numeric-expr )**

Compute the cotangent of **numeric-expr**, expressed in radians.

### **EXP( numeric-expr )**

Compute the exponential function of **numeric-expr**.

### **FLOOR( numeric-expr )**

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

### **LOG( numeric-expr )**

Compute the logarithm of **numeric-expr**.

### **LOG10( numeric-expr )**

Compute the logarithm base 10 of **numeric-expr**.

### **MOD( dividend, divisor )**

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

### **PI( \* )**

Return the numeric value PI.

**REMAINDER( dividend, divisor )**

Same as the MOD function.

**SIGN( numeric-expr )**

Return the sign of **numeric-expr**.

**SIN( numeric-expr )**

Compute the sine of **numeric-expr**, expressed in radians.

**SQRT( numeric-expr )**

Compute the square root of **numeric-expr**.

**TAN( numeric-expr )**

Compute the tangent of **numeric-expr**, expressed in radians.

**String**

**ASCII( string-expr )**

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

**CHAR( numeric-expr )**

Returns the character with the ASCII value **numeric-expr**.

**LCASE( string-expr )**

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

**LEFT( string-expr, numeric-expr )**

Returns the leftmost **numeric-expr** characters of **string-expr**.

**LENGTH( string-expr )**

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

**LOCATE( string-expr1, string-expr2 [, numeric-expr ] )**

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

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

**LTRIM( string-expr )**

Returns **string-expr** with leading blanks removed.

**PLAN( string-expr )**

Returns the optimization strategy of the SELECT statement **string-expr**.

**RIGHT( string-expr, numeric-expr )**

Returns the rightmost **numeric-expr** characters of **string-expr**.

**RTRIM( string-expr )**

Returns **string-expr** with trailing blanks removed.

**SIMILAR( string-expr1, string-expr2 )**

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

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

**SOUNDEX( string-expr )**

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

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

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

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

is based on the letters A, P, L and S.

**STRING( string1, [ string2, ..., string99 ] )**

Concatenates the strings into one large string. NULL values are treated as empty strings (''). Any numeric or date parameters are automatically converted to strings before concatenation. Note that this function can be used to force any single expression to be a string by supplying that expression as the only parameter.



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

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

**TRACEBACK( \* )**

Returns a traceback of the procedures and triggers that were executing when the most recent exception (error) occurred. This is useful for debugging procedures and triggers.

**TRIM( string-expr )**

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

**UCASE( string-expr )**

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

**Date arithmetic**

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

**YEAR( date-expr )**

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

**YEARS( date-expr )**

Same as the YEAR function.

**YEARS( date-expr, date-expr )**

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

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

**YEARS( date-expr, integer-expr )**

Add **integer-expr** years to the given date. If the new date is past the end of the month (such as YEARS( '1992-02-29', 1 ) ) the result is set to the last

day of the month. If the **integer-expr** is negative, the appropriate number of years are subtracted from the date. Hours, minutes, and seconds are ignored.

### **MONTH( date-expr )**

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

### **MONTHS( datetime-expr )**

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

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

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

### **MONTHS( date-expr, date-expr )**

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

### **MONTHS( date-expr, integer-expr )**

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

### **WEEKS( datetime-expr )**

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

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

### **WEEKS( date-expr, date-expr )**

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

### **WEEKS( date-expr, integer-expr )**

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

**DAY( date-expr )**

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

**DAYS( datetime-expr )**

Return the number of days since an arbitrary starting date.

**DAYS( date-expr, date-expr )**

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

**DAYS( date-expr, integer-expr )**

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

**DOW( date-expr )**

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

**HOUR( datetime-expr )**

Returns a number from 0 to 23 corresponding to the hour component of the given date.

**HOURS( datetime-expr )**

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

**HOURS( datetime-expr, datetime-expr )**

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

**HOURS( datetime-expr, integer-expr )**

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

**MINUTE( datetime-expr )**

Returns a number from 0 to 59 corresponding to the minute component of the given date/time.

**MINUTES( datetime-expr )**

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

**MINUTES( datetime-expr, datetime-expr )**

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

**MINUTES( datetime-expr, integer-expr )**

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

**SECOND( datetime-expr )**

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

**SECONDS( datetime-expr )**

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

**SECONDS( datetime-expr, datetime-expr )**

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

**SECONDS( datetime-expr, integer-expr )**

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

**Other date and time**

**DATE( expression )**

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

**DATEFORMAT( date-expr, string-expr )**

Returns a string representing the date **date-expr** in the format specified by **string-expr**. Any allowable date format can be used for **string-expr** (for more information, see the discussion of embedded SQL in ODBC in PowerBuilder online Help). For example,

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

is

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

**DATETIME( expression )**

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

**NOW( \* )**

Returns the current date and time. This is the historical syntax for CURRENT\_TIMESTAMP.

**TODAY( \* )**

Returns today's date. This is the historical syntax for CURRENT\_DATE.

**YMD( year-num, month-num, day-num )**

Returns a date value corresponding to the given year, month, and day of the month. If the month is outside the range 1-12, the year is adjusted accordingly. Similarly, the day is allowed to be any integer: the date is adjusted accordingly. For example,

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

# Expressions

## Syntax

expression:

- | constant
- | [correlation-name .] column-name
- | variable-name
- | function-name ( expression, ... )
- | - expression
- | expression + expression
- | expression - expression
- | expression \* expression
- | expression / expression
- | expression + expression
- | expression || expression
- | ( expression )
- | ( subquery )
- | CAST ( expression AS data-type )
- | if-expression

constant:

- | integer
- | number
- | 'string'
- | special-constant
- | host-variable

special-constant:

- | CURRENT DATE
- | CURRENT TIME
- | CURRENT TIMESTAMP
- | NULL
- | SQLCODE
- | SQLSTATE
- | USER

if-expression:

IF condition THEN expression [ ELSE expression ] ENDIF

## Purpose

To specify an arithmetic, string or date/time expression.

<b>Usage</b>	In the SELECT statement in PowerBuilder and InfoMaker, and in embedded SQL in PowerBuilder.
<b>Authorization</b>	Must be connected to the database.
<b>Side effects</b>	None.
<b>See also</b>	Conditions, Data Types.
<b>Description</b>	Expressions are formed using constants, column names, functions, subqueries, and operators.
<b>Constants</b>	<p>Constants are numbers or strings. String constants are enclosed in apostrophes ('single quotes'). An apostrophe is represented inside the string by two apostrophes in a row.</p> <p>There are several special constants:</p> <p><b>CURRENT DATE</b> The current year, month and day represented in the DATE data type.</p> <p><b>CURRENT TIME</b> The current hour, minute, second and fraction of a second represented in the TIME data type. Although the fraction of a second is stored to 6 decimal places, the current time is limited by the accuracy of the system clock. The clock is only accurate to approximately 1/18th of a second rounded to two decimal places.</p> <p><b>CURRENT TIMESTAMP</b> Combines CURRENT DATE and CURRENT TIME to form a TIMESTAMP value containing year, month, day, hour, minute, second and fraction of a second. Like CURRENT TIME, the accuracy of the fraction of a second is limited by the system clock. In Embedded SQL, a host variable can also be used in an expression wherever a constant is allowed.</p> <p><b>NULL</b> The NULL value (see "NULL value" on page 263).</p> <p><b>SQLCODE</b> Current SQLCODE value (see "Database Error Messages" on page 357).</p>

### **SQLSTATE**

Current SQLSTATE value (see "Database Error Messages" on page 357).

### **USER**

A string containing the user ID of the current connection.

In Embedded SQL, a host variable can also be used in an expression wherever a constant is allowed.

### **Column names**

A column name is an identifier preceded by an optional correlation name. (A correlation name is usually a table name. See "FROM" on page 243 for more information on correlation names.) If a column name has characters other than letters, digits and underscore, it must be surrounded by quotation marks (""). For example, the following are valid column names:

```
employee.name  
address  
"date hired"  
"salary"."date paid"
```

### **Variable names**

A variable name is as an identifier corresponding to a variable created with the CREATE VARIABLE command.

### **Functions**

See "Functions" on page 71 for a description of the functions available in Watcom SQL.

### **Subqueries**

A subquery is a SELECT statement enclosed in parentheses. The SELECT statement must contain one and only one select list item. Usually, the subquery is allowed to return only one row. See "Conditions" on page 90 for other uses of subqueries. A subquery can be used anywhere that a column name can be used. For example, a subquery can be used in the select list of another SELECT statement.

### **Operators**

The normal precedence of operations apply. Expressions in parentheses are evaluated first; then multiplication and division before addition and subtraction. String concatenation happens after addition and subtraction.

#### **expression + expression**

Addition. If either expression is the NULL value, the result is the NULL value.



**expression - expression**

Subtraction. If either expression is the NULL value, the result is the NULL value.

**- expression**

Negation. If the expression is the NULL value, the result is the NULL value.

**expression \* expression**

Multiplication. If either expression is the NULL value, the result is the NULL value.

**expression / expression**

Division. If either expression is the NULL value or if the second expression is 0, the result is the NULL value.

**expression || expression**

String concatenation (two vertical bars). If either string is the NULL value, it is treated as the empty string for concatenation.

**( expression )**

Parentheses.

**IF condition THEN expression1 [ELSE expression2] ENDIF**

Evaluates to the value of **expression1** if the specified search condition is TRUE, the value of **expression2** if **condition** is FALSE, and the NULL value if **condition** is UNKNOWN. (See "NULL value" on page 263 and "Conditions" on page 90 for more information about TRUE, FALSE and UNKNOWN conditions.)

**Type conversions**

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

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

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

All date constants are specified as strings. The string will automatically be converted to a date before use. See "Data types" on page 66.

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

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

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

CAST can be used to force type conversions. The cast operation

```
CAST( expression AS data-type )
```

forces a conversion of the expression to the named data type. If the length is omitted for character string types, Watcom SQL chooses an appropriate length. Similarly, if neither precision nor scale is specified for a DECIMAL conversion, Watcom SQL selects appropriate values. Examples:

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

The following functions can also be used to force type conversions (see "Functions" on page 71).

- date( value )**      Converts the expression into a date, and removes any hours, minutes or seconds. Conversion errors may be reported.
- string( value )**     Similar to CAST( value AS CHAR ), except that string( NULL ) is the empty string ( ''), while CAST( NULL AS CHAR ) is the NULL value.
- value+0.0**            Equivalent to CAST( value AS DECIMAL ).

### Date format

When converting a string to a date, time or timestamp, Watcom SQL allows many date formats. The following are all valid dates (depending on the DATE\_ORDER):

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

When a string is converted to a date, parts of the date might not be in the string. The following defaults are used:

**year**

This year

**month**

No default

**day**

1 (useful for month fields; for example, 'May 1992' will be the date '1992-05-01 00:00' )

**hour, minute, second, fraction**

0

For information about specifying the date format, see the discussion of escape clauses in *PowerScript Language*. There are many functions dealing with dates and times (see "Functions" on page 71). In addition to all of the date and time functions, the following arithmetic operators are allowed on dates:

**timestamp + integer**

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

**timestamp - integer**

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

**date - date**

Compute the number of days between two dates or timestamps.

**date + time**

Create a timestamp combining the given date and time.

# Conditions

## Syntax

condition:

```
| expression compare expression  
| expression compare ANY ( subquery )  
| expression compare ALL ( subquery )  
| expression IS [NOT] NULL  
| expression [NOT] LIKE expression [ESCAPE expression]  
| expression [NOT] BETWEEN expression AND expression  
| expression [NOT] IN ( expression, ... )  
| expression [NOT] IN ( subquery )  
| EXISTS ( subquery )  
| NOT condition  
| condition AND condition  
| condition OR condition  
| ( condition )  
| ( condition , estimate )  
| condition IS [NOT] TRUE  
| condition IS [NOT] FALSE  
| condition IS [NOT] UNKNOWN
```

compare:                    one of = > < >= <= <> != ~=

## Purpose

To specify a search condition for a WHERE clause, a HAVING clause, a CHECK clause, a JOIN clause or an IF expression.

## Usage

DBA notepad.

## Authorization

Must be connected to the database.

## Side effects

None.

## See also

Expressions.

## Description

Conditions are used as search conditions to choose a subset of the rows from a table, or in an IF-THEN-ELSE-ENDIF. The simplest form of a condition is a comparison of two values:

=  
equal to

<  
less than

<=

less than or equal to

>

greater than

>=

greater than or equal to

~=

not equal to

<>

not equal to (equivalent to ~=)

!=

not equal to (equivalent to ~=)

**Case**

All string comparisons are *case insensitive* unless the database was created as case sensitive.

In SQL, every condition can either be TRUE, FALSE or UNKNOWN. The result of a comparison is UNKNOWN if either value being compared is the NULL value. Rows satisfy a search condition if and only if the result of the condition is TRUE. See "NULL value" on page 263.

SQL provides several other operators for use in conditions. In the following descriptions, the optional keyword NOT reverses the meaning of the condition.

**expression IS [NOT] NULL**

TRUE if the expression is the NULL value, FALSE otherwise.

**expr [NOT] BETWEEN start-expr AND end-expr**

TRUE if **expr** is between **start-expr** and **end-expr**. This is equivalent to:

`expr >= start-expr AND expr <= end-expr`

Note that BETWEEN can result in TRUE, FALSE or UNKNOWN.

**expression [NOT] IN ( value-expr1 [, value-expr2 ] ... )**

TRUE if **expression** equals any of the listed values, UNKNOWN if **expression** is the NULL value and FALSE otherwise.

**expression [NOT] LIKE pattern [ESCAPE escape-expr]**

TRUE if **expression** matches the **pattern**. The pattern may contain any number of special characters. The special characters are:

- ◆ **\_ (underscore)** " Matches any character
- ◆ **% (percent)** Matches any string of characters including the empty string

All other characters must match exactly. If either **expression** or **pattern** is the NULL value, this condition is UNKNOWN. For example, the search condition

```
name LIKE 'a%b_'
```

will be TRUE for any row where the **name** starts with the letter a and has the letter b as its second last character.

If an **escape-expr** is specified, it must evaluate to a single character. The character can precede a percent or an underscore in the **pattern** to prevent the special character from having its special meaning. A percent will match a percent, and an underscore will match an underscore.

Subqueries that return one column and zero or one row can be used in any SQL statement anywhere that a column name could be used, including in the middle of an expression. If a subquery returns no rows, its value is considered to be NULL.

The following are additional conditions that involve subqueries. The result of each subquery must contain one column.

**expression [NOT] IN ( subquery )**

TRUE if **expression** equals any of the values in the result of the subquery, or FALSE if the subquery result does not contain any rows. This condition is UNKNOWN if **expression** is the NULL value unless the result of the subquery has no rows.

**EXISTS (subquery)**

TRUE if the subquery result contains at least one row, and FALSE if the subquery result does not contain any rows.

**expression > ANY (subquery)**

TRUE if **expression** is greater than any of the values in the result of the

subquery, or FALSE if the subquery result does not contain any rows. Any comparison operator can be used in place of >. Note that "= ANY" is equivalent to IN. This condition is UNKNOWN if **expression** is the NULL value unless the result of the subquery has no rows, in which case the condition is always FALSE.

**expression > SOME (subquery)**

Same as ANY.

**expression > ALL (subquery)**

TRUE if **expression** is greater than all of the values in the result of the subquery or if the result of the subquery does not contain any rows. Any comparison operator can be used in place of >. Note that "~= ALL" is equivalent to NOT IN. This condition is UNKNOWN if **expression** is the NULL value unless the result of the subquery has no rows, in which case the condition is always TRUE.

**expression > (subquery)**

TRUE if **expression** is greater than the result of the subquery, FALSE otherwise. The result of the subquery must contain zero or one row. Any comparison operator can be used in place of >. This condition is UNKNOWN if the expression is the NULL value, if the subquery result has no rows, or if the subquery result is the NULL value.

Search conditions can also be used for joining rows from different tables.

```
Employee.emp_lname = 'Johnson' AND Employee.emp_id =  
department.emp_id
```

This will result in combined rows from the Employee and Department tables where the employee name (emp\_lname) is Johnson and the information from the Department table is selected such that the emp\_ID column matches the emp\_ID column in the Employee table. See "FROM" on page 243 for a full description of joins.

**Logical operators**

Search conditions can be combined using AND, OR and NOT.

**condition1 AND condition2**

TRUE if both conditions are TRUE, FALSE if either condition is FALSE, and UNKNOWN otherwise.

**condition1 OR condition2**

TRUE if either condition is TRUE, FALSE if both conditions are FALSE, and UNKNOWN otherwise.

**NOT condition**

TRUE if the condition is FALSE, FALSE if condition is TRUE and UNKNOWN if the condition is UNKNOWN.

**condition IS [NOT] truth-value**

TRUE if the **condition** evaluates to the **truth-value** (TRUE, FALSE, or UNKNOWN). Otherwise, the value is FALSE.

SQL uses what is known as a three-valued logic. The following figure shows how the logical operators of SQL work in three-valued logic.

<b>AND</b>	<b>TRUE</b>	<b>FALSE</b>	<b>UNKNOWN</b>
TRUE	TRUE	FALSE	UNKNOWN
FALSE	FALSE	FALSE	FALSE
UNKNOWN	UNKNOWN	FALSE	UNKNOWN

<b>OR</b>	<b>TRUE</b>	<b>FALSE</b>	<b>UNKNOWN</b>
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	UNKNOWN
UNKNOWN	TRUE	UNKNOWN	UNKNOWN

<b>NOT</b>	<b>TRUE</b>	<b>FALSE</b>	<b>UNKNOWN</b>
	FALSE	TRUE	UNKNOWN

<b>IS</b>	<b>TRUE</b>	<b>FALSE</b>	<b>UNKNOWN</b>
TRUE	TRUE	FALSE	FALSE
FALSE	FALSE	TRUE	FALSE
UNKNOWN	FALSE	FALSE	TRUE



**When does a row satisfy a condition?**

Rows satisfy a condition in a WHERE or HAVING clause if the condition is TRUE. It is important to remember that rows where the condition is UNKNOWN are rejected in addition to rows where the condition is FALSE.

Example

```
employee.emp_lname > 'Smith'  
age = 20  
employee.emp_lname = 'Johnson' AND student.age > 18  
emp_lname LIKE 'Br%'  
birthdate BETWEEN '1966-1-1' AND '1966-6-30'  
Employee IN (SELECT level FROM Skill)
```



## CHAPTER 8

# SELECT Command Syntax

**About this chapter**      When you define a data source for a report, form, or a PowerBuilder DataWindow object that accesses a database, you are defining a **SELECT** statement (also called a **SELECT** command, or query).

This chapter describes the syntax of the **SELECT** statement. Understanding this statement helps you to construct powerful queries.

**Contents**

- ◆ "Building **SELECT** statements" on page 98
- ◆ "**SELECT**" on page 99

## **Building SELECT statements**

You do not need to know SQL to create a SELECT command. You paint the SELECT command by making selections in the painters. The application then uses your selections to build a SELECT command and submit it to Watcom SQL. Watcom SQL executes the command and retrieves the data you requested.

Although you do not need to know SQL to create a SELECT command in PowerBuilder or InfoMaker, you should learn about the SELECT command. If you understand the SELECT statement, you can make informed selections in the painters that take full advantage of SELECT command options and thus create a more powerful query.

This chapter describes syntax of the SELECT command.

In PowerBuilder you can create scripts that contain embedded SQL commands and you can execute SQL commands immediately in the Database Administration painter (DBA notepad). The Watcom SQL commands (other than SELECT) that you can use in embedded SQL and in the Database Administration painter are described in "Command Syntax" on page 181.

# SELECT

## Syntax

```

SELECT [ ALL | DISTINCT ] select-list
    ... [ | INTO host-variable-list          | ]
        | INTO variable-list                |
    ... FROM table-list
    ... [ WHERE search-condition ]
    ... [ GROUP BY column-name, ... ]
    ... [ HAVING search-condition ]
    ... |[ ORDER BY expression [ ASC | DESC ], ... ] |
        |[ ORDER BY integer [ ASC | DESC ], ... ]   |

select-list:
    | table-name.*                               |, ...
    | expression [ AS alias-name ]              |
    | *                                          |

```

## Purpose

To retrieve information from the database.

## Usage

PowerScript embedded SQL and in the painters.

PowerBuilder and InfoMaker generate the SELECT command automatically when you use a painter to create a data source that uses a Watcom SQL database. For more information about data sources, see the PowerBuilder or InfoMaker *User's Guide* or online Help.

The INTO clause with **host-variable-list** is used in Embedded SQL only. The INTO clause with **variable-list** is used in procedures and triggers only.

## Authorization

Must have SELECT permission on the named tables and views.

## Side effects

None.

## See also

FROM, Conditions, CREATE VIEW, UNION, Expressions, DECLARE, OPEN, FETCH.

### Description

The SELECT command is used for retrieving results from the database.

A SELECT command can also be used in procedures and triggers or in Embedded SQL. The SELECT command with an INTO clause is used for retrieving results from the database when the SELECT statement only returns one row. For multiple row queries, you must use cursors. (See DECLARE, OPEN and FETCH.) A SELECT statement can also be used to return a result set from a procedure (see "Result sets from procedures" on page 170).

The various parts of the SELECT command are described below:

#### **ALL or DISTINCT**

If neither ALL nor DISTINCT is specified, ALL rows which satisfy the clauses of the SELECT command are retrieved. If DISTINCT is specified, duplicate output rows are eliminated. This is called the **projection** of the result of the command. In many cases, commands take significantly longer to execute when DISTINCT is specified. Thus, the use of DISTINCT should be reserved for cases where it is necessary.

If DISTINCT is used, the command cannot contain an aggregate function with a DISTINCT parameter.

#### **select list**

The select list is a list of expressions separated by commas specifying what will be retrieved from the database. If asterisk (\*) is specified, it is expanded to select all columns of all tables in the FROM clause (**table-name.\*** is expanded to select all columns of the named table). Aggregate functions are allowed in the select list (see "Functions" on page 71). Subqueries are also allowed in the select list (see "Expressions" on page 84). Each subquery must be within parentheses.

**Alias-names** can be used throughout the query to represent the aliased expression.

#### **INTO host-variable-list**

This clause is used in Embedded SQL only. It specifies where the results of the SELECT statement will go. There must be one host-variable item for each item in the select list. Select list items are put into the host variables in order. An indicator host variable is also allowed with each **host-variable** so the program can tell if the select list item was NULL.

#### **INTO variable-list**

This clause is used in procedures and triggers only. It specifies where the results of the SELECT statement will go. There must be one variable for

each item in the select list. Select list items are put into the variables in order.

### **FROM table-list**

Rows are retrieved from the tables and views specified in the **table list**. Joins can be specified using join operators. For a full description see "FROM" on page 243.

### **WHERE search-condition**

The **search-condition** restricts the rows that will be selected from the tables named in the FROM clause. It is also used to do joins between multiple tables. This is accomplished by putting a condition in the WHERE clause that relates a column or group of columns from one table with a column or group of columns from another table. Both tables must be listed in the FROM clause.

See "Conditions" on page 90 for a full description.

### **GROUP BY column-name, ...**

Group multiple rows together from the database. The result will contain one row for each distinct set of values in the named columns. The resulting rows are often referred to as **groups** since there is one row in the result for each group of rows from the table list. For the sake of GROUP BY, all NULL values are treated as identical. Aggregate functions can then be applied to these groups to get meaningful results.

When GROUP BY is used, the select list, HAVING clause and ORDER BY clause cannot reference any columns except those named in the GROUP BY clause.

### **HAVING search-condition**

Restricts which groups will be selected based on the group values and not on the individual row values. The HAVING clause can only be used if either the command has a GROUP BY clause or if the select list consists solely of aggregate functions. Any column names referenced in the HAVING clause must either be in the GROUP BY clause or be used as a parameter to an aggregate function in the HAVING clause.

### **ORDER BY expression, ...**

Sort the results of a query. Each item in the ORDER BY list can be labeled as ASC for ascending order or DESC for descending order. Ascending is assumed if neither is specified. If the expression is an integer **N**, then the query results will be sorted by the **N**'th item in the select list.

## SELECT

---

In embedded SQL, the SELECT command is used for retrieving results from the database and placing the values into host variables via the INTO clause. The SELECT statement must return only one row. For multiple row queries, you must use cursors.

### Examples

```
SELECT * FROM SYS.SYSCATALOG WHERE TNAME LIKE 'SYS%'

SELECT skill.emp_ID, max(skill_level)
FROM Skill, Employee
WHERE s.Address LIKE '%WATERLOO'
      AND s.Birthdate < 'July 15, 1966'
      AND skill.emp_ID = Employee.emp_ID
GROUP BY skill.emp_ID
ORDER BY 2 DESC

SELECT count(*) FROM Employee

SELECT emp_ID, (SELECT skill FROM skill
                WHERE emp_ID.skill = skill.skill)
FROM Employee
```



## CHAPTER 9

# Locking and Concurrency

- About this chapter      Transactions initiated by different connections can overlap. When transactions overlap, and they involve common data in the database, they can affect each other. Watcom SQL uses automatic row level locking to prevent concurrent transactions from interfering with each other.
- Contents
- ◆ "Consistency" on page 104
  - ◆ "Isolation levels" on page 105
  - ◆ "Concurrency" on page 107
  - ◆ "Choosing an isolation level" on page 109
  - ◆ "Primary key generation" on page 110
  - ◆ "Data definition commands" on page 111

## Consistency

There are three inconsistencies 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.

**Non-repeatable read** Transaction A reads a row. Transaction B then modifies or deletes the row and performs a COMMIT. If transaction A then attempts to read the same row again, the row will have changed or been 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.

## Isolation levels

The degree to which the operations in one transaction are visible to the operations in a concurrent transaction is defined by the isolation level. Watcom SQL has 4 different isolation levels that will prevent all or some of the inconsistent behavior.

All isolation levels guarantee that each transaction will execute completely or not at all, and that no updates will be lost. 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.

SQLCA.lock Isolation level	RU 0	RC 1	RR 2	TS 3
Dirty reads	✓	X	X	X
Non-repeatable reads	✓	✓	X	X
Phantom rows	✓	✓	✓	X

The term **cursor stability** means that any row that is the current position of a cursor will not be modified until the cursor leaves the row. Watcom SQL automatically provides cursor stability at isolation levels 1, 2 and 3. (This is actually inherent in the definition of isolation levels 2 and 3).

Isolation level is a database option that can be different for each user. Database options are changed by using the SET command. The default isolation level is 0.

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

```
// Set the lock attribute to read uncommitted
// in the default transaction object SQLCA.
SQLCA.lock = "RU"
```

**When is Lock honored?**

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

## Types of locks

Watcom SQL uses automatic row level locking to prevent concurrent transactions from interfering with each other. The types of locks are described here to help you understand how locking works. All locks for a transaction are held until the transaction is complete (COMMIT or ROLLBACK), with a single exception noted below.

At all isolation levels, **write locks** are used whenever a transaction inserts, updates, or deletes a row. When a row is write-locked, no other transaction can have a lock on the same row (write locks are exclusive).

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—at isolation levels 2 and 3 all locks are held until the end of a transaction.

At isolation levels 2 and 3, **read locks** are used whenever a transaction reads a row. Several transactions can have read locks on the same row (read locks are nonexclusive). A transaction cannot acquire a read lock on a row that already has a write lock by a different transaction. Also, write locks are prevented on any row that is already read locked.

**Phantom locks** are read locks that have been acquired by a transaction with isolation level of 3. They are used to prevent phantom rows.

## Concurrency

**Concurrency** is the degree to which many transactions can be active concurrently. Greater concurrency means improved response time when several users are accessing the database.

## Transaction blocking

Locking is used to prevent concurrent transactions from adversely affecting each other. When locking conflicts arise, one transaction must wait for another transaction to complete. A transaction becomes **blocked** on another transaction.

There is a database option for changing the behavior when blocking occurs. When the database option **BLOCKING** is **ON** (the default), a transaction will just wait until the lock can be acquired. When **BLOCKING** is **OFF**, the request that caused the locking conflict will get an error. (See "SET OPTION" on page 283 for a description of this database option.)

Blocking is more likely to occur with higher isolation levels because more locking and more checking is done. Higher isolation levels provide less concurrency.

Lower isolation levels provide a higher degree of concurrency. When all transactions are running at isolation level 0, the only time a locking conflict will occur 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.

## Deadlock

A deadlock can arise for two different 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. One of the transactions must be canceled. The same situation can arise with many transactions that are blocked in a cycle.

**All active database threads are blocked** When a transaction becomes blocked, its database thread is not relinquished.

Watcom SQL will automatically cancel the last transaction that became blocked (eliminating the deadlock situation), and return an error to that transaction indicating which form of deadlock occurred.

## Choosing an isolation level

Choose isolation levels that are suitable for your application.

Transactions that involve browsing or performing data entry should use isolation level 0 or 1. This type of transaction frequently reads a large number of rows, and lasts several minutes. Concurrency will suffer with isolation level 2 or 3.

Some applications require **serializable** transactions due to the nature of the application. 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 will read few or no rows and last at most a few seconds. 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.

You can change isolation levels within the transaction. When the Lock option is changed in the middle of a transaction, the new setting affects cursors opened after the change and statements executed after the change. This is useful when there is one table or group of tables that require serialized access.

## Primary key generation

Many applications generate primary key values automatically. 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.

- ◆ Use an AUTOINCREMENT value on the field.

Watcom SQL supports an AUTOINCREMENT default value on fields. However, this type of field cannot be used by a PowerBuilder data window for primary key generation, because PowerBuilder is unable to find the record after insertion. If a table in your database is only added to from a PowerBuilder script, you may wish to use the AUTOINCREMENT default value mechanism.



## **Data definition commands**

CREATE INDEX, ALTER TABLE, and DROP will be prevented whenever the command affects a table that is currently being used by another connection. These commands can be time consuming and the server will not process requests referencing the same table while the command is being processed.

CREATE TABLE will not cause any concurrency conflicts.

GRANT, REVOKE, and SET will also not cause concurrency conflicts. These commands will affect any new SQL statements sent to the database engine, but will not affect existing outstanding statements.

## 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 single-user runtime Watcom SQL database engine running on the portable computer. Later, when they return to the office, the transaction log can be translated into an ascii SQL command file (using DBTRAN) and applied to the database server.

## Applying updates

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. Frequently, you can design applications to avoid this sort of problem, but you need to be aware of the possibility.

One solution is to use the `-v` option when starting the database engine on the portable computer. This will cause the engine to record the previous values of each of the columns whenever a row of the database is updated. When the translated transaction log is applied to the database, a warning will be issued if all of the column values currently in the database do not match the values recorded in the transaction log. This will indicate that the particular database row has already been updated by someone else.

## Large databases

With large database files, it might 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, the translated transaction logs from the smaller database can be applied to the main database.



## CHAPTER 10

# Backup and Recovery

**About this chapter** This chapter explains how to use the Watcom SQL logs to protect your data, how to make backup copies of your database files and the Watcom logs, and the recovery procedures for system and media failures.

**Contents**

- ◆ "The need for backups" on page 116
- ◆ "Logs" on page 117
- ◆ "Backups" on page 120
- ◆ "Recovery from system failure" on page 122
- ◆ "Recovery from media failure" on page 123

## The need for backups

Since even the most dependable computers are at risk of failure, Watcom SQL has features to protect your data from two categories of failures: **system failure** and **media failure**.

- ◆ System Failure

Power failure or some other failure causes the machine to go down while there are partially completed transactions. This could be as simple as turning off or rebooting the computer.

- ◆ Media Failure

The database file, the file system, or the device storing the database file has become unusable.

**Recovery from failure** When failures occur, the recovery mechanism must treat transactions as atomic units of work. Any incomplete transaction must be rolled back and any committed transaction must not be lost.

## Logs

Watcom SQL uses three different logs to protect your data from system and media failure. The checkpoint log, the rollback log, and the transaction log all play a role in recovery.

## Checkpoint log

A Watcom SQL 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

A checkpoint can occur for several reasons:

- ◆ The database engine is stopped.
- ◆ The amount of time since the last checkpoint has exceeded 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` command.
- ◆ The database engine is running without a transaction log and a transaction is committed.

### Priority

The priority of writing dirty pages to the disk will increase as the time and the amount of work since the last checkpoint grows. This will be important when the database engine does not have enough idle time to write dirty pages. The database option `CHECKPOINT_TIME` will control the maximum desired time between checkpoints. The database option `RECOVERY_TIME` will control the maximum desired time for recovery in the event of system failure. Both times are specified in minutes.

When the database engine is running with multiple databases, the `CHECKPOINT_TIME` and `RECOVERY_TIME` specified by the first

database started will be used, unless overridden by command line switches. See DBSTART in "Program Summary" for a description of the command line switches.

## 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 ROLLBACK requests, and is also used for recovering from system failure. There is a separate rollback log for each transaction. When a transaction is complete, its rollback log is deleted.

## Transaction log

Everything the database engine does is stored in the transaction log in the order that it occurs. Inserts, updates, deletes, commits, rollbacks, and database structure changes are all logged. The transaction log is frequently referred to as a forward log file.

The transaction log is optional. When you are running Watcom SQL with no transaction log, a checkpoint will be done whenever any transaction is committed. The checkpoint is necessary to ensure that all committed transactions are written to the disk. Writing dirty pages 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.

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.



**Performance hint**

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. By converting a transaction log into a SQL command file, it can serve as an audit trail of changes made to the database. Here is an example of using DBTRAN to convert a transaction log:

```
dbtran sample.log changes.sql
```

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 DBTRAN with the `-a` switch, you can recover transactions that were accidentally canceled by a user. (If `-a` is not specified, DBTRAN will omit transactions that were rolled back.) This would not be a common thing to do, but it can prove useful for exceptional cases.

## Backups

Performing a full backup involves making a copy of the database file. Performing a daily backup involves making a copy of the transaction log. Both full and daily 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.

### Online backup

Backups can be made without stopping the database engine. The DBBACKUP utility that comes with Watcom SQL can be run against a single-user or multiuser database server. See "Program Summary" on page 307 for a full description of the online backup facility.

### Offline backup

The database engine should not be running when you do offline backups by copying database files. Moreover, it should be taken down cleanly.

### Full backup

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 wish to check, execute the validation utility that comes with Watcom SQL.

```
dbvalid -c dba,sql
```

The validation utility will scan every record in every table and look each record up in each index on the table. If the database file is corrupt, you will need to recover from your previous backup.

A full backup is done offline by copying the database file(s) to the backup media. To do a backup while the database engine is running, use the DBBACKUP utility to make a copy of the database file.

```
dbbackup
```

You may need to specify connection parameters.

A full backup should be done according to a regular schedule that you follow carefully. Once every week will work well for most situations.

Keep several previous full backups. If you were to backup 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 off-site 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.

## Daily backup

Daily backups can be done offline by making a copy of the transaction log. The transaction log will have all changes since the most recent full backup. Alternatively, you can make a copy of the transaction log online by running the following command:

```
dbbackup -t
```

Daily backups of the transaction log are recommended. This is even more 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 will 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 (see "Media failure on the database file" on page 123).

## Recovery from system failure

It is advisable to run the system disk verification program after a power failure or other system failure. The DOS or NT console command:

```
chkdsk /f
```

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.

After a system error occurs, Watcom SQL will automatically recover when you restart the database. The results of all transactions that were committed prior to the system error are intact. All changes by transactions that were not committed prior to the system failure are canceled.

The database engine will automatically take three steps to recover from a system failure:

- 1 Restore all pages to the most recent checkpoint by using the checkpoint log.
- 2 Apply any changes that were made between the checkpoint and the system failure. These changes are in the transaction log.
- 3 Rollback all uncommitted transactions by using the rollback logs. There is a separate rollback log for every connection.

Frequent checkpoints will make recovery from system failure take less time, but they will also create work for the database engine writing out dirty pages. 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 Watcom SQL with no transaction log, a checkpoint will be done whenever any transaction is committed. In the event of system failure, the database engine will use 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.

## Recovery from 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 will 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. Alternately, you could use a different device to fill in for the failed device.

### Media failure on the database file

**Situation** Your transaction log is still usable but you have lost your database file.

**Solution** The solution depends on whether you have one transaction log or multiple transaction logs.

### One 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 to apply the transaction log with the `-a` transaction log switch:

```
db32w wsample.db -a sample.log
```

The database engine will apply the transaction log to bring the database up to date.

- 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 the last full backup was done right after a system failure, and you archived the transaction log, you will get an error message because the archived transaction log is required for recovery. Should this happen, you can recover by using both transaction logs (next section).

### 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, the last full backup was on Sunday and the database file is lost during the day on Thursday.

```
db32w wsample.db -a mon.log
db32w wsample.db -a tue.log
db32w wsample.db -a wed.log
db32w wsample.db -a sample.log
```

Watcom SQL will not allow you to apply the transaction logs in the wrong order or to skip a transaction log in the sequence.

- 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.

Provided you have backups, you can always recover all transactions that were committed before the media failure.

### Media failure on the transaction log

**Situation** Your database file is still usable but you have lost your transaction log.

**Solution** To recover:

- 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.

```
db32w wsample.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 rollback any transactions that were not committed at the time of the checkpoint. A new transaction log will be created.

## Consequences

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 will be 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 will minimize the potential for lost data, but they will 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 using a special purpose device, such as a storage device that will mirror the transaction log automatically.





## CHAPTER 11

# Improving Performance

- About this chapter      One of the most important factors affecting database performance is the set of indexes defined on your database tables. This chapter describes indexes and how you can use them to improve performance.
- Contents
- ◆ "Other factors affecting performance" on page 128
  - ◆ "Keys" on page 129
  - ◆ "Indexes" on page 130
  - ◆ "Optimizing joins" on page 131
  - ◆ "Sorting" on page 132
  - ◆ "How the optimizer works" on page 133
  - ◆ "Temporary tables" on page 134
  - ◆ "Using estimates to improve performance" on page 135

## **Other factors affecting performance**

In addition to indexes, other factors can affect performance:

- ◆ You are not running with a transaction log (see "Transaction log" on page 118). A transaction log improves commit time for transactions that insert, update, or delete rows.
- ◆ The database engine does not have an adequate amount of memory for caching database pages. See "Program Summary" on page 307 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 database engines cannot do direct (fast) reading and writing when the database file is very fragmented.

There are several utilities available for DOS, Windows and Windows NT to unfragment your hard disk. One of these should be run periodically. On a computer running DOS, you could put the database on a DOS disk partition by itself to eliminate fragmentation problems.

- ◆ 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.

## Keys

**Foreign keys** and **primary keys** are used for validation purposes. However, Watcom SQL also uses these keys to improve performance where possible.

### Example

This example illustrates how keys are used to make commands execute faster:

```
SELECT * FROM Employee WHERE emp_id = 479
```

The simplest way for Watcom SQL to perform this command is to look at all rows in the Employee table and check the Employee ID in each row to see if it is 479. This does not take long if the table is small, but it will take a long time for very large tables.

Watcom SQL has a built-in mechanism for quickly finding primary and foreign key values. So if your table has a primary or foreign key, Watcom SQL automatically uses the same mechanism to quickly find the employee ID 479. This quick search takes almost the same amount of time whether there are a hundred rows or a million rows in the table.

## Indexes

Sometimes you need to search for something that is not in a key. Hence, Watcom SQL cannot use a key to improve performance.

### Example

Suppose you wanted to look up all the employees named Houston.

The Employee table is searched using an **index** called Emp\_Names. This has been created to facilitate searching for employees by their last name. It was created using the Database painter Create Index window. The user entered the name of the index Emp\_Names and selected the columns (emp\_lname and initials) to be index columns: the SQL statement was generated and submitted to the DBMS:

```
CREATE INDEX Emp_Names ON Employee (emp_lname, initials)
```

The column names emp\_lname and initials indicate that those two columns will be in the index. An index can contain one 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 sorts 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 know the first name only.

In this example, Watcom SQL was able to use the Emp\_Names index to find a particular last name even though the initials were not specified. However, you could not use this index if you were looking for the initials F.K.H. without searching for a particular surname. (You could create an index on just initials to speed up that search.)

Note also that Watcom SQL chose to use the index automatically. Once an index is created, Watcom SQL automatically keeps it up to date and uses it to improve performance whenever it can.

In PowerBuilder and InfoMaker, use the index options in the Database painter to create and drop an index. PowerBuilder and InfoMaker automatically generate the required SQL statements and submit them to Watcom SQL.

For more information about using indexes, see the PowerBuilder or InfoMaker *User's Guide*.

## Optimizing joins

Assume the Employee and Department tables have been **joined** using the key operator. Watcom SQL first examines each row in the Employee table sequentially, then finds the corresponding department for the current row of the Employee table using the primary key for the Department table.

If you modify the command to look up employees in the sales department, Watcom SQL looks in the Department table first, using the foreign key for the Employee table. Once the dept\_id is known, it uses the foreign key for the Employee table (dept\_id) to find all employees in the department.

The interesting thing to note in these two examples is that the tables are examined in a different order. The first time, the Employee table is examined first and then the Department table. The second time, the tables are searched in the opposite order.

When you open more than one table in PowerBuilder or InfoMaker, the tables are automatically joined on a common column or if there is no common column, on a logical column. You can use the Join button or the Join menu option to specify another join. PowerBuilder and InfoMaker automatically generate the required SQL statements and submit them to Watcom SQL.

For more information about joins, see the PowerBuilder or InfoMaker *User's Guide*.

## Sorting

Most queries against a database will have an `ORDER BY` clause so that the rows will come out in a predictable order. Watcom SQL can use indexes to accomplish the ordering quickly.

To specify a sort order, use the `ORDER BY` clause. For example, if you use

```
SELECT * FROM Employee ORDER BY emp_lname
```

then you can use the index on the `emp_lname` column to access the rows of the `Employee` table in alphabetical order by `emp_lname`.

A potential performance problem arises when a query has both a `WHERE` clause and an `ORDER BY` clause.

```
SELECT * FROM Employee
WHERE dept_id = '200'
ORDER BY state
```

The database engine must decide between two strategies:

- ◆ Go through the entire `Employee` table in order by state checking each row to see if it is for department 200.
- ◆ Use the key on the `dept_id` column to read only the employees in department 200. The results would then need to be sorted.

If there are very few employees in department 200, the second strategy is better because only a few rows are scanned and quickly sorted. If most of the employees are in department 200, the first strategy is much better because no sorting is necessary.

This example could be solved by creating a two-column index on state and department number. (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. 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.

In PowerBuilder and InfoMaker, use the Sort options in the painters to sort rows and then automatically generate the required SQL statements and submit them to Watcom SQL.

For more information about sorting, see the PowerBuilder or InfoMaker *User's Guide*.

## 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 must decide which order to access the tables in a query, and whether or not to use an index for each table. The optimizer uses **heuristics** (educated guesses) to help decide the best strategy. 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 operation	Percentage of rows
=	5
<>	95
<, <=, >, >=	25
between	6

The optimizer also makes use of **indexes**, **unique indexes**, and **keys** to improve its guess of the number of rows that will be accessed and the number of I/O operations involved.

## Self tuning

One of the most common constraints in a query is **equality with a column value**. For example,

```
SELECT * FROM Employee WHERE Status = 'a'
```

tests for equality of the Status column. For this type of constraint, the Watcom SQL optimizer will learn from its mistakes. 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.

## Temporary tables

Sometimes Watcom SQL needs to make a **temporary table** for a query. This occurs in the following cases:

- ◆ When a query has an ORDER BY or a GROUP BY and Watcom SQL 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.

In these cases, Watcom SQL 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. The 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 the first case above will improve the performance of these queries since it will not be necessary to build a temporary table.

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.



## Using estimates to improve performance

The Watcom SQL query optimizer uses a heuristic algorithm (educated guesses) to estimate the number of rows a particular query will return and the number of I/O operations that will occur. An estimate is derived for each possible way of performing a particular query. The way that is estimated to take the fewest I/O operations is chosen to retrieve the rows.

Since the query optimizer is guessing at the number of rows in a result based on the size of the tables and particular restrictions 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.

For example, suppose you want a list of all employees in department 200 making over \$75,000 a year. The query optimizer (with no index on the column) guesses that a test for greater than will succeed 25 percent of the time. In this example, the condition on the salary column:

```
salary >= 75000
```

is assumed to choose 25 percent of rows in the salary table. If the actual percentage is closer to, say, 1 percent, the number of rows will be greatly overestimated, and this may lead to poor performance, as tables may be searched in an inefficient order.

We can use an **estimate** to tell the database explicitly what percentage of rows we expect to satisfy the condition. 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 will select. In this case, we estimate:

```
(salary >= 75000, 1 )
```

Supplying explicit estimates can improve performance substantially in those cases where the default estimates lead to poorly optimized queries.



## CHAPTER 12

# User IDs and Permissions

### About this chapter

Most databases are used by many different people. Different users of a database can be given different sets of permissions on various tables in the database to allow them to carry out their tasks while maintaining the security of the database.

This chapter describes how new user IDs can be added to Watcom SQL databases, and how permissions can be managed.

### Contents

- ◆ "Granting and revoking user IDs and permissions" on page 138
- ◆ "User groups" on page 140

# Granting and revoking user IDs and permissions

## Granting new user IDs

Adding new user IDs is the responsibility of the **database administrator (DBA)**. When first created, all Watcom SQL databases have the single user ID **DBA** with password **SQL**.

To create a new user of the database, you must be connected to the database as **DBA**.

Each new user has a user ID and a password. The command to create a new user ID called **new\_user** and password **new\_passwd** is:

```
GRANT CONNECT TO new_user IDENTIFIED BY new_passwd
```

To verify that the new user has been properly created, connect to the database with the new user ID and password.

## Granting permissions on tables

Initially, new users have no permissions to look at tables in the database. Permissions are given using the **GRANT** command, which must be issued from the administrator's user ID.

For example, you can give **new\_user** permission to use the **SELECT** command on the **Employee** table.

```
GRANT SELECT ON Employee TO new_user
```

There are other types of permission you can grant. For example:

```
GRANT UPDATE ON Employee TO new_user  
GRANT DELETE ON Department TO new_user  
GRANT INSERT ON Skills TO new_user
```

These illustrate the various types of permission you can **GRANT** to other user IDs. Update permissions can be granted for a subset of the columns in a table. See "GRANT" on page 249 for the complete syntax of the **GRANT** command.

In addition, you can GRANT more than one type of permission at one time as follows:

```
GRANT INSERT, UPDATE ON Employee TO new_user
```

Finally, you can GRANT permission on a particular table to every user ID by giving permission to the group user ID PUBLIC. For example:

```
GRANT SELECT ON Employee TO PUBLIC
```

## Execute permission on 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 creator of a procedure myproc could allow anotheruser to execute myproc with the statement:

```
GRANT EXECUTE ON myproc TO anotheruser
```

You can revoke the permission granted above with the statement:

```
REVOKE EXECUTE ON myproc FROM anotheruser
```

## DBA and resource authority

By default, new users are not permitted to create tables or procedures. A user requires **resource authority** to create tables or procedures. To give a user resource authority, you must be connected to the database administrator userid.

```
GRANT RESOURCE TO new_user
```

You may want to have more than one user ID with database administrator authority.

```
GRANT DBA TO admin
```

The admin user ID would now have permissions to do anything. (Note that you would need to be connected to DBA to grant DBA authority.)

## User groups

### Creating user groups

When there are many users, they will frequently fall into various categories. It would be cumbersome to grant permissions to each member of a category. A group is just a user ID that is allowed to have members. A user ID may be a member of many groups and a group can have many members. You may connect to a group, and a group can be a member of other groups.

A user group is created by creating the user ID, granting the user ID GROUP authority, and then granting membership in the group to other users.

### Group permissions

A user ID inherits permissions on tables and procedures that have been granted to groups in which the user ID has membership. Since the group ID can be a member of other groups, a user ID can inherit permissions from a hierarchy of groups. Members of a group do not automatically inherit permissions on tables and procedures created by the group; the permissions must be granted explicitly by the group to itself, or by another user. The special user privileges, DBA, RESOURCE, and GROUP authority, are never inherited.

If your database is to have many users, we recommend using groups for controlling permissions for tables and procedures in the database. Never grant permissions to individual users, just GRANT MEMBERSHIP to groups that have the appropriate permissions. Later, when the permissions are no longer needed, you can REVOKE MEMBERSHIP from the group.

### Group tables

Groups are also 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.)

## An example of user groups

Consider the following example of a corporate database. All the tables are created by the **company** user ID. 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 Customers, Products, and Orders. To do this, create a **Sales** user 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:

```
select * from company.Customers
```

To rectify the situation, make the Sales user 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 user group, are indirectly members of the Company user group, and can reference their tables without qualifiers. The command:

```
select * from Customers
```

will now work. Joe and Sally do not have any extra permissions because of their membership in the Company user group. Company 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 will 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.



## CHAPTER 13

# Views

- About this chapter      SQL provides **views** which allow you to give names to frequently executed SELECT commands.
- This chapter describes how to create views, and indicates some of the uses of views.
- Contents                    ♦ "Defining a view" on page 144
- ♦ "Using views for security" on page 146

## Defining a view

### 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
```

This command creates a view called `DepartmentSize` which looks exactly like any other table in the database.

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` command to find 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.

### Column names

Default column names in views are often uninformative. For example, the heading for the column containing the number of employees in each department will be named **expression** by default.

You can rename columns in views to make them clear and informative. In the previous example, you would first get rid of the original view definition as follows:

```
DROP VIEW DepartmentSize
```

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

```
CREATE VIEW DepartmentSize (Dept_ID, NumEmployees) AS
SELECT dept_ID, count(*)
FROM Employee
GROUP BY dept_ID
```

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

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`, 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 dept_name, NumEmployees
       FROM Department, DepartmentSize
       WHERE Department.dept_ID = DepartmentSize.dept_ID
```

## Using views for security

Views can be used to restrict access to information in the database. For example, suppose you wanted to create a user ID for an administrative worker in a particular department, allowing access only to information about their own department.

First, you need to create the new user ID using the GRANT command. (Recall that you need to be connected to the DBA user ID to add a new user.)

The command to create the new user would look like the following:

```
GRANT CONNECT TO new_user IDENTIFIED BY new_passwd
```

Next, you need to restrict the new user to be able to look at information about employees in their own department, which we will identify by department ID 200. For this, you can define a view which only looks at that department as follows:

```
CREATE VIEW Dept200 AS
SELECT emp_lname, emp_fname
FROM Employees
WHERE dept_ID = '200'
```

Now you must give the user permission to look at the new view. Enter the following:

```
GRANT SELECT ON Dept200 TO new_user
```

Notice that you use exactly the same type of command to grant permission for a view as you use to grant permission for a table.

The special tables SYSCATALOG and SYSCOLUMNS are actually views. The definitions of these views are shown in "Watcom SQL System Views" on page 445.

## CHAPTER 14

# Procedures and Triggers

**About this chapter** Procedures and triggers are features for storing procedural SQL statements in the database for use by all applications. This chapter describes the use of procedures and triggers as implemented in the Watcom SQL.

**Contents**

- ◆ "Overview of procedures and triggers" on page 148
- ◆ "Advantages" on page 149
- ◆ "Using procedures" on page 150
- ◆ "Statements" on page 153
- ◆ "Warnings in procedures and triggers" on page 161
- ◆ "Errors in procedures and triggers" on page 162
- ◆ "Transactions and savepoints" on page 165
- ◆ "Single row SELECT" on page 166
- ◆ "Cursors in procedures and triggers" on page 167
- ◆ "Result sets from procedures" on page 170

## Overview of procedures and triggers

Procedures and triggers are features for storing procedural SQL statements in the database for use by all applications. Procedures and triggers allow control statements that allow repetition (LOOP) and conditional execution (IF and CASE) of SQL statements.

Procedures are invoked with a CALL statement, and use parameters to accept values and return values to the caller. A procedure can also return result sets to the caller. A procedure can call other procedures.

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. A trigger can call procedures.

## Advantages

Procedures and triggers are defined in the database, separate from any one database application. This separation provides a number of advantages.

### Standardization

Most importantly, procedures and triggers allow standardization of any actions that are performed by more than one application program. The action is simply 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 only made in one place, and all applications that use the action will 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 involving 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 will be compiled and kept in virtual memory.

### Security

Procedures execute with the table permissions of the creator but can be called by any user that has been granted permission to do so. Triggers execute under the table permissions of the creator 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 program that invoked it. Procedures and triggers can be used to provide security by allowing users limited access to data in tables that they cannot directly examine or modify.

## Using 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.

## Creating procedures

Procedures are created using the `CREATE PROCEDURE` statement. You must have `RESOURCE` authority in order to create a procedure

The following example creates a procedure that accepts two integers in parameters `a` and `b` and returns the one that is greater using parameter `c`.

```
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
```

This statement creates a procedure in the database with the name **greater**. See "CREATE PROCEDURE" on page 207 for a complete description of `CREATE PROCEDURE`. The body of a procedure consists of a compound statement (see "Compound statements in procedures and triggers" on page 153). There is no practical limit to the size of a procedure. There are likewise no limits to the number of parameters that a procedure can have.

Parameter names must conform to the rules for other database identifiers such as column names. They must be one of the types supported by Watcom SQL (see "Data types" on page 66), and must be prefixed by one of the keywords `IN`, `OUT` or `INOUT`. The keywords have the following meanings:

- ◆ **IN** argument is an expression that provides a value to the procedure.
- ◆ **OUT** argument is a variable that could be given a value by the procedure.



- ◆ **INOUT** argument is a variable that provides a value to the procedure, and could be given a new value by the procedure.

For example, the following procedure changes the value provided in the parameter to its absolute value. The parameter **value** provides data to the procedure and also returns the result.

```
CREATE PROCEDURE makeabsolute( INOUT value INT )
BEGIN
    IF value < 0 THEN
        SET value = -value
    END IF
END
```

## Dropping a procedure

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.

To remove the procedure **greater** from the database, execute the statement:

```
DROP PROCEDURE greater
```

## Calling procedures

A procedure is invoked with a **CALL** statement (see "CALL" on page 192). Procedures can be called by an application program or they can be called by other procedures and triggers.

Consider the following example:

```
BEGIN
    DECLARE bigger INT;
    CALL greater( 5, 8, bigger );
END
```

This shows a **CALL** to the procedure created in the previous section from another procedure. It is passing two constants as values for the first two arguments, and receiving the result in the variable **bigger**.

## **Permission 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 creator of a procedure myproc could allow anotheruser to execute myproc with the statement:

```
GRANT EXECUTE ON myproc TO anotheruser
```

You can revoke the permission granted above with the statement:

```
REVOKE EXECUTE ON myproc FROM anotheruser
```

## Statements

This section describes those statements that can be used in constructing procedures and triggers.

### Compound statements in procedures and triggers

The body of a procedure or trigger is a **compound statement**. Compound statements can also be used in control statements within a procedure or trigger.

A compound statement allows one or more SQL statements to be grouped together and treated as a unit. A compound statement starts with the keyword **BEGIN** and ends with the keyword **END**. Immediately following the **BEGIN**, a compound statement can have local declarations that only exist within the compound statement. A compound statement can have a local declaration for a variable, a cursor, a temporary table, or an exception. Local declarations can be referenced by any statement in that compound statement, or in any compound statement nested within it. Local declarations are not visible to other procedures that are called from within a compound statement.

Consider the following example:

```
CREATE PROCEDURE someproc()  
  BEGIN  
    DECLARE x INT;  
    DECLARE y INT;  
  
    SELECT count(*) INTO x FROM sometable;  
    SELECT count(*) INTO y FROM anothertable;  
    BEGIN  
      DECLARE z INT;  
  
      CALL greater( x, y, z );  
    END;  
  END
```

This simple example declares two variables, then uses **SELECT** statements to set values for the variables. It then calls **greater** to determine the greater of the two counts.

## SQL statements in procedures and triggers

The body of a procedure consists of a compound statement. Only certain SQL statements are allowed within a compound statement.

- ◆ SELECT, UPDATE, DELETE, INSERT and SET Variable.
- ◆ Control statements (see "Control statements" on the next page).
- ◆ Cursor statements (see "Cursors in procedures and triggers" on page 167).
- ◆ Exception handling statements (see "Errors in procedures and triggers" on page 162).

COMMIT, ROLLBACK and SAVEPOINT statements are allowed with certain restrictions (see "Transactions and savepoints" on page 165). All other statements (CONNECT, GRANT, REVOKE, CREATE, ALTER, DROP, etc.) are not allowed in procedures and triggers. For further information, see **Usage** for each SQL statement in "Command Syntax" on page 181.

## Atomic statements

An **atomic** statement is a statement that must either execute completely or not execute 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.

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 emp_ID = 123 WHERE emp_ID = 467;
  UPDATE Employee SET birthdate = 'baddata';
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 will succeed. The second one will cause a data conversion error since the value being assigned to the column birthdate cannot be converted to a date.

The result is that the atomic compound statement will fail and the effect of both UPDATE statements will be undone. Even if the currently executing

transaction is eventually committed, neither statement in the atomic compound statement will take effect.

COMMIT and ROLLBACK and some ROLLBACK TO SAVEPOINT statements are not permitted within an atomic compound statement (see "Transactions and savepoints" on page 165).

## **Control statements**

There are a number of control statements for logical flow and decision making in the body of the procedure or trigger. The following is a list of control statements available. See "Command Syntax" on page 181 for complete descriptions.

Control statement	Syntax
Compound statements	BEGIN [ ATOMIC ] statement-list END
Conditional execution	IF condition THEN statement-list ELSEIF condition THEN statement-list ELSE statement-list END IF
CASE	CASE expression WHEN value THEN statement-list WHEN value THEN statement-list ELSE statement-list END CASE
Repetition	WHILE condition LOOP statement-list END LOOP

<b>Control statement</b>	<b>Syntax</b>
Cursor loop	FOR statement-list END FOR
LEAVE	Leave a labeled loop or compound statement. LEAVE label
CALL	Invoke a procedure. CALL procname( arg, ... )

## Triggers

Triggers are used whenever referential integrity and other declarative constraints (see "Referential integrity" on page 62 and "CREATE TABLE" on page 209) are not sufficient. 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.

### Execution permissions

Triggers execute with the table permissions of the creator of the associated table. A trigger can modify rows in a table that a user could not modify directly.

## 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.

There are four types of triggers:

- ◆ **INSERT** Invoked whenever a new row is inserted into the table associated with the trigger.
- ◆ **DELETE** Invoked whenever a row of the associated table is deleted.
- ◆ **UPDATE** Invoked whenever a row of the associated table is updated.
- ◆ **UPDATE OF column-list** Invoked whenever a row of the associated table is updated and a column in the **column-list** has been modified.

Each type of trigger can be defined to execute BEFORE or AFTER the insert, update, or delete. The body of a trigger consists of a compound statement (see "Compound statements in procedures and triggers" on page 153).

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 statements" on page 154). When they fail, all effects of the



statement (including the effects of triggers and any procedures called by triggers) are undone.

COMMIT and ROLLBACK and some ROLLBACK TO SAVEPOINT statements are not permitted within a trigger (see "Transactions and savepoints" on page 165).

## Insert trigger

Consider the following example:

```
TRIGGER mytrigger AFTER INSERT ON Employee
NEW AS new_employee
FOR EACH ROW
BEGIN
    DECLARE err_user_error EXCEPTION FOR SQLSTATE '99999';

    IF new_employee.birthdate > 'June 6, 1994' THEN
        SIGNAL err_user_error
    END IF;
END
```

This example creates a trigger which is fired just after any new row is inserted into the Employee table. It detects and disallows any new rows that correspond to birth dates later than June 6, 1994. Notice that the phrase REFERENCING NEW AS new\_employee allows statements in the trigger code to refer to the data in the new row with the alias new\_employee. Notice also that signaling an error causes the operation that caused the trigger as well as any previous effects of the trigger to be undone.

You can specify that the trigger fire before the row is inserted by changing the first line of the example to:

```
CREATE TRIGGER mytrigger BEFORE INSERT ON Employee
```

## Delete trigger

When defining DELETE triggers, use the following CREATE statement:

```
CREATE TRIGGER mytrigger BEFORE DELETE ON Employee
REFERENCING OLD AS oldrow
FOR EACH ROW
BEGIN
    ...
END
```

This would allow the delete trigger code to refer to the values in the row being deleted using the alias oldrow.

## Update trigger

When defining UPDATE triggers, use the following CREATE statement:

```
CREATE TRIGGER mytrigger BEFORE UPDATE ON Employee
REFERENCING NEW AS after_update
                OLD AS before_update
FOR EACH ROW
BEGIN
    ...
END
```

This allows the UPDATE trigger code to refer to both the old and new values of the row being updated. Columns in the new row are referred to with the alias `after_update` and columns in the old row are referred to with the alias `before_update`.

## Dropping a trigger

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 .

To remove the trigger `mytrigger` from the database, execute the statement:

```
DROP TRIGGER mytrigger
```

## Executing triggers

Triggers are executed automatically whenever an INSERT, UPDATE, or DELETE operation is performed on the table named in the trigger. Triggers are fired for each row. 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 above 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.

## Warnings in procedures and triggers

Whenever a SQL statement is executed, a value is placed in a special variable called `SQLSTATE`. That value indicates whether or not there were any unusual conditions encountered while the statement was being performed.

For example, when using cursors, the `SQLSTATE` variable is used to indicate if a row has been successfully fetched.

```
DECLARE err_notfound EXCEPTION FOR SQLSTATE '02000';
DECLARE c1 CURSOR FOR SELECT emp_ID, emp_lname FROM Employee;

OPEN c1;
emp:
  LOOP
    FETCH NEXT c1 INTO var1, var2;
    IF SQLSTATE = err_notfound THEN
      LEAVE emp
    END IF;
    ...
  END LOOP
CLOSE c1
```

Possible values for the `SQLSTATE` variable are listed in "Database Error Messages" on page 357.

## Errors in procedures and triggers

After an application program executes a SQL statement, it can examine a return code. This return code indicates that the statement executed successfully or that it failed and gives the reason for the failure. This same mechanism is used to indicate the success or failure of a call statement to a procedure.

### Without exception handlers

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 return code. This is true even if the error occurred in a procedure or trigger that was 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.

Consider the following example. The application calls the procedure `myproc`; and `myproc` in turn calls the procedure `anotherproc`, which then encounters an error.

```
CREATE PROCEDURE myproc()
  BEGIN
    ...
    ...
    CALL anotherproc();
    ...*
    ...*
  END

CREATE PROCEDURE anotherproc()
  BEGIN
    ...
    ...
    ...error encountered
    ...*
    ...*
  END
```

In the above example, the lines flagged with the asterisk (\*) will not be executed.

The **traceback** function will provide you with a list of the statements that were executing when the error occurred. The following command will can be executed following an error:

```
SELECT traceback(*) from dummy
```

## With exception handlers

It is often desirable to intercept certain types of errors and handle them within a procedure or trigger in specific ways. This is done through the use of **exception handlers**. An exception handler is defined with the **EXCEPTION** part of a compound statement (see "Compound statements in procedures and triggers" on page 153). It is executed whenever an error occurs in the compound statement. It will also be 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.

Consider the following example:

```
CREATE PROCEDURE myproc()
  BEGIN
    DECLARE column_not_found EXCEPTION FOR SQLSTATE
                                           '52003';

    ...
    ...
    CALL anotherproc();
    ...*
    ...*

    EXCEPTION
      WHEN column_not_found THEN
        ...code to handle the column_not_found error
      WHEN OTHERS THEN
        RESIGNAL
  END

CREATE PROCEDURE anotherproc()
  BEGIN
    ...
    ...
    ...column_not_found error encountered
    ...*
  END
```

In the above example, an exception handler is defined in the procedure `myproc`.

The `DECLARE ... EXCEPTION` statement simply declares a symbolic name for one of the predefined `SQLSTATE` values associated with error conditions already known to the database engine.

The `EXCEPTION` statement declares the exception handler itself. Each `WHEN ... THEN` clause specifies an exception name and the statement(s) 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` simply means that the exception is to be passed on to a higher-level exception handler. This is the default action if `WHEN OTHERS THEN` is not specified in an exception handler.

During execution of the procedure `myproc`, it calls the second procedure `anotherproc`. When the **column not found** error is encountered, an active exception is created. Since there is no exception handler declared at the lower level, the handler declared in the calling procedure is executed to handle the error. If that exception handler were to finish execution without resignaling the error (or signaling another error), the compound statement would be complete and execution would continue with any statements that followed it. The lines flagged with the asterisk (\*) will not be executed.

When an exception is handled in an atomic compound statement, the compound statement completes without an **active** exception and the changes before the exception are not undone.

## Transactions and savepoints

SQL statements in a procedure or trigger are part of the current transaction (see "Transactions" on page 55). 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 statements" on page 154). 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" on page 58) 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.

## Single row SELECT

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 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.



## Cursors in procedures and triggers

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. 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:

```

DECLARE err_notfound EXCEPTION FOR SQLSTATE '02000';
DECLARE c1 CURSOR FOR SELECT emp_ID, emp_lname FROM Employee;

OPEN c1;
emp:
  LOOP
    FETCH NEXT c1 INTO var1, var2;
    IF SQLSTATE = err_notfound THEN
      LEAVE emp
    END IF;
    ...
    ...
  END LOOP
CLOSE c1

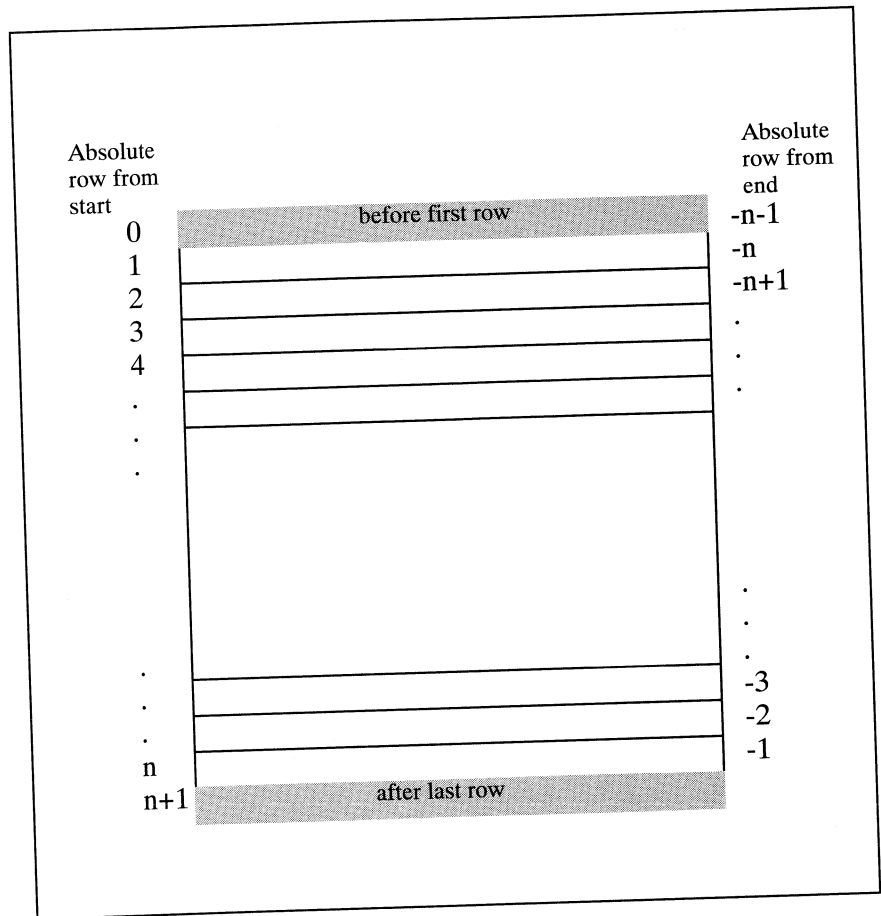
```

### Cursor positioning

A cursor is positioned in one of three places:

- ◆ On a row
- ◆ Before the first row
- ◆ After the last row

This is illustrated below:



When a cursor is opened, it is positioned before the first row. The cursor position can be moved using the FETCH command (see "FETCH" on page 238 ). 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.

**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 Watcom SQL, this occurs if a temporary table had to be created to open the cursor (see "Temporary tables" on page 134 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).

## Result sets from procedures

If a procedure needs to return values to the calling program or procedure, it can use OUT or INOUT parameters. If a procedure needs to return multiple rows, it can use result sets.

Consider the following example:

```
DECLARE err_notfound EXCEPTION FOR SQLSTATE '02000';
DECLARE c1 CURSOR FOR CALL emp_active();

OPEN c1;
emp:
  LOOP
    FETCH NEXT c1 into var1, var2;
    IF SQLSTATE = err_notfound THEN
      LEAVE emp
    END IF;
    ...
    ...
  END LOOP;
RESUME c1;
CLOSE c1;

CREATE PROCEDURE emp_active()
  RESULT ( emp_ID CHAR(10), emp_lname CHAR(20) )
  BEGIN
    SELECT emp_ID, emp_lname FROM Employee WHERE
      Status = 'a';
  END
```

In the above example, the application code declares and opens a cursor on a CALL to the procedure emp\_active. This is analogous to the more common technique of opening a cursor on a SELECT statement.

- ◆ The OPEN statement in the caller causes the procedure emp\_active to begin executing.
- ◆ When the SELECT statement in the procedure is executed, the procedure is suspended and control returns to the statement after the OPEN.
- ◆ The result set of the SELECT is processed by the FETCH statements in the caller.
- ◆ When the caller has finished with the result set, it exits its loop and executes the RESUME statement.
- ◆ The procedure emp\_active resumes execution at the statement after the SELECT. When it completes, control returns to the caller at the statement after the RESUME.
- ◆ The CLOSE statement closes the cursor.

If no RESUME statement was executed by the caller, the CLOSE statement would terminate the procedure without completing execution of the procedure.

The procedure emp\_active has a RESULT set with two columns emp\_ID and emp\_lname: eosname.. A SELECT statement within a procedure must have select-list items that correspond to the RESULT clause of the procedure. Appropriate type conversions will be performed automatically.

## Multiple result sets

It is possible for a procedure to return more than one result set to the calling program. Consider the following example:

```

DECLARE err_notfound EXCEPTION FOR SQLSTATE '02000';
DECLARE c1 CURSOR FOR CALL people();

OPEN c1;
WHILE( SQLSTATE = '00000' ) LOOP
    emp:
        LOOP
            FETCH NEXT c1 INTO var1, var2
            IF SQLSTATE = err_notfound THEN
                LEAVE emp
            END IF;
        END LOOP;
    RESUME c1;
END LOOP;
CLOSE c1;

CREATE PROCEDURE people()
RESULT ( lname CHAR(20), address CHAR(40) )
BEGIN
    SELECT emp_lname, emp_address FROM Employee;
    SELECT head_lname, head_address FROM Department;
END

```

As in the previous example, this application program declares and opens a cursor on the CALL statement.

The WHILE ... LOOP executes once for each SELECT statement in **people**. The nested LOOP fetches rows from each result set.



## CHAPTER 15

# Database Collations

### About this chapter

With Watcom SQL, you can customize sorting orders according to different **collations**. Each collation corresponds to a different character set, so that sorting operations will produce the proper results for the native language of the database.

This chapter documents the built-in collations provided with Watcom SQL.

### Contents

- ◆ "Collations" on page 174
- ◆ "Countries, languages, and code pages" on page 175
- ◆ "Form of the custom collation file" on page 177

## Collations

With Version 3.2, Watcom SQL introduced customizable sorting and comparison order (collation sequences). Built-in collation sequences support many single-byte character sets for various country/language/code-page combinations. Customizable collation sequences allow the user to specify a custom collation sequence to be used by the database in sorting and comparison operations.

Note that the sorting capabilities of the database assume a single-byte, one pass, sorting algorithm. It is not currently possible to sort ligatures (single-byte characters containing two language characters or their equivalent) as two characters, or to subsort within a set of similar characters (such as accented letters).

If no collation is specified when a new database is created, the default collation will be used, which is normal ASCII (binary) ordering for the first 128 characters (hex 00 to 7F). The upper 128 (extended) characters are assumed to be from code page 850 (multinational). Characters that are accented forms of ASCII letters are sorted into the same position as the ASCII letters. Other characters will sort to their binary positions.



## Countries, languages, and code pages

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 Watcom SQL. The table and the corresponding collations were derived from several manuals from IBM concerning National Language Support, subject to the restrictions mentioned above. (At the time of this writing, this table represents the best information that was available. 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
Belorus	Belorussian	855	855CYR		
Brazil	Portuguese	850	850LATIN1	437	437LATIN1
Bulgaria	Bulgarian	855	855CYR	850	850CYR
Canada	Canada French	850	850LATIN1	863	863LATIN1
Canada	English	437	437LATIN1	850	850LATIN1
Czechoslovakia	Czech	852	852LATIN2	850	850LATIN2
Czechoslovakia	Slovakian	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
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
Netherlands	Dutch	850	850LATIN1	437	437LATIN1

*Countries, languages, and code pages*

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
Spain	Spanish	850	850ESP	437	437ESP
Sweden	French	850	850SVE	437	437SVE
Sweden	Italian	850	850SVE	437	437SVE
Sweden	Swedish	850	850SVE	437	437SVE
Switzerland	French	850	850LATIN1	437	437LATIN1
Switzerland	German	850	850LATIN1	437	437LATIN1
Switzerland	Italian	850	850LATIN1	437	437LATIN1
Turkey	Turkish	857	857TRK	850	850TRK
UK	English	850	850LATIN1	437	437LATIN1
USA	English	437	437LATIN1	850	850LATIN1
Venezuela	Spanish	850	850ESP	437	437ESP
Yugoslavia	Croatian	852	852LATIN2	850	850LATIN2
Yugoslavia	Macedonian	852	852LATIN2	850	850LATIN2
Yugoslavia	Serbian Cyrillic	855	855CYR	852	852CYR
Yugoslavia	Serbian Latin	852	852LATIN2	850	850LATIN2
Yugoslavia	Slovenian	852	852LATIN2	850	850LATIN2

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 DOS 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.

## Form of the custom collation file

It is recommended that DBCOLLAT be used to extract a collation from an existing database. SAMPLE.DB may be used, if necessary. A collation should be chosen which is likely to be as close as possible to the wanted collation. The collation may then be edited and specified as input to DBINIT (using the `-z` switch), creating a new database that will use the specified collation.

In the collation file, spaces are generally ignored. Comment lines start with either a `%` or a `--`. The first noncomment line must look like this:

```
Collation label (name)
```

where:

Parameter	Description
Collation	is a keyword and is required
label	is the <code>collation_label</code>
name	is a descriptive term for documentation purposes.

Label will appear in `SYS.SYSCOLLATION.collation_label` and `SYS.SYSINFO.default_collation`, and should be no more than 10 characters. It should not be the same as one of the built-in collations (in particular, the collation label should not be left unchanged.) Name should be no more than 128 characters.

After the "Collation" 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) will sort before characters that appear later.

The general form of each line in the file is:

```
[sort-position] : character
```

or

```
[sort-position] : character [lowercase uppercase]
```

where:

**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 will 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 will be sorted and compared as if they were the same character.

Each character and sort-position is specified in one of the following ways:

<b>\dnnn</b>	Decimal number, using digits 0-9 (such as \d001)
<b>\xhh</b>	Hexadecimal number, using 2 digits 0-9 and/or letters a-f or A-F (such as \xB4)
<b>'c'</b>	Any character in place of c (such as ',')
<b>c</b>	Any character other than quote ('), back-slash (\), colon (:) or comma (,). These 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
: -
: ', '
: ;
: ':'
: !

```

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.

Any characters omitted from the collation will be added to the collation at the position equal to their binary value. `DBINIT` will issue a message for each omitted character. However, it is recommended that any collation contain all 256 characters.



## CHAPTER 16

# Command Syntax

- About this chapter      This chapter presents detailed descriptions of all Watcom SQL commands that are available through PowerBuilder except SELECT, which is discussed in "SELECT" on page 99.
- Some commands (OPEN, CLOSE, FETCH, etc.) are also PowerScript embedded SQL commands. Refer to *PowerScript Language* for a description of how to use these commands in PowerBuilder. The description that is included in the following sections refers to the use of these commands in procedures and triggers.
- Each command begins on a new page. The discussion begins with a summary of the syntax, followed by the purpose, usage, authorization, side effects, other related commands and a detailed description.
- Contents                      The commands are listed alphabetically.

## Conventions

The following conventions are used in the syntax:

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

```
CONNECT [USING TransactionObject];
```

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

```
[ ASC | DESC ]
```

indicates that **ASC** or **DESC** are optional.



# Language elements

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

**column-name**

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

**condition**

An expression that evaluates to TRUE, FALSE, or UNKNOWN. See "Conditions" on page 90.

**connection-name**

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

**creator**

An **identifier** representing a user ID.

**data-type**

A storage data type as described in "Data types" on page 66.

**expression**

An expression as described in "Expressions" on page 84.

**filename**

A **string** containing a filename.

**identifier**

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

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

**number**

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

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

**role-name**

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

**search-condition**

A condition that evaluates to TRUE, FALSE, or UNKNOWN. See "Conditions" on page 90.

**string**

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

```
'This is a string.'
```

```
'John''s database'
```

```
'\x00\x01\x02\x03'
```

**savepoint-name**

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

**statement-label**

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

**table-list**

A list of table names which may include correlation names and join operators. See "FROM" on page 243.

**table-name**

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

**userid**

An **identifier** representing a user name.

**variable**

An **identifier** representing a variable name.

# ALTER DBSPACE

<b>Syntax</b>	ALTER DBSPACE [creator.]dbspace-name  ...   ADD number     RENAME filename
<b>Purpose</b>	To modify the characteristics of the main database file or an extra dbspace.
<b>Usage</b>	DBA notepad.
<b>Authorization</b>	Must have DBA authority.
<b>Side effects</b>	Automatic commit.
<b>See also</b>	CREATE DBSPACE.
<b>Description</b>	<p>The ALTER DBSPACE command modifies the main database file or an extra dbspace. Dbspace names can be found in the SYSDATABASES system table.</p> <p>The ADD command extends the size of the dbspace by the number of pages given by <b>number</b>. This command allows these files to be extended in large amounts before the space is required, rather than the normal 32 pages at a time when the space is needed. This can improve performance for loading large amounts of data and also serves to keep the dbspace files more contiguous within the file system.</p> <p>The RENAME command is used to rename an existing dbspace file, or to change the reference to a file that has been moved, perhaps to a different device.</p>

# ALTER TABLE

## Syntax

ALTER TABLE [creator.]table-name

```

| ADD column-definition [column-constraint ...] |
| ADD table-constraint |
| MODIFY column-definition |
| MODIFY column-name DEFAULT default-value |
| MODIFY column-name [ NOT ] NULL |
... | DELETE column-name | ,...
| DELETE CHECK |
| DELETE UNIQUE ( column-name, ... ) |
| DELETE PRIMARY KEY |
| DELETE FOREIGN KEY role-name |
| RENAME new-table-name |
| RENAME column-name TO new-column-name |

```

column-definition:

```
column-name data-type [ NOT NULL ] [ DEFAULT default-value ]
```

column-constraint:

```

| UNIQUE |
| PRIMARY KEY |
| REFERENCES table-name [( column-name )] [ actions ] |
| CHECK ( condition ) |

```

default-value:

```

| string |
| number |
| AUTOINCREMENT |
| CURRENT DATE |
| CURRENT TIME |
| CURRENT TIMESTAMP |
| NULL |
| USER |

```

table-constraint:

```
| UNIQUE ( column-name, ... ) |
| PRIMARY KEY ( column-name, ... ) |
| CHECK ( condition ) |
| foreign-key-constraint |
```

foreign-key-constraint:

```
[ NOT NULL ] FOREIGN KEY [ role-name ] [(column-name, ... )]
... REFERENCES table-name [(column-name, ... )]
... [ actions ] [ CHECK ON COMMIT ]
```

actions:

```
[ ON UPDATE action ] [ ON DELETE action ]
```

action:

```
| CASCADE |
| SET NULL |
| SET DEFAULT |
| RESTRICT |
```

<b>Purpose</b>	To modify a table definition.
<b>Usage</b>	DBA notepad.
<b>Authorization</b>	Must be the creator of the table or have DBA authority.
<b>Side effects</b>	Automatic commit. The MODIFY and DELETE options close all cursors for the current connection. The ISQL data window is also cleared.
<b>See also</b>	CREATE TABLE, DROP, Data Types.
<b>Description</b>	The ALTER TABLE command changes table attributes (column definitions, constraints) in a table that was previously created. Note that the syntax allows a list of alter clauses; however, only one table-constraint or column-constraint can be added, modified or deleted in one ALTER TABLE statement.

**ADD column-definition**

Add a new column to the table. The table must be empty to specify NOT NULL.

**NULL values**

Watcom SQL optimizes the creation of columns which are allowed to contain the NULL value. The first column that is allowed to contain the NULL value allocates room for eight such columns, and initializes all eight to be the NULL value. (This requires no extra storage.) Thus, the next seven columns added require no changes to the rows of the table. Adding one more column will then allocate room for another eight such columns and then modify each row of the table to allocate the extra space. Consequently, seven out of eight column additions run quickly.

**ADD table-constraint**

Add a constraint to the table. See "CREATE TABLE" on page 209 for a full explanation of table constraints.

If PRIMARY KEY is specified, the table must not already have a primary key created by the CREATE TABLE command or another ALTER TABLE command.

**MODIFY column-definition**

Change the length or data type of an existing column in a table. If NOT NULL is specified, a NOT NULL constraint is added to the named column. Otherwise, the NOT NULL constraint for the column will not be changed. If necessary, the data in the modified column will be converted to the new data type. If a conversion error occurs, the operation will fail and the table will be left unchanged.

**Deleting an index, constraint, or key**

If the column is contained in a uniqueness constraint, a foreign key, or a primary key then the constraint or key must be deleted before the column can be modified. If a primary key is deleted, all foreign keys referencing the table will also be deleted. You cannot MODIFY a table or column constraint. To change a constraint, you must DELETE the old constraint and ADD the new constraint.

**MODIFY column-name DEFAULT default-value**

Change the default value of an existing column in a table. To remove a default value for a column, specify DEFAULT NULL.

### **MODIFY column-name [ NOT ] NULL**

Change the NOT NULL constraint on the column to allow or disallow NULL values in the column.

### **DELETE column-name**

Delete the column from the table. If the column is contained in any index, uniqueness constraint, foreign key, or primary key then the index, constraint or key must be deleted before the column can be deleted. This does not delete CHECK constraints that refer to the column.

### **DELETE CHECK**

Delete all check constraints for the table. This includes both table check constraints and column check constraints.

### **DELETE UNIQUE (column-name,...)**

Delete a uniqueness constraint for this table. Any foreign keys referencing this uniqueness constraint (rather than the primary key) will also be deleted.

### **DELETE PRIMARY KEY**

Delete the primary key constraint for this table. All foreign keys referencing the primary key for this table will also be deleted.

### **DELETE FOREIGN KEY role-name**

Delete the foreign key constraint for this table with the given role name.

### **RENAME new-table-name**

Change the name of the table to the **new-table-name**. Note that any applications using the old table name will need to be modified. Also, any foreign keys which were automatically assigned the same name as the old table name will not change names.

### **RENAME column-name TO new-column-name**

Change the name of the column to the **new-column-name**. Note that any applications using the old column name will need to be modified.

ALTER TABLE will be prevented whenever the command affects a table that is currently being used by another connection. ALTER TABLE can be time consuming and the server will not process requests referencing the same table while the command is being processed.



## Examples

1. ALTER TABLE student ADD weight decimal(4,2)  
CHECK ( weight > 0 )
2. ALTER TABLE student ADD FOREIGN KEY advisor (advisor)  
REFERENCES employee (empnum)

# CALL

<b>Syntax</b>	CALL procedure-name ( expression [ ,... ] ) CALL procedure-name ( parameter-name = expression [ ,... ] )
<b>Purpose</b>	Invoke a procedure.
<b>Usage</b>	Anywhere.
<b>Authorization</b>	Must be the creator of the procedure, have EXECUTE permission for the procedure, or have DBA authority.
<b>Side effects</b>	None.
<b>See also</b>	"Procedures and Triggers" on page 147, CREATE PROCEDURE, GRANT, DECLARE.
<b>Description</b>	<p>The CALL statement invokes a procedure that has been previously created with a CREATE PROCEDURE statement. When the procedure completes, any INOUT or OUT parameter values will be copied back.</p> <p>The argument list can be specified by position or by using keyword format. By position, the arguments will match up with the corresponding parameter in the parameter list for the procedure. By keyword, the arguments are matched up with the named parameters. In either format, arguments for OUT parameters are optional and all other arguments are mandatory.</p> <p>Inside a procedure, a CALL statement can be used in a DECLARE statement when the procedure returns result sets (see "Result sets from procedures" on page 170).</p>

## Examples

1. `CALL Calculate_Report( 'Marketing', 'April' )`
2. `DECLARE median_mark SMALLINT;  
    .  
    .  
    CALL Calculate_Median( median_mark );`

# CASE

<b>Syntax</b>	<pre>CASE value-expression     ... WHEN [ constant   NULL ] THEN statement-list ...     ... [ WHEN [ constant   NULL ] THEN statement-list ] ...     ... ELSE statement-list END CASE</pre>
<b>Purpose</b>	Select execution path based on multiple cases.
<b>Usage</b>	Procedures and triggers only.
<b>Authorization</b>	None.
<b>Side effects</b>	None.
<b>See also</b>	"Procedures and Triggers" on page 147, Compound statements.
<b>Description</b>	The CASE statement is a control statement that allows you to choose a list of SQL statements to execute based on the value of an expression. If a WHEN clause exists for the value of <b>value-expression</b> , the <b>statement-list</b> in the WHEN clause is executed. If no appropriate WHEN clause exists, and an ELSE clause exists, the <b>statement-list</b> in the ELSE clause is executed. Execution resumes at the first statement after the END CASE.

## Example

```
CASE sex
WHEN 'f' THEN
    set salutation = 'Madam'
WHEN 'm' THEN
    set salutation = 'Sir'
WHEN NULL THEN
    set salutation = 'Sir/Madam'
END CASE
```

# CHECKPOINT

<b>Syntax</b>	CHECKPOINT
<b>Purpose</b>	To <b>checkpoint</b> the database.
<b>Usage</b>	Anywhere.
<b>Authorization</b>	Must have DBA authority to CHECKPOINT when you are running with multiple users over a LAN. For single-user, no authorization is required.
<b>Side effects</b>	None.
<b>Description</b>	The CHECKPOINT command will checkpoint the database. Checkpoints are also performed automatically by the database engine. It is not normally required for an application to ever issue the CHECKPOINT command. For a full description of checkpoints, see "Backup and Recovery" on page 115.

# CLOSE

<b>Syntax</b>	CLOSE cursor-name  cursor-name:            identifier
<b>Purpose</b>	To close a cursor.
<b>Usage</b>	Procedures and triggers only.
<b>Authorization</b>	The cursor must have been previously opened.
<b>Side effects</b>	None.
<b>See also</b>	CLOSE in <i>PowerScript Language</i> , OPEN, DECLARE CURSOR.
<b>Description</b>	This statement closes the named cursor.

## Procedure/trigger example

```
BEGIN
  DECLARE student_cursor CURSOR FOR
    SELECT surname FROM student;
  DECLARE name CHAR(40);

  OPEN student_cursor;
  LOOP
    FETCH NEXT student_cursor into name;
    ...
  ENDLOOP
  CLOSE student_cursor;
END
```

# COMMENT

**Syntax** COMMENT ON

```

      | TABLE [creator.]table-name           |
      | INDEX [creator.]index-name           |
      | COLUMN [creator.]table-name.column-name | IS comment
      | FOREIGN KEY [creator.]table-name.role-name |
      | USER userid                          |
      | PROCEDURE [creator.]procedure-name    |
      | TRIGGER [creator.]trigger-name       |

```

comment:

```

      | string                                |
      | NULL                                  |

```

**Purpose** Store a comment in the system tables for a table, an index, a column, a foreign key, a userid, a procedure, or a trigger.

**Usage** DBA notepad.

**Authorization** Must either be the creator of the table, index, procedure, or trigger, or have DBA authority.

**Side effects** Automatic commit.

**Description** Several system tables have a column named Remarks that allows you to associate a comment with a database item (SYSUSERPERM, SYSTABLE, SYSCOLUMN, SYSINDEX, SYSFORIGNKEY, SYSPROCEDURE, SYSTRIGGER). The COMMENT ON command allows you to set the Remarks column in these system tables. A comment can be removed by setting it to NULL.

## Examples

1. COMMENT ON TABLE Student IS "Student information"
2. COMMENT ON TABLE Student IS NULL

# COMMIT

<b>Syntax</b>	COMMIT WORK
<b>Purpose</b>	To make any changes to the database permanent.
<b>Usage</b>	Anywhere.
<b>Authorization</b>	Must be connected to the database.
<b>Side effects</b>	Closes all cursors that were not opened with the WITH HOLD option.
<b>See also</b>	ROLLBACK, PREPARE TO COMMIT.
<b>Description</b>	<p>The COMMIT command ends a logical unit of work (transaction) and makes all changes made during this transaction permanent in the database. A transaction is defined as the database work done between successful COMMIT and ROLLBACK commands on a single database connection.</p> <p>The COMMIT command is also used as the second phase of a two-phase commit operation. See "Two-phase commit" on page 59 and "PREPARE TO COMMIT" on page 271 for more information.</p> <p>The changes committed are those made by the data manipulation commands: INSERT, UPDATE, and DELETE, as well as the ISQL load command INPUT.</p> <p>The data definition commands all do an automatic commit. They are:</p> <ul style="list-style-type: none"><li>ALTER</li><li>COMMENT</li><li>CREATE</li><li>DROP</li><li>GRANT</li><li>REVOKE</li><li>SET OPTION</li></ul> <p>The COMMIT command will fail if Watcom SQL detects any invalid foreign keys. This makes it impossible to end a transaction with any invalid foreign keys. Usually, foreign key integrity is checked on each data manipulation operation. However, if either the database option WAIT_FOR_COMMIT is set ON or a particular foreign key was defined with a CHECK ON COMMIT clause, the database engine will not check integrity until the COMMIT</p>

## *COMMIT*

---

command is executed. For a two-phase commit operation, these errors will be reported on the first phase (PREPARE TO COMMIT), not on the second phase (COMMIT).



## Compound statements

<b>Syntax</b>	<pre>[ statement-label : ]     ... BEGIN [ [ NOT ] ATOMIC ]     ...   [ local-declaration ; ... ]     ...   statement-list     ...   [ EXCEPTION [ exception-case ... ] ]     ... END [ statement-label ]</pre> <p>local-declaration:</p> <pre>      variable-declaration                   cursor-declaration                     exception-declaration                  temporary-table-declaration    </pre> <p>variable-declaration:</p> <pre>    DECLARE variable-name data-type</pre> <p>exception-declaration:</p> <pre>    DECLARE exception-name EXCEPTION FOR SQLSTATE     [ VALUE ] string</pre> <p>exception-case:</p> <pre>      WHEN exception-name [ ,... ] THEN statement-list         WHEN OTHERS THEN statement-list                  </pre>
<b>Purpose</b>	Specify a statement that groups other statements together.
<b>Usage</b>	Procedures and triggers only.
<b>Authorization</b>	None.
<b>Side effects</b>	None.

**See also** "Procedures and Triggers" on page 147, DECLARE CURSOR, DECLARE TEMPORARY TABLE, LEAVE, SIGNAL, RESIGNAL.

**Description** The body of a procedure or trigger is a **compound statement**. Compound statements can also be used in control statements within a procedure or trigger.

A compound statement allows one or more SQL statements to be grouped together and treated as a unit. A compound statement starts with the keyword BEGIN and ends with the keyword END. Immediately following the BEGIN, a compound statement can have local declarations that only exist within the compound statement. A compound statement can have a local declaration for a variable, a cursor, a temporary table, or an exception. Local declarations can be referenced by any statement in that compound statement, or in any compound statement nested within it. Local declarations are not visible to other procedures that are called from within a compound statement.

If the ending **statement-label** is specified, it must match the beginning **statement-label**. The LEAVE statement can be used to resume execution at the first statement after the compound statement. The compound statement that is the body of a procedure or triggers has an implicit label that is the same as the name of the procedure or trigger.

See "Procedures and Triggers" on page 147 for a complete description of compound statements and exception handling.

**Example**

```
BEGIN
  DECLARE column_not_found
    EXCEPTION FOR SQLSTATE '52003';
  DECLARE str CHAR(10);
  DECLARE i INTEGER;

  . . .

  EXCEPTION
    WHEN column_not_found THEN
      set message='column not found'
    WHEN OTHERS THEN
      RESIGNAL
END
```

# CONFIGURE

**Syntax** CONFIGURE

**Purpose** To activate the ISQL configuration window.

**Usage** ISQL.

**Authorization** None.

**Side effects** None.

**See also** SET OPTION.

**Description** The CONFIGURE command activates the ISQL configuration window. This window displays the current settings of all ISQL options. It does not display or allow you to modify database options or options for other software that are stored in the database. The SET OPTIONS command (see "SET OPTION" on page 283) allows you to set all options—ISQL, database and other options.

The **Tab** key can be used to change fields. If a hardware cursor appears in a field, then you are allowed to type a value for the option. If the hardware cursor does not appear, you can use the cursor up and down keys to select one of the allowed values. When you have changed the options, press **Enter** to save the changes and exit, or **Esc** to undo the changes and exit.

If you press **Enter** and you have selected "Save Options to Database" then the options will be written to the SYSOPTION table in the database and the database engine will perform an automatic COMMIT. If you do not select "Save Options to Database", then the options are set temporarily and remain in effect for the current database connection only.

# CONNECT

**Syntax**                   CONNECT [ TO engine-name ] [ DATABASE database-name ]  
                              ... [ AS connection-name ]  
                              ... [ USER ] userid IDENTIFIED BY password

engine-name:                identifier, string or host-variable  
database-name:             identifier, string or host-variable  
connection-name:          identifier, string or host-variable  
userid:                     identifier, string or host-variable  
password:                  identifier, string or host-variable

**Purpose**                    To establish a connection to a database.

**Usage**                    ISQL.

**Authorization**         None.

**Side effects**            None.

**See also**                 GRANT, DISCONNECT, SET CONNECTION.

**Description**            The CONNECT command establishes a connection to the database identified by **database-name** running on the engine or server identified by **engine-name**.

If no **engine-name** is specified, the default local database engine will be assumed (the first database engine started).

No other database commands are allowed until a successful CONNECT command has been executed.

The userid and password are used for checking the permissions on SQL commands. If the password or the userid and password are not specified, the user will be prompted to type the missing information.

If you are connected to a userid with DBA authority, you can connect to another userid without specifying a password. (The output of DBTRAN requires this capability.) In ISQL, you can connect without a password with the command:

```
connect admin
```

A connection can optionally be named by specifying the AS clause. This allows multiple connections to the same database, or multiple connections to the same or different database servers, all simultaneously. Each connection has its own associated transaction. You may even get locking conflicts between your transactions if, for example, you try to modify the same record in the same database from two different connections.

Multiple connections are managed through the concept of a current connection. After a successful connect statement, the new connection becomes the current one. To switch to a different connection, use the SET CONNECTION statement. The DISCONNECT statement is used to drop connections.

### ISQL examples

1. CONNECT  
- isql will prompt for a userid and password
2. CONNECT USER admin  
- isql will prompt for password
3. CONNECT TO sample USER admin IDENTIFIED BY school

## CREATE DBSPACE

<b>Syntax</b>	CREATE DBSPACE [creator.]dbspace-name AS filename
<b>Purpose</b>	To create a new database file. This file may be on a different device.
<b>Usage</b>	DBA notepad.
<b>Authorization</b>	Must have DBA authority.
<b>Side effects</b>	Automatic commit.
<b>See also</b>	DROP.
<b>Description</b>	<p>The CREATE DBSPACE command creates a new database file. When a database is first initialized using DBINIT, it is composed of one file. All tables and indexes created are placed in that file. CREATE DBSPACE will add a new file to the database. This file can be on a different disk drive than the root file allowing the creation of databases larger than one physical device.</p>

A *filename* without an explicit directory will be created in the same directory as the main database file. Any relative directory will be relative to the main database file. When you are using a Watcom SQL Network Server, the *filename* is a filename on the server machine. When you are using the Network Server for NetWare, the *filename* should use a volume name (not a drive letter) when an absolute directory is specified.

Tables are contained entirely within one database file. This means that a database table and its associated indexes cannot use more space than one file. The IN clause of the CREATE TABLE command specifies into which file a table will go. Tables are put into the root database file by default.

### Example

```
CREATE DBSPACE library AS 'library.db';

CREATE TABLE Library_Books (
    title          char(100),
    author         char(50),
    isbn           char(30),
) IN library;
```

# CREATE INDEX

<b>Syntax</b>	<pre>CREATE [UNIQUE] INDEX [creator.]index-name     ... ON [creator.]table-name     ... ( column-name [ ASC   DESC ], ... )</pre>
<b>Purpose</b>	To create an index on a specified table. Indexes are used to improve database performance.
<b>Usage</b>	DBA notepad.  To create an index using PowerBuilder or InfoMaker, use Create Index icon on the Database painter toolbar. For more information about creating a database table using PowerBuilder or InfoMaker, see the PowerBuilder or InfoMaker <i>User's Guide</i> .
<b>Authorization</b>	Must be the creator of the table or have DBA authority.
<b>Side effects</b>	Automatic commit.
<b>See also</b>	DROP.
<b>Description</b>	<p>The CREATE INDEX command creates a sorted index on the specified columns of the named table. Indexes are automatically used by Watcom SQL to improve the performance of queries issued to the database as well as for sorting queries where an ORDER BY clause is specified. Once an index is created, it is never referenced again except to delete it using the DROP INDEX command.</p> <p>The UNIQUE constraint ensures that there will not be two rows in the table with identical values in all the columns in the index.</p> <p>Columns are sorted in ascending (increasing) order unless descending (DESC) is explicitly specified. An index will be used by Watcom SQL for both an ascending and a descending ORDER BY, no matter whether the index was ascending or descending. However, if an ORDER BY is performed with mixed ascending and descending attributes, an index will be used only if the index was created with the same ascending and descending attributes.</p> <p>Indexes cannot be created for views.</p>

## CREATE INDEX

---

CREATE INDEX will be prevented whenever the command affects a table that is currently being used by another connection. CREATE INDEX can be time consuming and the server will not process requests referencing the same table while the command is being processed.

### Examples

1. 

```
CREATE INDEX department_name_index
ON department.dept (dept_name)
```
2. 

```
CREATE INDEX birthdate_index ON employee
(birthdate DESC, emp_lname ASC )
```



# CREATE PROCEDURE

**Syntax**

```
CREATE PROCEDURE [creator.]procedure-name ( [ parameter , ... ] )
    ... [ RESULT ( result-column , ... ) ]
    ... compound-statement
```

parameter:

```
| parameter_mode parameter-name data-type |
| SQLCODE |
| SQLSTATE |
```

parameter\_mode:

```
| IN |
| OUT |
| INOUT |
```

result-column:

```
column-name data-type
```

**Purpose** To create a new procedure in the database.

**Usage** DBA notepad.

**Authorization** Must have RESOURCE authority.

**Side effects** Automatic commit.

**See also** "Procedures and Triggers" on page 147, DROP, CALL, Compound statements, GRANT.

**Description** The CREATE PROCEDURE command creates (stores) a procedure in the database. A procedure can be created for another user by specifying a **creator name**. A procedure is invoked with a CALL statement.

Parameter names must conform to the rules for other database identifiers such as column names. They must be one of the types supported by Watcom

SQL (see "Data types" on page 66), and must be prefixed by one of the keywords IN, OUT or INOUT. The keywords have the following meanings:

- ◆ **IN** argument is an expression that provides a value to the procedure.
- ◆ **OUT** argument is a variable that could be given a value by the procedure.
- ◆ **INOUT** argument is a variable that provides a value to the procedure, and could be given a new value by the procedure.

**SQLSTATE** and **SQLCODE** are special parameters that will output the SQLSTATE or SQLCODE value when the procedure ends (they are **OUT** parameters).

A procedure that returns result sets ("Result sets from procedures" on page 170) can have a **RESULT** clause. The parenthesized list following the **RESULT** keyword defines the number of result columns and name and type. This information is returned by the **DESCRIBE** command when a **CALL** statement is being described. Allowable data types are listed in "Data types" on page 66.

The body of a procedure consists of a compound statement.

### Command delimiter

The default ISQL command delimiter (a single semi-colon) must be changed in order to create procedures from ISQL. See the "COMMAND\_DELIMITER" option in "SET OPTION" on page 283)

### Example

```
CREATE PROCEDURE change_name( INPUT nbr INTEGER,
                              INPUT newname CHAR( 20 ) )
BEGIN
  UPDATE Student SET surname=newname
  WHERE studnum = nbr;
END
```

# CREATE TABLE

## Syntax

```
CREATE [ GLOBAL TEMPORARY ] TABLE [creator.]table-name
    ... ( | column-definition [ column-constraint ... ] |, ... )
        | table-constraint |
    ... [ IN [creator.]dbspace-name ]
    ... [ | ON COMMIT DELETE ROWS | ]
        | ON COMMIT PRESERVE ROWS | ]
```

column-definition:

```
column-name data-type [ NOT NULL ] [ DEFAULT default-value ]
```

column-constraint:

```
| UNIQUE |
| PRIMARY KEY |
| REFERENCES table-name [( column-name )] [ actions ] |
| CHECK ( condition ) |
```

default-value:

```
| string |
| number |
| AUTOINCREMENT |
| CURRENT DATE |
| CURRENT TIME |
| CURRENT TIMESTAMP |
| NULL |
| USER |
```

## CREATE TABLE

---

table-constraint:

```
| UNIQUE ( column-name, ... ) |
| PRIMARY KEY ( column-name, ... ) |
| CHECK ( condition ) |
| foreign-key-constraint |
```

foreign-key-constraint:

```
[NOT NULL] FOREIGN KEY [role-name] [(column-name, ... )]
... REFERENCES table-name [(column-name, ... )]
... [ actions ] [ CHECK ON COMMIT ]
```

actions:

```
[ ON UPDATE action ] [ ON DELETE action ]
```

action:

```
| CASCADE |
| SET NULL |
| SET DEFAULT |
| RESTRICT |
```

### Purpose

To create a new table in the database.

### Usage

DBA notepad.

### Authorization

Must have RESOURCE authority. To create a table for another user, you must have DBA authority.

### Side effects

Automatic commit.

### See also

"Referential integrity" on page 62, DROP, ALTER TABLE, CREATE DBSPACE, Data Types.

### Description

The CREATE TABLE command creates a new table. A table can be created for another user by specifying a **creator** name. If GLOBAL TEMPORARY is not specified, the table is referred to as a **base** table. Otherwise, the table is a **temporary** table.

The IN clause is used to specify in which database file the base table will be created. See "CREATE DBSPACE" on page 204 for more information.

A created temporary table is a table that exists in the database like a base table and remains in the database until it is explicitly removed by a DROP TABLE command. The rows in a temporary table are only visible to the connection that inserted the rows. Multiple connections from the same or different applications can use the same temporary table at the same time and each connection will only see its own rows. The rows of a temporary table are deleted when the connection ends.

The ON COMMIT clause is only allowed for temporary tables. By default, the rows of a temporary table are deleted on COMMIT.

The parenthesized list following the CREATE TABLE command can contain the following clauses in any order:

**column-name data-type [ NOT NULL ] [ DEFAULT default-value ]**

Define a column in the table. Allowable data types are described in "Data types" on page 66. Two columns in the same table cannot have the same name.

If NOT NULL is specified, or if the column is in a UNIQUE or PRIMARY KEY constraint, the column cannot contain any NULL values. If a DEFAULT value is specified, it will be used as the value for the column in any INSERT statement which does not specify a value for the column. If no DEFAULT is specified, it is equivalent to DEFAULT NULL.

When using DEFAULT AUTOINCREMENT, the **data-type** must be one of INTEGER, SMALLINT, FLOAT, or DOUBLE. On INSERTs into the table, if a value is not specified for the autoincrement column, a unique value will be generated. If a value is specified, it will be used. If the value is larger than the current maximum value for the column, that value will be used as a starting point for subsequent INSERTs.

Deleting rows will not decrement the autoincrement counter. Gaps created by deleting rows can only be filled by explicit assignment when using an insert. After doing an explicit insert of a row number less than the maximum, subsequent rows without explicit assignment will be autoincremented with a value of one greater than the previous maximum.

For performance reasons, it is highly recommended that DEFAULT AUTOINCREMENT only be used with columns defined as a PRIMARY KEY or with a UNIQUE constraint; or columns that are the first column of an index. This will allow the maximum value determined at startup time to be found without scanning the entire table.

### table-constraint

Table constraints help ensure the integrity of data in the database. There are four types of integrity constraints:

- ◆ **Unique constraints** identify one or more columns that uniquely identify each row in the table.
- ◆ A **primary key constraint** is the same as a unique constraint except that a table can have only one primary key constraint. The primary key usually identifies the best identifier for a row. For example, the customer number might be the primary key for the customer table.
- ◆ A **foreign key constraint** restricts the values for a set of columns to match the values in a primary key or uniqueness constraint of another table. For example, a foreign key constraint could be used to ensure that a customer number in an invoice table corresponds to a customer number in the customer table.
- ◆ A **check constraint** allows arbitrary conditions to be verified. For example, a check constraint could be used to ensure that a column called Sex only contains the values male or female.

If a statement would cause changes to the database that would violate an integrity constraint, the statement is effectively not executed and an error is reported. (*Effectively* means that any changes made by the statement before the error was detected are undone.)

### column-constraint

Column constraints are abbreviations for the corresponding table constraints. For example, the following are equivalent:

1. 

```
CREATE TABLE Product (  
    product_num    integer UNIQUE  
)
```
2. 

```
CREATE TABLE Product (  
    product_num    integer,  
    UNIQUE ( product_num )  
)
```

Column constraints are normally used unless the constraint references more than one column in the table. In these cases, a table constraint must be used.

## Integrity constraints

### column-definition **UNIQUE** or **UNIQUE ( column-name, ... )**

No two rows in the table can have the same values in all the named column(s). A table may have more than one unique constraint.

#### **Unique constraint versus unique index**

There is a difference between a **unique constraint** and a **unique index**. Columns in a unique index are allowed to be NULL, while columns in a unique constraint are not. Also, a foreign key can reference a unique constraint but not a unique index.

### column-definition **PRIMARY KEY** or **PRIMARY KEY ( column-name, ... )**

The primary key for the table will consist of the listed column(s), and none of the named column(s) can contain any NULL values. Watcom SQL ensures that each row in the table will have a unique primary key value. A table can have only one **PRIMARY KEY**.

When the second form is used (**PRIMARY KEY** followed by a list of columns), the primary key is created including the columns in the order in which they are defined, not the order in which they are listed.

### column-definition **REFERENCES primary-table-name [(primary-column-name)]**

The column is a **foreign key** for the primary key or a unique constraint in the primary table. Normally, a foreign key would be for a primary key rather than a unique constraint. If a primary column name is specified, it must match a column in the primary table which is subject to a unique constraint or primary key constraint, and that constraint must consist of only that one column. Otherwise the foreign key references the primary key of the second table.

A temporary table cannot have a foreign key that references a base table and a base table cannot have a foreign key that references a temporary table.

### **FOREIGN KEY [role-name] [(...)] REFERENCES primary-table-name [(...)]**

The table contains a **foreign key** for the primary key or a unique constraint in another table. Normally, a foreign key would be for a primary key rather than a unique constraint. (In this description, this other table will be called the **primary table**.)

If the primary table column names are not specified, then the primary table columns will be the columns in the table's primary key. If foreign key column names are not specified then the foreign key columns will have the same names as the columns in the primary table. If foreign key column names are specified, then the primary key column names must be specified, and the column names are paired according to position in the lists.

Any foreign key column not explicitly defined will automatically be created with the same data type as the corresponding column in the primary table. These automatically created columns cannot be part of the primary key of the foreign table. Thus, a column used in both a primary key and foreign key must be explicitly created.

The role name is the name of the foreign key. The main function of the role name is to distinguish two foreign keys to the same table. If no role name is specified, the role name will be set to the name of the primary table.

The referential integrity action defines the action to be taken to maintain foreign key relationships in the database. Whenever a primary key value is changed or deleted from a database table, there may be corresponding foreign key values in other tables that should be modified in some way. You can specify either an ON UPDATE clause, an ON DELETE clause, or both, followed by one of the following actions:

**CASCADE** When used with ON UPDATE, update the corresponding foreign keys to match the new primary key value. When used with ON DELETE, deletes the rows from the table that match the deleted primary key.

**SET NULL** Sets to NULL all the foreign key values that correspond to the updated or deleted primary key.

**SET DEFAULT** Sets to the value specified by the column(s) DEFAULT clause, all the foreign key values that match the updated or deleted primary key value.

**RESTRICT** Generates an error if an attempt is made to update or delete a primary key value while there are corresponding foreign keys elsewhere in the database. This was the only form of referential integrity prior to Watcom SQL Version 4.0 and is the default action if no action is specified.

The CHECK ON COMMIT clause will cause the database to wait for a COMMIT before checking the integrity of this foreign key, overriding the setting of the WAIT\_FOR\_COMMIT database option.



A temporary table cannot have a foreign key that references a base table and a base table cannot have a foreign key that references a temporary table.

**column-definition CHECK ( condition ) or CHECK ( condition )**

No row is allowed to fail the condition. If an INSERT or UPDATE statement would cause a row to fail the condition, the operation is not permitted and the effects of the statement are undone.

**When is the change rejected?**

The change is rejected only if the condition is FALSE; in particular, the change is allowed if the condition is UNKNOWN. (See "NULL value" on page 263 and "Conditions" on page 90 for more information about TRUE, FALSE, and UNKNOWN conditions.)

**Examples**

```

1. CREATE TABLE library_books (
    // NOT NULL is assumed for primary key columns
    isbn          CHAR(20) PRIMARY KEY,
    copyright_date DATE,
    title         CHAR(100),
    author        CHAR(50),
    // column(s) corresponding to primary key of room
    // will be created
    FOREIGN KEY location REFERENCES room
)

2. CREATE TABLE borrowed_book (
    // Default on insert is that book is borrowed today
    date_borrowed DATE NOT NULL DEFAULT CURRENT DATE,
    // date_returned will be NULL until the book is returned
    date_returned DATE,
    book          CHAR(20) REFERENCES library_books
(isbn),
    // The check condition is UNKNOWN until
    // the book is returned, which is allowed
    CHECK( date_returned >= date_borrowed )
)

```

## CREATE TABLE

---

```
3. CREATE TABLE Orders (
    order_num    INTEGER NOT NULL PRIMARY KEY,
    date_ordered DATE,
    name         CHAR(80)
);

CREATE TABLE Order_item (
    order_num    INTEGER NOT NULL,
    item_num     SMALLINT NOT NULL,
    PRIMARY KEY (order_num, item_num),
    // When an order is deleted, delete all of its items.
    FOREIGN KEY (order_num) REFERENCES Orders (order_num)
    ON DELETE CASCADE
);
```

# CREATE TRIGGER

## Syntax

```
CREATE TRIGGER trigger-name trigger-time trigger-event
    ... [ ORDER integer ] ON table-name
    ... [ REFERENCING [ OLD AS old-name ] [ NEW AS new-name ] ]
    ... FOR EACH ROW
    ... [ WHEN ( search-condition ) ]
    ... compound-statement
```

trigger-time:

```
| BEFORE
| AFTER
```

trigger-event:

```
| DELETE
| INSERT
| UPDATE
| UPDATE OF column-list
```

## Purpose

To create a new trigger in the database.

## Usage

DBA notepad.

## Authorization

Must have RESOURCE authority and have ALTER permissions on the table, or must have DBA authority.

## Side effects

Automatic commit.

## See also

"Procedures and Triggers" on page 147, Compound statements, CREATE PROCEDURE, DROP.

## Description

The CREATE TRIGGER command creates a trigger associated with a table in the database and stores the trigger in the database.

There are four types of triggers:

- ◆ **INSERT** Invoked whenever a new row is inserted into the table associated with the trigger.
- ◆ **DELETE** Invoked whenever a row of the associated table is deleted.
- ◆ **UPDATE** Invoked whenever a row of the associated table is updated.
- ◆ **UPDATE OF column-list** Invoked whenever a row of the associated table is updated and a column in the **column-list** has been modified.

Each type of trigger can be defined to execute **BEFORE** or **AFTER** the insert, update, or delete. The body of a trigger consists of a compound statement.

Triggers of the same type (insert, update, or delete) that fire at the same time (before or after) can use the **ORDER** clause to determine the order that the triggers are fired.

The **REFERENCING OLD** clause allows you to refer to the values in a row prior to an update or delete. The **REFERENCING NEW** clause allows you to refer to the inserted or updated values. The **OLD** and **NEW** rows can be referenced in **BEFORE** and **AFTER** triggers. The **REFERENCING NEW** clause allows you to modify the new row in a **BEFORE** trigger before the insert or update operation takes place.

The **WHEN** clause will cause the trigger to only fire for rows where the search-condition evaluates to true.

### Command delimiter

The default ISQL command delimiter (a single semi-colon) must be changed in order to create triggers from ISQL. See the "COMMAND\_DELIMITER" option in "SET OPTION" on page 283)

### Example

```
CREATE TRIGGER record_mark_changes AFTER UPDATE ON Mark
REFERENCING NEW AS newmark
FOR EACH ROW
BEGIN
  INSERT INTO Mark_History
    (studnum, course, newmark, changed_by)
  VALUES (newmark.studnum, newmark.course,
    newmark.mark, USER )
END
```

# CREATE VARIABLE

**Syntax** CREATE VARIABLE identifier data-type

**Purpose** To create a SQL variable.

**Usage** DBA notepad.

**Authorization** None.

**Side effects** None.

**See also** Data Types, DROP VARIABLE, SET variable, Compound statements.

**Description** The CREATE VARIABLE command creates a new variable of the specified data type. The variable contains the NULL value until it is assigned a different value by the SET VARIABLE command.

A variable can be used in a SQL statement anywhere a column name is allowed. If there is no column name that matches the identifier, Watcom SQL checks to see if there is a variable that matches and uses its value.

Variables belong to the current connection, and disappear when you disconnect from the database or when you use the DROP VARIABLE command. Variables are not visible to other connections. Variables are not affected by COMMIT or ROLLBACK statements.

Variables are useful for creating large text or binary objects for INSERT or UPDATE statements from Embedded SQL programs.

Variables created by CREATE VARIABLE can be used in any SQL statement or in any procedure or trigger. Local variables in procedures and triggers are declared within a compound statement (see "Compound statements in procedures and triggers" on page 153).

**Example** The following code fragment could be used to insert a large text value into the database.

## CREATE VARIABLE

---

```
EXEC SQL BEGIN DECLARE SECTION;
char          buffer[5000];
EXEC SQL END DECLARE SECTION;

EXEC SQL CREATE VARIABLE hold_blob LONG VARCHAR;

EXEC SQL SET hold_blob = '';
for(;;) {
    /* read some data into buffer ... */
    size = fread( buffer, 1, 5000, fp );
    if( size <= 0 ) break;

    /* add data to blob using concatenation
       Note that concatenation works for binary data too! */
    EXEC SQL SET hold_blob = hold_blob || :buffer;
}

EXEC SQL INSERT INTO some_table VALUES ( 1, hold_blob );

EXEC SQL DROP VARIABLE hold_blob;
```

# CREATE VIEW

<b>Syntax</b>	<pre>CREATE VIEW [creator.]view-name [( column-name, ... )]     ... AS select-without-order-by</pre>
<b>Purpose</b>	To create a view on the database. Views are used to give a different perspective on the data even though it is not stored that way.
<b>Usage</b>	DBA notepad.  To create a view using PowerBuilder or InfoMaker, use the View painter. To open the View painter, click the View button on the Database painter toolbar. For more information about creating a database table using PowerBuilder or InfoMaker, see the PowerBuilder or InfoMaker <i>User's Guide</i> .
<b>Authorization</b>	Must have RESOURCE authority and SELECT permission on the tables in the view definition. Must have DBA authority to create a view for another user.
<b>Side effects</b>	Automatic commit.
<b>See also</b>	DROP, CREATE TABLE.
<b>Description</b>	<p>The CREATE VIEW command creates a view with the given name. A view can be created for another user by specifying the <b>creator</b>. A view name can be used in place of a table name in SELECT, DELETE, UPDATE, and INSERT commands. Views, however, do not physically exist in the database as tables. They are derived each time they are used. The view is derived as the result of the SELECT statement specified in the CREATE VIEW command. Table names used in a view should be qualified by the userid of the table creator. Otherwise, a different userid might not be able to find the table or might get the wrong table.</p> <p>The columns in the view are given the names specified in the column name list. If the column name list is not specified, then the view columns are given names from the select list items. In order to use the names from the select list items, the items must be a simple column name or they must have an alias-name specified (see "SELECT" on page 99).</p>

**ORDER BY clause**

The SELECT statement is not allowed to have an ORDER BY clause on it. It may have a GROUP BY clause and may be a UNION.

**Examples**

1. 

```
CREATE VIEW male_employee AS
SELECT * FROM Employee WHERE Sex = 'M'
```
2. 

```
CREATE VIEW emp_dept as
SELECT emp_id, sum( factor * salary ) / 5
FROM Employee KEY JOIN Department
GROUP BY emp_id
```



# DBTOOL

## Syntax

```

| alter-database
| alter-writefile
| backup-to
| compress-database
| create-database
| create-writefile

```

DBTOOL

```

| dbinfo-database
| drop-database
| translate
| uncompress-database
| unload-collation
| unload-tables
| validate-tables

```

alter-database:

```
ALTER DATABASE name
```

```

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

```

alter-writefile:

```
ALTER WRITEFILE name [ REFER TO dbname ]
```

backup-to:

```
BACKUP TO directory
```

```

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

```

```

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

```

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

compress-database:

COMPRESS DATABASE filename [ TO filename ]

create-database:

CREATE DATABASE filename

```

... | [ NO [ TRANSACTION ] LOG ] |
    | [ [ TRANSACTION ] LOG TO filename ] |
... | [ IGNORE CASE ] |
    | [ RESPECT CASE ] |
... [ PAGE SIZE n ] [ COLLATION name ]
... [ ENCRYPT ] [ TRAILING SPACES ]

```

create-writefile:

CREATE WRITEFILE name FOR DATABASE name

... [ [ TRANSACTION ] LOG TO logname ] [ NOCONFIRM ]

dbinfo-database:

DBINFO DATABASE filename TO filename [ [ WITH ] PAGE USAGE ]

drop-database:

DROP DATABASE name [ NOCONFIRM ]

translate:

TRANSLATE [ TRANSACTION ] LOG FROM logname

```

... [ TO sqlfile ] [ WITH ROLLBACKS ]
... | [ USERS u1, u2, ... , un ] |
    | [ EXCLUDE USERS u1, u2, ... , un ] |
... [ LAST CHECKPOINT ] [ ANSI ] [ NOCONFIRM ]

```

uncompress-database:

```
UNCOMPRESS DATABASE filename [ TO filename ] [ NOCONFIRM ]
```

unload-collation:

```
UNLOAD COLLATION [ name ] TO filename
```

```
... [ EMPTY MAPPINGS ] [ HEX | HEXADEDECIMAL ] [ NOCONFIRM ]
```

unload-tables:

```
UNLOAD TABLES TO directory [ RELOAD FILE TO filename ]
```

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

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

validate-tables:

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

connection-string: string of connection parameters

<b>Purpose</b>	To invoke one of the database tools.
<b>Usage</b>	ISQL.
<b>Authorization</b>	None.
<b>Side effects</b>	None.
<b>See also</b>	"Program Summary" on page 307, "Connection parameters" on page 53.
<b>Description</b>	<p>The DBTOOL command invokes one of the database utilities. All of the database utilities are available without leaving ISQL.</p> <p>The following table lists the database utility invoked by each DBTOOL command. See "Program Summary" on page 307 for more information on the database utility programs.</p>

## DBTOOL

Command	Database tool
DBTOOL ALTER DATABASE	DBLOG
DBTOOL ALTER WRITEFILE	DBWRITE
DBTOOL BACKUP TO	DBBACKUP
DBTOOL COMPRESS DATABASE	DBSHRINK
DBTOOL CREATE DATABASE	DBINIT
DBTOOL CREATE WRITEFILE	DBWRITE
DBTOOL DBINFO DATABASE	DBINFO
DBTOOL DROP DATABASE	DBERASE
DBTOOL TRANSLATE	DBTRAN
DBTOOL UNCOMPRESS DATABASE	DBEXPAND
DBTOOL UNLOAD COLLATION	DBCOLLAT
DBTOOL UNLOAD TABLES	DBUNLOAD
DBTOOL VALIDATE TABLES	DBVALID

# DECLARE CURSOR

<b>Syntax</b>	<pre> DECLARE cursor-name [ SCROLL   NO SCROLL   DYNAMIC SCROLL ]     ...             CURSOR FOR statement            ...           [ FOR UPDATE   FOR READ ONLY ]  cursor-name:      identifier </pre>
<b>Purpose</b>	To declare a cursor. Cursors are the primary means for retrieving data from the database using PowerScript embedded SQL.
<b>Usage</b>	Procedures and triggers only.
<b>Authorization</b>	None.
<b>Side effects</b>	None.
<b>See also</b>	DECLARE CURSOR in <i>PowerScript Language</i> , "Compound statements in procedures and triggers" on page 153, OPEN, SELECT, CALL.
<b>Description</b>	<p>The DECLARE statement declares a cursor with the specified name for a SELECT statement or a CALL statement.</p> <p>A cursor declared FOR READ ONLY may not be used in an UPDATE (positioned) or a DELETE (positioned) operation. FOR UPDATE is the default.</p> <p>A cursor declared NO SCROLL is restricted to FETCH NEXT and FETCH RELATIVE 0 seek operations. A cursor declared SCROLL or DYNAMIC SCROLL can use all formats of the FETCH command. DYNAMIC SCROLL is the default.</p> <p>SCROLL cursors behave differently from DYNAMIC SCROLL cursors when the rows in the cursor are modified or deleted after the first time the row is read. SCROLL cursors have more predictable behavior when changes happen.</p> <p>Each row fetched in a SCROLL cursor is remembered. If one of these rows is deleted, either by your program or by another program in a multiuser environment, it creates a "hole" in the cursor. If you fetch the row at this "hole" with a SCROLL cursor, Watcom SQL returns the error</p>

SQLSTATE\_NO\_CURRENT\_ROW indicating that the row has been deleted, and leaves the cursor positioned on the "hole". (A DYNAMIC SCROLL cursor will just skip the "hole" and retrieve the next row.) This allows your application to remember row positions within a cursor and be assured that these positions will not change. For example, an application could remember that EADIE is the second row in the cursor for SELECT \* FROM STUDENT. If the first student (RUSSELL) is deleted while the SCROLL cursor is still open, FETCH ABSOLUTE 2 will still position on EADIE while FETCH ABSOLUTE 1 will return SQLSTATE\_NO\_CURRENT\_ROW. Similarly, if the cursor is on EADIE, FETCH PREVIOUS will return SQLSTATE\_NO\_CURRENT\_ROW.

In addition, a fetch on a SCROLL cursor will return the warning SQLSTATE\_ROW\_UPDATED\_WARNING if the row has changed since it was last read. (The warning only happens once; fetching the same row a third time will not produce the warning.) Similarly, an UPDATE (positioned) or DELETE (positioned) statement on a row that has been modified since it was last fetched will return the error SQLSTATE\_ROW\_UPDATED\_SINCE\_READ and abort the statement. An application must FETCH the row again before the UPDATE or DELETE will be permitted. Note that an update to any column will cause the warning/error, even if the column is not referenced by the cursor. For example, a cursor on Surname and Initials would report the update even if only the Birthdate column were modified. These update warning and error conditions will not occur in bulk operations mode (-b database engine command line switch) when row locking is disabled.

Clearly, Watcom SQL maintains more information about SCROLL cursors than DYNAMIC SCROLL cursors; thus, DYNAMIC SCROLL cursors are more efficient and should be used unless the consistent behavior of SCROLL cursors is required. There is no extra overhead in Watcom SQL for DYNAMIC SCROLL cursors versus NO SCROLL cursors.

### NOTE

The behavior of SCROLL cursors has changed since Watcom SQL Version 3.0. The SCROLL cursors in Watcom SQL Version 3.0 were equivalent to DYNAMIC SCROLL cursors now. Cursors declared with the default scrolling behavior will not have changed since the old default behavior was the SCROLL and the new default behavior is DYNAMIC SCROLL.

### Procedure/trigger example

```
BEGIN
  DECLARE student_cursor CURSOR FOR
    SELECT surname FROM student;
  DECLARE name CHAR(40);

  OPEN student_cursor;
  LOOP
    FETCH NEXT student_cursor into name;
    ...
  ENDLOOP
  CLOSE student_cursor;
END
```

# DECLARE TEMPORARY TABLE

**Syntax**                    DECLARE LOCAL TEMPORARY TABLE table-name

```
      ... ( | column-definition [ column-constraint ... ] | , ... )
           | table-constraint                               |
      ... [ | ON COMMIT DELETE ROWS                       | ]
           | ON COMMIT PRESERVE ROWS                     |
```

**Purpose**                    To declare a local temporary table.

**Usage**                    Procedures and triggers only.

**Authorization**         Must have RESOURCE authority.

**Side effects**            None.

**See also**                "Compound statements in procedures and triggers" on page 153, CREATE TABLE

**Description**            The DECLARE LOCAL TEMPORARY TABLE statement declares a temporary table. See "CREATE TABLE" on page 209 for definitions of **column-definition**, **column-constraint**, and **table-constraint**.

Declared local temporary tables within compound statements exist within the compound statement (see "Compound statements in procedures and triggers" on page 153). Otherwise, the declared local temporary table exists until the end of the connection.

By default, the rows of a temporary table are deleted on COMMIT.

## Procedure/trigger example

```
BEGIN
  DECLARE LOCAL TEMPORARY TABLE TempTab (
    number INT
  );
  ...
END
```



# DELETE

<b>Syntax</b>	DELETE FROM [creator.]table-name [WHERE search-condition]
<b>Purpose</b>	To delete rows from the database.
<b>Usage</b>	Anywhere.
<b>Authorization</b>	Must have DELETE permission on the table.
<b>Side effects</b>	None.
<b>See also</b>	INSERT, INPUT.
<b>Description</b>	<p>The DELETE command deletes all the rows from the named table that satisfy the search condition. If no WHERE clause is specified, all rows from the named table are deleted.</p> <p>The DELETE command can be used on views provided the SELECT command defining the view has only one table in the FROM clause and does not contain a GROUP BY clause, an aggregate function, or involve a UNION operation.</p> <p>Internally, PowerBuilder processes DELETE, INSERT, and UPDATE commands the same way. PowerBuilder inspects them for any PowerBuilder variable references and replaces all references with a constant that conforms to Watcom SQL rules for the data type.</p>

## Examples

1. DELETE FROM employee WHERE emp\_id = 86000
2. DELETE FROM skill  
WHERE skill = 'salesman' AND level = 4

## **DELETE (positioned)**

**Syntax**                    `DELETE [FROM [creator.]table-name]`  
                              `...     WHERE CURRENT OF cursor-name`

cursor-name:                identifier, or host-variable  
table-name:                identifier  
creator:                    identifier

**Purpose**                    To delete the data at the current location of a cursor.

**Usage**                    Procedures and triggers only.

**Authorization**         Must have DELETE permission on tables used in the cursor.

**Side effects**            None.

**See also**                UPDATE, UPDATE (positioned), INSERT.

**Description**            This form of the DELETE statement deletes the current row of the specified cursor. The current row is defined to be the last row fetched from the cursor.

The positioned DELETE command can be used on a cursor open on a view as long as the SELECT command defining the view does not contain the GROUP BY clause or an aggregate function.

If the cursor is for a joined query (including using a view containing a join), then the FROM clause must be used. Only the current row of the specified table is deleted. The other tables involved in the join are not affected.

### **Example**

```
DELETE WHERE CURRENT OF emp_cursor;
```

# DISCONNECT

## Syntax

DISCONNECT

```

      | connection-name
    ... | [ CURRENT ]
      | ALL

```

connection-name: identifier, string or host-variable

## Purpose

To drop a connection with the database.

## Usage

ISQL.

## Authorization

None.

## Side effects

None.

## See also

CONNECT, SET CONNECTION.

## Description

The DISCONNECT command drops a connection with the database engine and releases all resources used by it. If the connection to be dropped was named on the CONNECT statement, then the name can be specified. Specifying ALL will drop all of the application's connections to all database environments. CURRENT is the default and will drop the current connection.

An implicit ROLLBACK is executed on connections that are dropped.

## ISQL example

```
DISCONNECT ALL
```

# DROP

<b>Syntax</b>	<pre> DROP   DBSPACE [creator.]dbspace-name   INDEX [creator.]index-name   TABLE [creator.]table-name   VIEW [creator.]view-name   PROCEDURE [creator.]procedure-name   TRIGGER [creator.]trigger-name </pre>
<b>Purpose</b>	To remove dbspaces, indexes, tables, views, procedures and triggers from the database.
<b>Usage</b>	<p>DBA notepad.</p> <p>To drop a database table, view or key using PowerBuilder or InfoMaker, use Drop table/View/Key on the Options menu in the Database painter. For more information about using PowerBuilder or InfoMaker to drop a table, view, or key, see the PowerBuilder or InfoMaker <i>User's Guide</i>.</p>
<b>Authorization</b>	For DROP DBSPACE, must have DBA authority. For others, must be the creator of the table, index, view, procedure, or trigger, or have DBA authority.
<b>Side effects</b>	Automatic commit. Clears the Data window in ISQL.
<b>See also</b>	CREATE DBSPACE, CREATE TABLE, CREATE INDEX, CREATE VIEW, CREATE PROCEDURE, CREATE TRIGGER, ALTER TABLE.
<b>Description</b>	<p>The DROP command removes the definition of the indicated database structure. If the structure is a dbspace, then all tables in that dbspace must be dropped prior to dropping the dbspace. If the structure is a table, all data in the table is automatically deleted as part of the dropping process. Also, all indexes and keys for the table are dropped by the DROP TABLE command.</p> <p>DROP TABLE, DROP INDEX and DROP DBSPACE will be prevented whenever the command affects a table that is currently being used by another connection. DROP PROCEDURE will be prevented when the procedure is in use by another connection.</p>
<b>Examples</b>	<ol style="list-style-type: none"> <li>1. DROP TABLE department</li> <li>2. DROP VIEW emp_dept</li> </ol>

# DROP OPTIMIZER STATISTICS

<b>Syntax</b>	DROP OPTIMIZER STATISTICS
<b>Purpose</b>	To reset optimizer statistics.
<b>Usage</b>	Anywhere.
<b>Authorization</b>	Must have DBA authority.
<b>Side effects</b>	None.
<b>Description</b>	<p>The DROP OPTIMIZER STATISTICS command resets optimizer statistics. The Watcom SQL optimizer maintains statistics as it evaluates queries and uses these statistics to make better decisions optimizing subsequent queries. These statistics are stored permanently in the database. The DROP OPTIMIZER STATISTICS command resets these statistics to default values. This command is most useful when first learning about the optimizer. It helps to better understand the process used by the optimizer.</p>

## **DROP VARIABLE**

<b>Syntax</b>	DROP VARIABLE identifier
<b>Purpose</b>	To eliminate a SQL variable.
<b>Usage</b>	DBA notepad.
<b>Authorization</b>	None.
<b>Side effects</b>	None.
<b>See also</b>	CREATE VARIABLE, SET variable.
<b>Description</b>	The DROP VARIABLE command eliminates a SQL variable previously created using the CREATE VARIABLE command. Variables will be automatically eliminated when the database connection is released. However, variables are often used for large objects. Thus, eliminating them after use may free up significant resources (primarily disk space).

# EXIT

<b>Syntax</b>	EXIT   QUIT   BYE	   
<b>Purpose</b>	To leave ISQL.	
<b>Usage</b>	ISQL.	
<b>Authorization</b>	None.	
<b>Side effects</b>	Will do a commit if option COMMIT_ON_EXIT is ON (default); otherwise will do a rollback.	
<b>See also</b>	SET COMMIT_ON_EXIT.	
<b>Description</b>	Leave the ISQL environment and return to the operating system. This will close your connection with the database. Before doing so, ISQL will perform a COMMIT operation if the COMMIT_ON_EXIT option is ON. If the option is OFF, ISQL will perform a ROLLBACK. The default action is to COMMIT any changes you have made to the database.	





forward and a negative number indicates moving backwards. Thus, a NEXT is equivalent to RELATIVE 1 and PRIOR is equivalent to RELATIVE -1. RELATIVE 0 retrieves the same row as the last fetch statement on this cursor.

The ABSOLUTE positioning parameter is used to go to a particular row. A zero indicates the position before the first row (see "Cursors in procedures and triggers" on page 167). A one (1) indicates the first row, and so on. Negative numbers are used to specify an absolute position from the end of the cursor. A negative one (-1) indicates the last row of the cursor. FIRST is a short form for ABSOLUTE 1. LAST is a short form for ABSOLUTE -1.

The OPEN statement initially positions the cursor before the first row (see "Cursors in procedures and triggers" on page 167). If the fetch includes a positioning parameter and the position is outside the allowable cursor positions, then the SQLSTATE\_NOTFOUND warning is issued.

#### **Cursor positioning problems**

Inserts and some updates to DYNAMIC SCROLL cursors can cause problems with cursor positioning. The database engine will not put inserted rows at a predictable position within a cursor unless there is an ORDER BY clause on the SELECT statement. In some cases, the inserted row will not appear at all until the cursor is closed and opened again. With Watcom SQL, this occurs if a temporary table had to be created to open the cursor (see "Temporary tables" on page 134 for a description). The UPDATE statement may cause a row to move in the cursor. This will happen if the cursor has an ORDER BY that uses an existing index (a temporary table is not created).

The FOR UPDATE clause indicates that the fetched row will subsequently be updated with an UPDATE WHERE CURRENT OF CURSOR statement. This clause causes the database engine to put a write lock on the row. The lock will be held until the end of the current transaction. See "Locking and Concurrency" on page 103.

**Procedure/trigger example**

```
BEGIN
  DECLARE student_cursor CURSOR FOR
    SELECT surname FROM student;
  DECLARE name CHAR(40);

  OPEN student_cursor;
  LOOP
    FETCH NEXT student_cursor into name;
    ...
  ENDLOOP
  CLOSE student_cursor;
END
```

# FOR

<b>Syntax</b>	<pre>[ statement-label : ] ...FOR for-loop-name AS cursor-name ...   CURSOR FOR statement   ... [ FOR UPDATE   FOR READ ONLY ] ... DO statement-list ...END FOR [ statement-label ]</pre>
<b>Purpose</b>	Repeat the execution of a statement list once for each row in a cursor.
<b>Usage</b>	Procedures and triggers only.
<b>Authorization</b>	None.
<b>Side effects</b>	None.
<b>See also</b>	LOOP, DECLARE CURSOR, FETCH, LEAVE.
<b>Description</b>	<p>The FOR statement is a control statement that allows you to execute a list of SQL statements once for each row in a cursor. The FOR statement is equivalent to a compound statement with a DECLARE for the cursor and a DECLARE of a variable for each column in the result set of the cursor followed by a loop that fetches one row from the cursor into the local variables and executes <b>statement-list</b> once for each row in the cursor.</p> <p>The name and data-type of the local variables that are declared are derived from the <b>statement</b> used in the cursor. With a SELECT statement, the data-type will be the data-type of the expressions in the select list. The names will be the select list item aliases where they exist; otherwise, they will be the name of the columns. Any select list item that is not a simple column reference must have an alias. With a CALL statement, the names and data-types will be taken from the RESULT clause in the procedure definition.</p> <p>The LEAVE statement can be used to resume execution at the first statement after the END FOR. If the ending <b>statement-label</b> is specified, it must match the beginning <b>statement-label</b>.</p>

## *FOR*

---

### **Example**

```
FOR names AS curs CURSOR FOR SELECT surname FROM Student
DO
    CALL search_for_name( surname );
END FOR;
```

# FROM

## Syntax

```
... FROM table-expr, ...
```

table-expr:

```
| table-spec |
| table-expr join-type table-spec [ ON condition ] |
| ( table-expr, ... ) |
```

table-spec:

```
[userid . ] table-name [ [AS] correlation-name ]
```

join-type:

```
| CROSS JOIN |
| [ NATURAL | KEY ] JOIN |
| [ NATURAL | KEY ] INNER JOIN |
| [ NATURAL | KEY ] LEFT OUTER JOIN |
| [ NATURAL | KEY ] RIGHT OUTER JOIN |
```

<b>Purpose</b>	To specify the database tables or views involved in a SELECT or UPDATE statement.
<b>Usage</b>	Anywhere.
<b>Authorization</b>	Must be connected to the database.
<b>Side effects</b>	None.
<b>See also</b>	SELECT, UPDATE, DELETE, Conditions.
<b>Description</b>	The SELECT and UPDATE commands require a table list to specify which tables will be used by the command.

**Note**

Although this description refers to tables, it applies to views unless otherwise noted.

The FROM table list creates a result set consisting of all the columns from all the tables specified. Initially, all combinations of rows in the component tables are in the result set, and the number of combinations is usually reduced by **join conditions** and/or WHERE conditions.

Tables owned by a different user can be qualified by specifying the **user ID**. Tables owned by groups to which the current user belongs will be found by default without specifying the user ID.

The **correlation name** is used to give a temporary name to the table for this SQL command only. This is useful when referencing columns which must be qualified by a table name but the table name is long and cumbersome to type. The correlation name is also necessary to distinguish between table instances when referencing the same table more than once in the same query. If no correlation name is specified, then the table name is used as the correlation name for the current command.

If the same correlation name is used twice for the same table in a table expression, that table is treated as if it were only listed once. For example, in

```
FROM Employee KEY JOIN Department,  
Employee KEY JOIN Skill
```

the two instances of the Employee table are treated as one instance, and is equivalent to

```
FROM Department KEY JOIN Employee KEY JOIN Skill
```

whereas

```
FROM Person HUSBAND, Person WIFE
```

would be treated as two instances of the Person table, with different correlation names HUSBAND and WIFE.

**Joining tables**

A JOIN reduces the result set based on the **join type** and **join condition**. The join types are described below and the join condition is specified after the keyword ON.

Parentheses can also be used to join one table to more than one other table. For example,

```
A JOIN (B,C)
```

joins table A to both tables B and C. This syntax is equivalent to

```
A JOIN B, A JOIN C.
```

Table expressions can be arbitrarily complex. For example,

```
A JOIN B JOIN C
```

```
A JOIN ( B, C JOIN D )
```

are legal and meaningful table list expressions. (Any of the valid join types could have been used in the above examples.)

### Cross join

A CROSS JOIN does not restrict the results of the join, so the following are equivalent:

```
FROM table1 CROSS JOIN table2
```

```
FROM table1, table2
```

### Natural join

A NATURAL JOIN restricts the results by comparing the values of columns in the two tables with the same column name. An error is reported if there are no common column names. A join condition can optionally be specified which further restricts the results of the join.

Column names such as Description or Address often cause a NATURAL JOIN to return different results than expected.

### Key join

A KEY JOIN restricts the result set based on a foreign key relationship between the two tables. (This was the only type of automatic join supported in Watcom SQL Version 3.0.) A join condition can optionally be specified which further restricts the results of the join. A key join is valid if exactly one foreign key is identified between the two tables; otherwise, an error indicating the ambiguity is reported.

Parentheses can also be used to join one table to more than one other table. For example

```
A KEY JOIN (B,C)
```

joins table A to both tables B and C. This syntax is equivalent to

```
A KEY JOIN B, A KEY JOIN C.
```

Thus, a join involving parentheses is valid if there is an unambiguous join for each table listed.

A KEY JOIN with a view is valid if there is a valid KEY JOIN with exactly one of the tables in the FROM table list of the view definition.

There are two conditions where the meaning of a KEY JOIN is ambiguous. In the first condition, there are two tables A and B where A has a foreign key for B and B has a foreign key for A. The second condition occurs when a table A has two foreign keys for a table B. In either case, the primary table *must* have a correlation name which is the same as the **role name** of the foreign key. Otherwise an error will be reported. For example, every Watcom SQL database has a table called SYSTABLEPERM to hold permission information. Every row in the permission table has two foreign keys for the table describing user IDs (SYSUSERPERM). One foreign key is for the user ID giving the permission (role name **grantor**) and the other is the user ID getting the permission (role name **grantee**). A KEY JOIN for the **grantor** would look like the following:

```
SYSUSERPERM grantor KEY JOIN SYSTABLEPERM
```

and a KEY JOIN for both would look like:

```
SYSUSERPERM grantor KEY JOIN SYSTABLEPERM  
KEY JOIN SYSUSERPERM grantee
```

**INNER and OUTER** All joins described thus far have been INNER JOINS, which is the default. Each row of

```
Customer INNER JOIN Invoice
```

contains the information from one Customer row and one Invoice row. If a particular customer has no invoices, there will be no information for that customer.

```
A LEFT OUTER JOIN B
```

includes all rows of table A whether or not there is a row in B that satisfies the join condition. If a particular row in A has no matching row in B, the columns in the join corresponding to table B will contain the NULL value. Similarly,

```
A RIGHT OUTER JOIN B
```

includes all rows of table B whether or not there is a row in A that satisfies the join condition.

**Join conditions** A **join-condition** can be specified for any join type except CROSS JOIN. The simplest form of join condition is to use it instead of using a KEY or NATURAL JOIN. In the sample database, the following are equivalent.



```

FROM Department JOIN Employee ON
    Department.dept_id = Employee.dept_id

FROM Department NATURAL JOIN Employee

FROM Department KEY JOIN Employee

```

With INNER JOINS, specifying a join condition is equivalent to adding the join condition to the WHERE clause. However, this is not true for OUTER JOINS.

A join condition on an OUTER JOIN is part of the join operation. For example, the statement:

```

SELECT *
FROM Employee KEY LEFT OUTER JOIN Skill
    ON skill_name = 'COBOL'

```

produces a table containing all employees whether or not they have the skill COBOL. Those who do not have the skill COBOL will have the NULL value in the Skill columns. On the other hand, the query:

```

SELECT *
FROM Employee KEY LEFT OUTER JOIN Skill
    WHERE skill_name = 'COBOL'

```

can be thought of as two separate stages.

- 1 The first stage creates the table specified by the FROM clause:

```

SELECT *
FROM Employee KEY LEFT OUTER JOIN Skill

```

which has at least one row for each employee. Employees who do not have the skill COBOL will have the NULL value in the Skill table columns. For all other employees, there will be one row for each of their skills.

- 2 The second stage takes this result and applies the condition:

```

WHERE skill_name = 'COBOL'

```

which removes employees without any skills (since the condition is UNKNOWN), and only keeps the skill COBOL for the other employees. Thus, the result has no rows with the NULL value in the Skill columns, so it is clearly different from the result achieved using the join condition.

OUTER JOINS with join conditions can be complicated. If you are having problems using a join condition, try removing the WHERE clause from the statement to verify that the join is retrieving the rows you expected.

**Join abbreviations** Watcom SQL provides the following abbreviations for joins.

**table1 [INNER | LEFT OUTER | RIGHT OUTER] JOIN table2**

If there is no ON **join condition** specified, the join is assumed to be a KEY join. All key joins are a Watcom SQL extension. Note that KEY JOIN is *not* assumed if a join condition is specified.

**table1 JOIN table2 ON join-condition**

The default join type is an INNER JOIN.

**Examples**

```
FROM Employee

FROM Employee NATURAL JOIN Skill

FROM Employee KEY JOIN
  (Expertise KEY JOIN Skill, Department, Office)

FROM Employee KEY LEFT OUTER JOIN Skill
  ON skill_name = 'Basic'
```



For Format 3, must have:

- ◆ Created the table, or
- ◆ Been granted permissions on the table with GRANT OPTION, or
- ◆ DBA authority

For Format 4, must have:

- ◆ Created the procedure, or
- ◆ DBA authority

### Side effects

Automatic commit.

### See also

REVOKE.

### Description

The GRANT command is used to grant database permissions to individual user IDs and groups. It is also used to create and delete users and groups.

Formats 1 and 2 of the GRANT command are used for granting special privileges to users as follows:

#### CONNECT

Creates a new user. GRANT CONNECT can also be used by any user to change their own password. To create a user with the empty string as the password, type:

```
GRANT CONNECT TO userid IDENTIFIED BY ""
```

To create a user with no password, type:

```
GRANT CONNECT TO userid
```

A user with no password cannot connect to the database. This is useful when creating groups when you do not want anyone to connect to the group user ID.

#### DBA

Database Administrator authority gives a user permission to do anything. This is usually reserved for the person in the organization who is looking after the database.

#### RESOURCE

Allows the user to create tables and views.

## **GROUP**

Allows the user(s) to have members. See "User groups" on page 140 for a complete description.

## **MEMBERSHIP IN GROUP userid,...**

This allows the user(s) to inherit table permissions from a group and to reference tables created by the group without qualifying the table name. See "User groups" on page 140 for a complete description.

Format 3 of the GRANT command is used to grant permission on individual tables or views. The table permissions can be listed together, or specifying ALL grants all six permissions at once. The permissions have the following meaning:

**ALTER** The users will be allowed to alter this table with the ALTER TABLE command. This permission is not allowed for views.

**DELETE** The users will be allowed to delete rows from this table or view.

**INSERT** The users will be allowed to insert rows into the named table or view.

**REFERENCES** The users will be allowed to create indexes on the named tables, and foreign keys which reference the named tables.

**SELECT** The users will be allowed to look at information in this view or table.

**UPDATE [(column-name,...)]** The users will be allowed to update rows in this view or table. If column names are specified, then the users will be allowed to update only those columns.

**ALL** All of the above permissions.

INDEX and ALTER permissions are not allowed on views. If columns are specified for an UPDATE permission, then UPDATE permission will be given on only those columns.

If WITH GRANT OPTION is specified, then the named user ID is also given permission to GRANT the same permissions to other user IDs.

Format 4 of the GRANT command is used to grant permission to execute a procedure.

### Examples

1. GRANT CONNECT to Laurel, Hardy IDENTIFIED BY Stan, Ollie
2. GRANT SELECT, UPDATE ( address ) ON student TO Laurel
3. GRANT EXECUTE on Calculate\_Report TO Hardy

## HELP command

<b>Syntax</b>	HELP [topic]
<b>Purpose</b>	To receive help in the ISQL environment.
<b>Usage</b>	ISQL.
<b>Authorization</b>	None.
<b>Side effects</b>	None.
<b>Description</b>	The HELP command is used to enter the ISQL interactive help facility. The <b>topic</b> for help can be optionally specified. If <b>topic</b> is not specified, then the help system is entered at the index.

## IF statement

<b>Syntax</b>	IF search-condition THEN statement-list ... [ ELSEIF search-condition THEN statement-list ] ... ... [ ELSE statement-list ] ... END IF
<b>Purpose</b>	Provide conditional execution of SQL statements.
<b>Usage</b>	Procedures and triggers only.
<b>Authorization</b>	None.
<b>Side effects</b>	None.
<b>See also</b>	"Procedures and Triggers" on page 147, Compound statements.
<b>Description</b>	The IF statement is a control statement that allows you to conditionally execute the first list of SQL statements whose <b>search-condition</b> evaluates to TRUE. If no <b>search-condition</b> evaluates to TRUE, and an ELSE clause exists, the <b>statement-list</b> in the ELSE clause is executed. Execution resumes at the first statement after the END IF.

### Example

```
IF mark >= 95 THEN
    set comments = 'Superb'
ELSEIF mark > 80 THEN
    set comments = 'Well Done'
ELSEIF mark > 65 THEN
    set comments = 'Good effort'
ELSEIF mark > 50 THEN
    set comments = 'Keep trying'
ELSE
    set comments = 'See you next year'
END IF
```



# INPUT

<b>Syntax</b>	<pre>INPUT INTO [creator.]table-name     ... [ FROM filename   PROMPT ]     ... [ FORMAT input-format ]     ... [ BY ORDER   BY NAME ]     ... [ DELIMITED BY string ]     ... [ COLUMN WIDTHS (integer,...) ]     ... [ NOSTRIP ]     ... [ ( column-name, ... ) ]</pre>
<b>Purpose</b>	To import data into a database table from an external file or from the keyboard.
<b>Usage</b>	ISQL.
<b>Authorization</b>	Must have INSERT permission on the table or view.
<b>Side effects</b>	None.
<b>See also</b>	OUTPUT, INSERT, UPDATE, DELETE, SET OPTION.
<b>Description</b>	<p>The INPUT command allows efficient mass insertion into a database table. Lines of input are read either from the user via an input window (if PROMPT is specified) or from a file (if FROM filename is specified). If neither is specified, the input will be read from the command file containing the input command—this can even be directly from the ISQL editor. In this case, input is ended with a line containing only the string END.</p> <p>These lines are inserted into the named table. If a column list is specified, the data is inserted into the specified columns of the named table.</p> <p>Normally, the INPUT command stops when it attempts to insert a row that causes an error. Errors can be treated in different ways by setting the ON_ERROR and CONVERSION_ERROR options (see "SET OPTION" on page 283). ISQL will print a warning in the statistics window if any string values are truncated on INPUT. Missing values for NOT NULL columns</p>

will be set to zero for numeric types and to the empty string for non-numeric types.

The BY clause allows the user to specify whether the columns from the input file should be matched up with the table columns based on their ordinal position in the lists (ORDER, the default) or by their names (NAME). Not all input formats have column name information in the file. NAME is allowed only for those formats that do. They are the same formats that allow automatic table creation listed below: DBASEII, DBASEIII, DIF, FOXPRO, LOTUS, and WATFILE.

The DELIMITED BY clause allows you to specify a string to be used as the delimiter in ASCII input format.

COLUMN WIDTHS can be specified for FIXED format only. It specifies the widths of the columns in the input file. If COLUMN WIDTHS is not specified, then the widths are determined by the database column types.

Normally, for ASCII input format, trailing blanks will be stripped before the value is inserted. NOSTRIP can be used to suppress trailing blank stripping.

Each set of values must occupy one input line and must be in the format specified by the FORMAT clause or the format set by the SET INPUT\_FORMAT command if the FORMAT clause is not specified. When input is entered by the user, an empty screen is provided for the user to enter one row per line in the input format.

Certain file formats contain information about column names and types. Using this information, the INPUT command will create the database table if it does not already exist. This is a very easy way to load data into the database. The formats that have enough information to create the table are: DBASEII, DBASEIII, DIF, FOXPRO, LOTUS, and WATFILE.

Allowable input formats are:

**ASCII** Input lines are assumed to be ASCII characters, one row per line, with values separated by commas. Alphabetic strings may be enclosed in apostrophes (single quotes) or quotation marks (double quotes). Strings containing commas must be enclosed in either single or double quotes. If single or double quotes are used, double the quote character to use it within the string. Optionally, you can use the DELIMITED BY clause to specify a different delimiter string than the default which is a comma.

Three other special sequences are also recognized. The two characters \n represent a newline character, \\ represents a single

\, and the sequence \xDD represents the character with hexadecimal code DD.

- DBASE** The file is in dBASE II or dBASE III format. ISQL will attempt to determine which of the two dBase formats the file is based on information in the file. If the table doesn't exist, it will be created.
- DBASEII** The file is in dBASE II format. If the table doesn't exist, it will be created.
- DBASEIII** The file is in dBASE III format. If the table doesn't exist, it will be created.
- DIF** Input file is in Data Interchange Format. If the table doesn't exist, it will be created.
- FIXED** Input lines are in fixed format. The width of the columns can be specified using the COLUMN WIDTHS clause. If they are not specified, then column widths in the file must be the same as the maximum number of characters required by any value of the corresponding database column's type.
- FOXPRO** The file is in FoxPro format (the FoxPro memo field is different than the dBASE memo field). If the table doesn't exist, it will be created.
- LOTUS** The file is a Lotus WKS format worksheet. INPUT assumes that the first row in the Lotus WKS format worksheet is column names. If the table doesn't exist, it will be created. In this case, the types and sizes of the columns created may not be correct because the information in the file pertains to a cell, not to a column.
- WATFILE** The input will be a WATFILE file. If the table doesn't exist, it will be created. WATFILE is a tabular file management tool available from WATCOM.

Input from a command file is terminated by a line containing END. Input from a file is terminated at the end of the file.

## *INPUT*

---

### **Example**

```
1. INPUT INTO student FROM newstud.inp FORMAT ascii;
2. INPUT INTO register (studnum, course, section);
   86004, 'CALC', 1
   86004, 'ALG', 1
END
```

# INSERT

**Syntax**

Format 1

```
INSERT INTO [creator.]table-name [( column-name, ... )]  
    ... VALUES ( expression | DEFAULT, ... )
```

Format 2

```
INSERT INTO [creator.]table-name [( column-name, ... )]  
    ... select-statement
```

**Purpose**

To insert a single row (format 1) or a selection of rows from elsewhere in the database (format 2) into a table.

**Usage**

Anywhere.

**Authorization**

Must have INSERT permission on the table.

**Side effects**

None.

**See also**

INPUT, UPDATE, DELETE.

**Description**

The INSERT command is used to add new rows to a database table.

Format 1 allows the insertion of a single row with the specified expression values. The keyword DEFAULT can be used to cause the default value for the column to be inserted. If the optional list of column names is given, the values are inserted one for one into the specified columns. If the list of column names is not specified, the values are inserted into the table columns in the order they were created (the same order as retrieved with SELECT \*). The row is inserted into the table at an arbitrary position. (In relational databases, tables are not ordered.)

Format 2 allows the user to do mass insertion into a table with the results of a fully general SELECT statement. Insertions are done in an arbitrary order unless the SELECT statement contains an ORDER BY clause. The columns from the select list are matched ordinally with the columns specified in the column list or the columns in the order they were created.

**Note**

The NUMBER(\*) function is very useful for generating primary keys with format 2 of the insert command (see "Functions" on page 71).

Inserts can be done into views provided the SELECT command defining the view has only one table in the FROM clause and does not contain a GROUP BY clause, an aggregate function, or involve a UNION operation.

**Examples**

1. `INSERT INTO skill VALUES ('86004', 'COBOL', 1)`
2. `INSERT INTO female_emp (name, initials)  
SELECT emp_lname, initials  
FROM employee  
WHERE sex = 'f'`

# LEAVE

<b>Syntax</b>	LEAVE statement-label
<b>Purpose</b>	Continue execution by leaving a compound statement or LOOP.
<b>Usage</b>	Procedures and triggers only.
<b>Authorization</b>	None.
<b>Side effects</b>	None.
<b>See also</b>	"Procedures and Triggers" on page 147, Compound statements, LOOP, FOR.

**Description** The LEAVE statement is a control statement that allows you to leave a labelled compound statement or a labelled loop. Execution resumes at the first statement after the compound statement or loop.

The compound statement that is the body of a procedure or triggers has an implicit label that is the same as the name of the procedure or trigger.

## Examples

```

1. SET i = 1;
   lbl:
     LOOP
       INSERT INTO Counters( number ) VALUES ( i );
       IF i >= 10 THEN
         LEAVE lbl
       END IF
       SET i = i + 1;
     END LOOP lbl

2. outer_loop:
   LOOP
     SET i = 1;
     inner_loop:
       LOOP
         . . .
         SET i = i + 1;
         IF i >= 10 THEN
           LEAVE outer_loop
         END IF
       END LOOP inner_loop
     END LOOP outer_loop

```

# LOOP

**Syntax**                    [ statement-label : ]  
                              ... [ WHILE search-condition ] LOOP  
                              ...     statement-list  
                              ... .END LOOP [ statement-label ]

**Purpose**                    Repeat the execution of a statement list.

**Usage**                    Procedures and triggers only.

**Authorization**         None.

**Side effects**            None.

**See also**                FOR, LEAVE.

**Description**            The WHILE and LOOP statements are control statements that allow you to repeatedly execute a list of SQL statements while a **search-condition** evaluates to TRUE. The LEAVE statement can be used to resume execution at the first statement after the END LOOP.

If the ending **statement-label** is specified, it must match the beginning **statement-label**.

## Examples

```
1.  SET i = 1;
    WHILE i <= 10 LOOP
        INSERT INTO Counters( number ) VALUES ( i );
        SET i = i + 1;
    END LOOP

2.  SET i = 1;
    lbl:
    LOOP
        INSERT INTO Counters( number ) VALUES ( i );
        IF i >= 10 THEN
            LEAVE lbl
        END IF
        SET i = i + 1;
    END LOOP lbl
```



## NULL value

<b>Syntax</b>	NULL
<b>Purpose</b>	To specify a value that is unknown or not applicable.
<b>Usage</b>	Anywhere.
<b>Authorization</b>	Must be connected to the database.
<b>Side effects</b>	None.
<b>See also</b>	Expressions, Conditions.

### Description

The NULL value is a special value which is different from any valid value for any data type. However, the NULL value is a legal value in any data type. The NULL value is used to represent missing or inapplicable information. Note that these are two separate and distinct cases where NULL is used:

Situation	Description
missing	The field does have a value, but that value is unknown.
inapplicable	The field does not apply for this particular row.

SQL allows columns to be created with the NOT NULL restriction. This means that those particular columns cannot contain the NULL value.

The NULL value introduces the concept of three valued logic to SQL. The NULL value compared using any comparison operator with any value including the NULL value is "UNKNOWN." The only search condition that will return "TRUE" is the IS NULL predicate. In SQL, rows are selected only if the search condition in the WHERE clause evaluates to TRUE; rows that evaluate to UNKNOWN or FALSE are *not* selected. This IS [NOT] <truth-value> clause where truth-value is one of TRUE, FALSE or UNKNOWN can also be used to select rows where the NULL value is involved. See "Conditions" on page 90 for a description of this clause.

In the following examples, the column Salary contains the NULL value.

Condition	Truth value	Selected?
Salary = NULL	UNKNOWN	NO
Salary <> NULL	UNKNOWN	NO
NOT (Salary = NULL)	UNKNOWN	NO
NOT (Salary <> NULL)	UNKNOWN	NO
Salary = 1000	UNKNOWN	NO
Salary IS NULL	TRUE	YES
Salary IS NOT NULL	FALSE	NO
Salary = 1000 IS UNKNOWN	TRUE	YES

The same rules apply when comparing columns from two different tables. Therefore, joining two tables together will not select rows where any of the columns compared contain the NULL value.

The NULL value also has an interesting property when used in numeric expressions. The result of *any* numeric expression involving the NULL value is the NULL value. This means that if the NULL value is added to a number, the result is the NULL value—not a number. If you want the NULL value to be treated as 0, you must use the **isnull( <expression>, 0 )** function (see "Functions" on page 71).

Many common errors in formulating SQL queries are caused by the behavior of NULL. You will have to be careful to avoid these problem areas. See "Conditions" on page 90 for a description of the effect of three-valued logic when combining search conditions.

### Example

```
INSERT INTO Borrowed_book
    (date_borrowed,date_returned,book)
VALUES (CURRENT DATE, NULL, '1234')
```

# OPEN

<b>Syntax</b>	<pre>OPEN cursor-name       . . . [ WITH HOLD ] [ ISOLATION LEVEL n ]</pre> <p>cursor-name:            identifier</p>
<b>Purpose</b>	To open a previously declared cursor to access information from the database.
<b>Usage</b>	Procedures and triggers only.
<b>Authorization</b>	<p>Must have <b>SELECT</b> permission on all tables in a <b>SELECT</b> statement or <b>EXECUTE</b> permission on the procedure in a <b>CALL</b> statement.</p> <p>When the cursor is on a <b>CALL</b> statement, <b>OPEN</b> causes the procedure to execute until the first result set (<b>SELECT</b> statement with no <b>INTO</b> clause) is encountered. If the procedure completes and no result set is found, the <b>SQLSTATE_PROCEDURE_COMPLETE</b> warning is set.</p>
<b>Side effects</b>	None.
<b>See also</b>	<b>OPEN</b> in <i>PowerScript Language</i> , <b>DECLARE CURSOR</b> , <b>FETCH</b> , <b>RESUME</b> , <b>CLOSE</b> , "Cursors in procedures and triggers" on page 167.
<b>Description</b>	<p>The <b>OPEN</b> statement opens the named cursor. The cursor must be previously declared.</p> <p>By default, all cursors are automatically closed at the end of the current transaction (<b>COMMIT</b> or <b>ROLLBACK</b> executed). The optional <b>WITH HOLD</b> clause will keep the cursor open for subsequent transactions. It will remain open until the end of the current connection or until an explicit <b>CLOSE</b> statement is executed. Cursors are automatically closed when a connection is terminated.</p> <p>The <b>ISOLATION LEVEL</b> clause allows this cursor to be opened at an isolation level different from the current setting of the <b>ISOLATION_LEVEL</b> option. All operations on this cursor will be performed at the specified isolation level regardless of the option setting. If this clause is not specified, then the cursor's isolation level for the entire time the cursor is open is the</p>

value of the ISOLATION\_LEVEL option when the cursor is opened. See "Locking and Concurrency" on page 103.

The cursor is positioned before the first row (see "Cursors in procedures and triggers" on page 167).

### Procedure/trigger example

```
BEGIN
  DECLARE student_cursor CURSOR FOR
    SELECT surname FROM student;
  DECLARE name CHAR(40);

  OPEN student_cursor;
  LOOP
    FETCH NEXT student_cursor into name;
    ...
  ENDLOOP
  CLOSE student_cursor;
END
```

# OUTPUT

## Syntax

```
OUTPUT TO filename
    ... [ FORMAT output_format ]
    ... [ DELIMITED BY string ]
    ... [ QUOTE string [ ALL ] ]
    ... [ COLUMN WIDTHS (integer,...) ]
```

## Purpose

To output the current query results to a file.

## Usage

ISQL.

## Authorization

None.

## Side effects

The current query results that are displayed in the Data window are repositioned to the top.

## See also

SELECT.

## Description

The **OUTPUT** command copies the information retrieved by the **current query to filename**. The output format can be specified with the optional **FORMAT** clause. If no **FORMAT** clause is specified, the **OUTPUT\_FORMAT** option setting is used (see "SET OPTION" on page 283).

The **current query** is the **SELECT** or **INPUT** command which generated the information displayed in the Data window. The **OUTPUT** command will report an error if there is no current query.

The **DELIMITED BY** and **QUOTE** clauses are for the ASCII output format only. The delimiter string will be placed between columns (default comma) and the quote string will be placed around string values (default '—single quote). If **ALL** is specified in the **QUOTE** clause, then the quote string will be placed around all values, not just strings.

The **COLUMN WIDTH** clause is used to specify the column widths for the **FIXED** format output.

Allowable output formats are:

- ASCII** The output is an ASCII format file with one row per line in the file. All values are separated by commas and strings are enclosed in apostrophes (single quotes). The delimiter and quote strings can be changed using the **DELIMITED BY** and **QUOTE** clauses. If **ALL** is specified in the **QUOTE** clause, then all values (not just strings) will be quoted.
- Three other special sequences are also used. The two characters `\n` represent a newline character, `\\` represents a single `\`, and the sequence `\xDD` represents the character with hexadecimal code `DD`. This is the default output format.
- DBASEII** The output is a dBASE II format file with the column definitions at the top of the file. Note that a maximum of 32 columns can be output. Also, note that columns longer than 255 characters will be truncated in the file.
- DBASEIII** The output is a dBASE III format file with the column definitions at the top of the file. Note that a maximum of 128 columns can be output. Also, note that columns longer than 255 characters will be truncated in the file.
- DIF** The output is a file in the standard Data Interchange Format.
- FIXED** The output is fixed format with each column having a fixed width. The width for each column can be specified using the **COLUMN WIDTH** clause. If this clause is omitted, the width for each column is computed from the data type for the column, and is large enough to hold any value of that data type. No column headings are output in this format.
- FOXPRO** The output is a FoxPro format file (the FoxPro memo field is different than the dBASE memo field) with the column definitions at the top of the file. Note that a maximum of 128 columns can be output. Also, note that columns longer than 255 characters will be truncated in the file.
- LOTUS** The output is a Lotus WKS format worksheet. Column names will be put as the first row in the worksheet. Note that there are certain restrictions on the maximum size of Lotus WKS format worksheets that other software (such as Lotus 1-2-3) can load. There is no limit to the size of file ISQL can produce.

**SQL** The output is an ISQL INPUT command required to recreate the information in the table.

**TEXT** The output is a TEXT format file which prints the results in columns with the column names at the top and vertical lines separating the columns. This format is similar to that used to display data in the ISQL data window.

**WATFILE** The output is a WATFILE format file with the column definitions at the top of the file. WATFILE is a tabular file management tool available from WATCOM.

**Example**

```
OUTPUT TO student.asc FORMAT ASCII
```

# PARAMETERS

<b>Syntax</b>	PARAMETERS parameter1, parameter2, ...
<b>Purpose</b>	To specify parameters to an ISQL command file.
<b>Usage</b>	ISQL.
<b>Authorization</b>	None.
<b>Side effects</b>	None.
<b>See also</b>	READ.
<b>Description</b>	The PARAMETERS command specifies how many parameters there are to a command file and also gives names to those parameters so that they can be referenced later in the command file.

Parameters are referenced by putting:

```
{parameter1}
```

into the file where you wish the named parameter to be substituted.

If a command file is invoked with fewer than the required number of parameters, ISQL will automatically prompt for values for the missing parameters.

## Example

```
PARAMETERS studnum, sex;
...
SELECT surname
FROM admin.student
WHERE studnum = {studnum} AND
      sex = '{sex}'
>#males.dat;
```



# PREPARE TO COMMIT

<b>Syntax</b>	PREPARE TO COMMIT
<b>Purpose</b>	To check whether a COMMIT can be performed.
<b>Usage</b>	Anywhere.
<b>Authorization</b>	None.
<b>Side effects</b>	None.
<b>See also</b>	COMMIT, ROLLBACK.
<b>Description</b>	The PREPARE TO COMMIT statement tests whether a COMMIT can be performed successfully. The statement will cause an error if a COMMIT is not possible without violating the integrity of the database.

## Example

```
EXECUTE IMMEDIATE "SET wait_for_commit = on";
EXECUTE IMMEDIATE "DELETE FROM Student WHERE Studnum =
'86004'";
EXECUTE IMMEDIATE "PREPARE TO COMMIT";
```

# READ

**Syntax** READ filename [ parameters ]

**Purpose** To read ISQL commands from a file.

**Usage** ISQL.

**Authorization** None.

**Side effects** None.

**See also** PARAMETERS.

**Description** The READ command reads a sequence of ISQL commands from the named file. This file can contain any valid ISQL command including other READ commands. READ commands can be nested to any depth. To find the command file, ISQL will first search the current directory, then the directories specified in the environment variable **SQLPATH**, then the directories specified in the environment variable **PATH**. If the named file has no file extension, ISQL will also search each directory for the same file name with the extension **.sql**.

Parameters can be listed after the name of the command file. These parameters correspond to the parameters named on the **PARAMETERS** command at the beginning of the command file (see "PARAMETERS" on page 270). ISQL will then substitute the corresponding parameter wherever the source file contains

```
{parameter-name}
```

where **parameter-name** is the name of the appropriate parameter.

The parameters passed to a command file can be identifiers, numbers, quoted identifiers, or strings. When quotes are used around a parameter, the quotes are put into the text during the substitution. Parameters which are not identifiers, numbers, or strings (contain spaces or tabs) must be enclosed in square brackets ( []). This allows for arbitrary textual substitution in the command file.

If not enough parameters are passed to the command file, ISQL will automatically prompt for values for the missing parameters.

## Examples

1. `READ repcards.rpt '86004'`
2. `READ birthday.sql [ >= '1988-1-1' ] [ <= '1988-1-30' ]`

## RELEASE SAVEPOINT

<b>Syntax</b>	RELEASE SAVEPOINT [savepoint-name]
<b>Purpose</b>	Release a savepoint within the current transaction.
<b>Usage</b>	Anywhere.
<b>Authorization</b>	There must have been a corresponding SAVEPOINT within the current transaction.
<b>Side effects</b>	None.
<b>See also</b>	SAVEPOINT, ROLLBACK TO SAVEPOINT.
<b>Description</b>	<p>Release a savepoint. The <b>savepoint-name</b> is an identifier that was specified on a SAVEPOINT command within the current transaction. If <b>savepoint-name</b> is omitted, the most recent savepoint is released.</p> <p>For a description of savepoints, see "Savepoints" on page 58. Releasing a savepoint does not do any type of COMMIT. It simply removes the savepoint from the list of currently active savepoints.</p>

# RESIGNAL

<b>Syntax</b>	RESIGNAL
<b>Purpose</b>	Resignal an exception condition.
<b>Usage</b>	Within an exception handler in a procedure or trigger.
<b>Authorization</b>	None.
<b>Side effects</b>	None.
<b>See also</b>	"Errors in procedures and triggers" on page 162, Compound statements, SIGNAL.
<b>Description</b>	Within an exception handler, RESIGNAL allows you to quit the compound statement with the exception still active. The exception will be handled by another exception handler or returned to the application. Any actions by the exception handler before the RESIGNAL are not undone.

## Example

```
DECLARE COLUMN_NOT_FOUND EXCEPTION FOR SQLSTATE '52003';
. . .
EXCEPTION
    WHEN COLUMN_NOT_FOUND THEN
        set message='column not found'
    WHEN OTHERS THEN RESIGNAL
```

## RESUME

**Syntax** RESUME cursor-name

cursor-name: identifier

**Purpose** To close a cursor.

**Usage** Procedures and triggers only.

**Authorization** The cursor must have been previously opened.

**Side effects** None.

**See also** "Result sets from procedures" on page 170, DECLARE CURSOR.

**Description** This statement resumes execution of a procedure that returns result sets. The procedure will execute until the next result set (SELECT statement with no INTO clause) is encountered. If the procedure completes and no result set is found, the SQLSTATE\_PROCEDURE\_COMPLETE warning is set. This warning is also set when you RESUME a cursor for a SELECT statement.

### Example

```
RESUME employee_cursor;
```

# REVOKE

## Syntax

### Format 1

```

REVOKE | CONNECT
      | DBA
      | RESOURCE
      | GROUP
      | MEMBERSHIP IN GROUP userid,...

```

... FROM userid,...

### Format 2

```

REVOKE | ALTER
      | DELETE
      | INSERT
      | REFERENCE
      | SELECT
      | UPDATE [(column-name,...)]
      | ALL [PRIVILEGES]

```

... ON [creator.]table-name FROM userid,...

### Format 3

```

REVOKE EXECUTE ON [creator.]procedure-name FROM userid,...

```

## Purpose

To remove permissions for specified user(s).

## Usage

DBA notepad.

## Authorization

Must be the grantor of the permissions that are being revoked or must have DBA authority.

## Side effects

Automatic commit.

## See also

GRANT.

## Description

The REVOKE command is used to remove permissions that were given using the GRANT command. Format 1 is used to revoke special user permissions and format 2 is used to revoke table permissions. Format 3 is used to revoke permission to execute a procedure. REVOKE CONNECT is

## *REVOKE*

---

used to remove a user ID from a database. REVOKE GROUP will automatically REVOKE MEMBERSHIP from all members of the group.

### **Examples**

1. REVOKE UPDATE ON student FROM Dave
2. REVOKE RESOURCE FROM Jim
3. REVOKE EXECUTE ON Calculate\_Report FROM Hardy



# ROLLBACK

<b>Syntax</b>	ROLLBACK WORK
<b>Purpose</b>	To undo any changes made since the last COMMIT or ROLLBACK.
<b>Usage</b>	Anywhere.
<b>Authorization</b>	Must be connected to the database.
<b>Side effects</b>	Closes all cursors not opened WITH HOLD.
<b>See also</b>	COMMIT, ROLLBACK TO SAVEPOINT.
<b>Description</b>	The ROLLBACK command ends a logical unit of work (transaction) and undoes all changes made to the database during this transaction. A transaction is defined as the database work done between COMMIT or ROLLBACK commands on one database connection.

## ROLLBACK TO SAVEPOINT

<b>Syntax</b>	ROLLBACK TO SAVEPOINT [savepoint-name]
<b>Purpose</b>	To cancel any changes made since a SAVEPOINT.
<b>Usage</b>	Anywhere.
<b>Authorization</b>	There must have been a corresponding SAVEPOINT within the current transaction.
<b>Side effects</b>	None.
<b>See also</b>	SAVEPOINT, RELEASE SAVEPOINT, ROLLBACK.
<b>Description</b>	<p>The ROLLBACK TO SAVEPOINT command will undo any changes that have been made since the SAVEPOINT was established. Changes made prior to the SAVEPOINT are not undone; they are still pending. For a description of savepoints, see "Savepoints" on page 58.</p> <p>The <b>savepoint-name</b> is an identifier that was specified on a SAVEPOINT command within the current transaction. If <b>savepoint-name</b> is omitted, the most recent savepoint is used. Any savepoints since the named savepoint are automatically released.</p>

# SAVEPOINT

<b>Syntax</b>	SAVEPOINT [savepoint-name]
<b>Purpose</b>	To establish a savepoint within the current transaction.
<b>Usage</b>	Anywhere.
<b>Authorization</b>	None.
<b>Side effects</b>	None.
<b>See also</b>	RELEASE SAVEPOINT, ROLLBACK TO SAVEPOINT.
<b>Description</b>	<p>Establish a savepoint within the current transaction. The <b>savepoint-name</b> is an identifier that can be used in an <b>RELEASE SAVEPOINT</b> or <b>ROLLBACK TO SAVEPOINT</b> command. All savepoints are automatically released when a transaction ends. See "Savepoints" on page 58.</p> <p>Savepoints that are established while a trigger is executing or while an atomic compound statement is executing are automatically released when the atomic operation ends.</p>

# SET CONNECTION

<b>Syntax</b>	SET CONNECTION [connection-name]  connection-name:    identifier, string or host-variable
<b>Purpose</b>	To change the active database connection.
<b>Usage</b>	ISQL.
<b>Authorization</b>	None.
<b>Side effects</b>	None.
<b>See also</b>	CONNECT, DISCONNECT.
<b>Description</b>	The SET CONNECTION command changes the active database connection to <b>connection-name</b> . The current connections state is saved and will be resumed when it again becomes the active connection. If <b>connection-name</b> is omitted and there is a connection that was not named, that connection becomes the active connection.

## ISQL example

```
SET CONNECTION conn1
```

# SET OPTION

## Syntax

1. SET [ TEMPORARY ] OPTION  
     ... [ userid. | PUBLIC. ]option-name = [ option-value ]
2. SET PERMANENT
3. SET

userid:                    identifier, string or host-variable  
 option-name:            identifier, string or host-variable  
 option-value:            host-variable (indicator allowed), string, identifier, or number

Note: Formats 2 and 3 are ISQL only.

## Purpose

To change database options.

## Usage

All.

## Authorization

None required to set your own options. Must have DBA authority to set options for another user or PUBLIC.

## Side effects

If TEMPORARY is not specified, an automatic commit is performed.

## See also

DBSTART.

## Description

The SET command is used to change options for both the database and the ISQL environment. The **userid** parameter can be specified to change someone else's options, however, you must have DBA authority to do so. Specifying TEMPORARY will change the option setting for the current connection only. In this case, **userid** cannot be specified.

Changing a PUBLIC option sets the option value that will be used for any userid which has not SET their own value for that option. An option cannot be set for a user ID other than PUBLIC unless there is already a PUBLIC setting for that option. Only a user with DBA authority is allowed to set options for another user ID (including PUBLIC).

If the **option-value** is omitted, the specified option setting will be deleted from the database. If it was a personal option setting, then the value used

will revert back to the PUBLIC setting. If a TEMPORARY option is deleted, the option setting will revert back to the permanent setting.

### **Temporary setting**

Only your own ISQL and database options can be made TEMPORARY; options for other software and any options set for another user will produce an error if TEMPORARY is specified.

SET PERMANENT (format 2) stores all current ISQL options in the **SYS.SYSOPTIONS** table in the database. These settings will automatically be established every time ISQL is started for the current userid.

Format 3 is used to display all of the current option settings. If there are temporary options set for ISQL or the database engine, these will be displayed; otherwise, the permanent option settings are displayed.

The options are divided into two groups: **Database options** and **ISQL options**.

## Database options

OPTION	VALUES	DEFAULT
BLOCKING	ON,OFF	ON
CHECKPOINT_TIME	number of minutes	60
COMMAND_DELIMITER	string	','
CONVERSION_ERROR	ON,OFF	ON
DATE_FORMAT	string	'YYYY-MM-DD'
DATE_ORDER	'YMD','DMY','MDY'	'MM/DD/YYYY' (1)
		'YMD'
		'MDY' (1)
ISOLATION_LEVEL	0,1,2,3	0
PRECISION	number of digits	30
RECOVERY_TIME	number of minutes	2
ROW_COUNTS	ON,OFF	OFF
SCALE	number of digits	6
THREAD_COUNT	number of threads	0
TIME_FORMAT	string	'HH:NN:SS.SSS'
TIMESTAMP_FORMAT	string	'YYYY-MM-DD HH:NN:SS.SSS'
WAIT_FOR_COMMIT	ON,OFF	OFF

\$(1) Default for databases created with Watcom SQL Version 3.0.

### BLOCKING

Controls whether locking conflicts result in one user becoming blocked or an error condition being returned. The default is ON. See "Transaction blocking" on page 107.

### CHECKPOINT\_TIME

Set the maximum desired length of time that the database engine will run without doing a checkpoint. The time is specified in minutes (the default value set by DBINIT is 60). This option is used with the RECOVERY\_TIME option to decide when checkpoints should be done. See "Checkpoint log" on page 117.

Because this is a global option for the database, only the PUBLIC setting is used. Individual settings for users will have no effect. Also, changing this option does not take effect immediately. You must shut down the database engine and restart for the change to take effect.

### COMMAND\_DELIMITER

Sets the string indicating the termination of a command in ISQL. The default setting is a single semi-colon ';'. This must be changed in order to create triggers and procedures from ISQL using the CREATE TRIGGER or CREATE PROCEDURE statement. The change prevents the semi-colons terminating individual statements within the trigger or procedure from being interpreted as the termination of the CREATE statement itself.

A common setting for this option is '\'.

If the command delimiter is set to a string beginning with a character that is valid in identifiers, the command delimiter must be preceded by a space.

### CONVERSION\_ERROR

Controls whether conversion errors will be reported by the database, or ignored. If conversion errors are ignored, the NULL value is used in place of the value that could not be converted.

### DATE\_FORMAT

Sets the format used for dates retrieved from the database. The format is a string using the following symbols:



Symbol	Description
yy	Two digit year
yyyy	Four digit year
mm	Two digit month, or two digit minutes if following a colon (as in 'hh:mm')
mmm[m...]	Character short form for months—as many characters as there are m's
dd	Two digit day of month
ddd[d...]	Character short form for day of the week
hh	Two digit hours
nn	Two digit minutes
aa	Am or pm (12 hour clock)
pp	Pm if needed (12 hour clock)
f	Use French days and months

Each symbol will be substituted with the appropriate data for the date being formatted. Any format symbol that represents character rather than digit output can be put in upper case which will cause the substituted characters to also be in upper case. For numbers, using mixed case in the format string will suppress leading zeros.

#### Note

The default date format has changed since Watcom SQL Version 3.0. The new format corresponds to ISO date format specifications ('YYYY-MM-DD'). Databases created with Watcom SQL Version 3.0 will still use the old option value. In addition, the new default date format does not specify hours and minutes. `TIMESTAMP_FORMAT` includes hours, minutes and seconds.

## DATE\_ORDER

The database option `DATE_ORDER` is used to determine whether 10/11/12 is Oct 11 1912, Nov 12 1910, or Nov 10 1912. The option can have the value 'MDY', 'YMD', or 'DMY'.

**Note**

The default date order has changed since Watcom SQL Version 3.0. The new order ('YMD') corresponds to ISO date format specifications. Databases created with Watcom SQL Version 3.0 will still use the old option value which was 'MDY'.

**ISOLATION\_LEVEL**

Controls the locking isolation level as follows.

- ◆ **0** Allow dirty reads, non-repeatable reads, and phantom rows.
- ◆ **1** Prevent dirty reads. Allow non-repeatable reads and phantom rows.
- ◆ **2** Prevent dirty reads and guarantee repeatable reads. Allow phantom rows.
- ◆ **3** Serializable. Do not allow dirty reads, guarantee repeatable reads, and do not allow phantom rows.

The default is 0. See "Locking and Concurrency" on page 103.

**PRECISION**

Specifies the maximum number of digits in the result of any decimal arithmetic. Precision is the total number of digits to the left and right of the decimal point. The SCALE option specifies the minimum number of digits after the decimal point when an arithmetic result is truncated to the maximum PRECISION.

Multiplication, division, addition, subtraction, and aggregate functions can all have results that exceed the maximum precision.

For example, when a DECIMAL(8,2) is multiplied with a DECIMAL(9,2), the result could require a DECIMAL(17,4). If PRECISION is 15, only 15 digits will be kept in the result. If SCALE is 4, the result will be a DECIMAL(15,4). If SCALE is 2, the result will be a DECIMAL(15,2). In both cases, there is a possibility for overflow.

**RECOVERY\_TIME**

Set the maximum desired length of time that the database engine will take to recover from system failure. The time is specified in minutes (the default value set by DBINIT is 2). This option is used with the

CHECKPOINT\_TIME option to decide when checkpoints should be done. See "Recovery from system failure" on page 122.

Watcom SQL uses a heuristic to measure the recovery time based on the operations since the last checkpoint. Thus, the recovery time is not exact.

Because this is a global option for the database, only the PUBLIC setting is used. Individual settings for users will have no effect. Also, changing this option does not take effect immediately. You must shut down the database engine and restart it for the change to take effect.

## ROW\_COUNTS

Specifies whether the database will always count the number of rows in a query when it is opened. If this option is off, the row count will usually only be an estimate. If this option is on, the row count will always be accurate but opening queries will take significantly longer in many cases.

## SCALE

Specifies the minimum number of digits after the decimal point when an arithmetic result is truncated to the maximum PRECISION.

Multiplication, division, addition, subtraction, and aggregate functions can all have results that exceed the maximum precision. See the PRECISION option for an example.

## THREAD\_COUNT

Sets the number of execution threads that will be used in the database engine while running with multiple users. This number controls the number of concurrent requests that the database engine will process. A value of 0 (the default) means an operating system specific value:

<b>16-bit Windows</b>	8
<b>All 32-bit</b>	20

Because this is a global option for the database, only the PUBLIC setting is used. Individual settings for users will have no effect. Also, changing this option does not take effect immediately. You must shut down the database engine and restart for the change to take effect.

In DOS, DBSTART and RTSTART have a limited 64K primary dataspace; increasing the thread count beyond 3 may cause NOCACHE and NOMEMORY errors. All other database engines have no such limitations.

**TIME\_FORMAT**

Sets the format used for times retrieved from the database. The format is a string using the following symbols:

- ◆ **hh** Two digit hours (24 hour clock)
- ◆ **nn** Two digit minutes
- ◆ **mm** Two digit minutes if following a colon (as in 'hh:mm')
- ◆ **ss[.s...]** Two digit seconds plus optional fraction

Each symbol will be substituted with the appropriate data for the date being formatted. Any format symbol that represents character rather than digit output can be put in uppercase which will cause the substituted characters to also be in uppercase. For numbers, using mixed case in the format string will suppress leading zeros.

**TIMESTAMP\_FORMAT**

Sets the format used for timestamps retrieved from the database. The format is a string using the following symbols:

Symbol	Description
yy	Two digit year
yyyy	Four digit year
mm	Two digit month, or two digit minutes if following a colon (as in 'hh:mm')
mmm[m...]	Character short form for months—as many characters as there are m's
dd	Two digit day of month
ddd[d...]	Character short form for day of the week
hh	Two digit hours
nn	Two digit minutes
aa	Am or pm (12 hour clock)
pp	Pm if needed (12 hour clock)
f	Use French days and months

Each symbol will be substituted with the appropriate data for the date being formatted. Any format symbol that represents character rather than digit output can be put in uppercase which will cause the substituted characters to

also be in uppercase. For numbers, using mixed case in the format string will suppress leading zeros.

### **WAIT\_FOR\_COMMIT**

If this option is on, the database will not check foreign key integrity until the next COMMIT command. Otherwise, all foreign keys not created with the CHECK ON COMMIT option are checked as they are inserted, updated or deleted.

ISQL options

OPTION	VALUES	DEFAULT
AUTO_COMMIT	ON,OFF	OFF
AUTO_REFETCH	ON,OFF	ON
BELL	ON,OFF	ON
COMMIT_ON_EXIT	ON,OFF	ON
ECHO	ON,OFF	ON
HEADINGS	ON,OFF	ON
INPUT_FORMAT	ASCII FIXED DIF DBASE DBASEII DBASEIII FOXPRO LOTUS WATFILE	ASCII
ISQL_LOG	file-name	''
NULLS	string	'(NULL)'
ON_ERROR	STOP CONTINUE PROMPT EXIT	PROMPT
OUTPUT_FORMAT	TEXT ASCII FIXED DIF DBASEII DBASEIII FOXPRO LOTUS SQL WATFILE	ASCII
OUTPUT_LENGTH	integer	0
STATISTICS	0,3,4,5,6	3
TRUNCATION_LENGTH	integer	30

## **AUTO\_COMMIT**

If **AUTO\_COMMIT** is 'on', a database **COMMIT** is performed after each successful command and a **ROLLBACK** after each failed command. Otherwise, a **COMMIT** or **ROLLBACK** is only performed when the user issues a **COMMIT** or **ROLLBACK** command or if the user issues a **SQL** command which causes an automatic commit such as the **CREATE TABLE** command.

## **AUTO\_REFETCH**

If **AUTO\_REFETCH** is 'on', then the current query results displayed in the Data window will be refetched from the database after **any** **INSERT**, **UPDATE** or **DELETE** command. Depending on how complicated the query is, this may take some time. For this reason, it can be turned 'off'.

## **BELL**

Controls whether the bell will sound when an error occurs.

## **COMMIT\_ON\_EXIT**

Controls whether a **COMMIT** or **ROLLBACK** is done when leaving **ISQL**.

## **ECHO**

Controls whether commands are echoed before they are executed. This is most useful when using the **READ** command to execute an **ISQL** command file.

## **HEADINGS**

Controls whether headings will be displayed for the results of a **SELECT** command.

## **INPUT\_FORMAT**

Sets the default data format expected by the **INPUT** command.

Certain file formats contain information about column names and types. Using this information, the **INPUT** command will create the database table if it does not already exist. This is a very easy way to load data into the database. The formats that have enough information to create the table are: **DBASEII**, **DBASEIII**, **DIF**, **FOXPRO**, **LOTUS**, and **WATFILE**.

Allowable input formats are:

- ASCII** Input lines are assumed to be ASCII characters, one row per line, with values separated by commas. Alphabetic strings may be enclosed in apostrophes (single quotes) or quotation marks (double quotes). Strings containing commas must be enclosed in either single or double quotes. If single or double quotes are used, double the quote character to use it within the string. Optionally, you can use the `DELIMITED BY` clause to specify a different delimiter string than the default which is a comma.
- Three other special sequences are also recognized. The two characters `\n` represent a newline character, `\\` represents a single `\`, and the sequence `\xDD` represents the character with hexadecimal code `DD`.
- DBASE** The file is in dBASE II or dBASE III format. ISQL will attempt to determine which of the two DBase formats the file is based on information in the file. If the table doesn't exist, it will be created.
- DBASEII** The file is in dBASE II format. If the table doesn't exist, it will be created.
- DBASEIII** The file is in dBASE III format. If the table doesn't exist, it will be created.
- DIF** Input file is in Data Interchange Format. If the table doesn't exist, it will be created.
- FIXED** Input lines are in fixed format. The width of the columns can be specified using the `COLUMN WIDTHS` clause. If they are not specified, then column widths in the file must be the same as the maximum number of characters required by any value of the corresponding database column's type.
- FOXPRO** The file is in FoxPro format (the FoxPro memo field is different than the dBASE memo field). If the table doesn't exist, it will be created.
- LOTUS** The file is a Lotus WKS format worksheet. `INPUT` assumes that the first row in the Lotus WKS format worksheet is column names. If the table doesn't exist, it will be created. In this case, the types and sizes of the columns created may not be correct



because the information in the file pertains to a cell, not to a column.

**WATFILE** The input will be a WATFILE file. If the table doesn't exist, it will be created. WATFILE is a tabular file management tool available from WATCOM.

## ISQL\_LOG

If ISQL\_LOG is set to a non-empty string, all ISQL commands are added to the end of the named file. Otherwise, if ISQL\_LOG is set to the empty string ISQL commands are not logged.

**NOTE:** This is logging of an individual ISQL session only. See "Backup and Recovery" on page 115 for a description of the transaction log which logs all changes to the database by all users.

## NULLS

Specifies how NULL values in the database will be displayed. The default is ' ( NULL ) ' (including the parentheses).

## ON\_ERROR

Controls what happens if an error is encountered while reading commands from a command file as follows:

**STOP** ISQL will stop reading commands from the file and return to the command window for input.

### PROMPT

ISQL will prompt the user to see if the user wishes to continue.

### CONTINUE

The error will be ignored and ISQL will continue reading commands from the command file. The INPUT command will continue with the next row, skipping the row that caused the error.

**EXIT** ISQL will terminate.

## OUTPUT\_FORMAT

Sets the output format for the data retrieved by the `SELECT` command and redirected into a file. This is also the default output format for the `OUTPUT` command. The valid output formats are:

**ASCII** The output is an ASCII format file with one row per line in the file. All values are separated by commas and strings are enclosed in apostrophes (single quotes). The delimiter and quote strings can be changed using the `DELIMITED BY` and `QUOTE` clauses. If `ALL` is specified in the `QUOTE` clause, then all values (not just strings) will be quoted.

Three other special sequences are also used. The two characters `\n` represent a newline character, `\\` represents a single `\`, and the sequence `\xDD` represents the character with hexadecimal code `DD`. This is the default output format.

**DBASEII** The output is a dBASE II format file with the column definitions at the top of the file. Note that a maximum of 32 columns can be output. Also, note that columns longer than 255 characters will be truncated in the file.

**DBASEIII** The output is a dBASE III format file with the column definitions at the top of the file. Note that a maximum of 128 columns can be output. Also, note that columns longer than 255 characters will be truncated in the file.

**DIF** The output is a file in the standard Data Interchange Format.

**FIXED** The output is fixed format with each column having a fixed width. The width for each column can be specified using the `COLUMN WIDTH` clause. If this clause is omitted, the width for each column is computed from the data type for the column, and is large enough to hold any value of that data type. No column headings are output in this format.

**FOXPRO** The output is a FoxPro format file (the FoxPro memo field is different than the dBASE memo field) with the column definitions at the top of the file. Note that a maximum of 128 columns can be output. Also, note that columns longer than 255 characters will be truncated in the file.

- LOTUS** The output is a Lotus WKS format worksheet. Column names will be put as the first row in the worksheet. Note that there are certain restrictions on the maximum size of Lotus WKS format worksheets that other software (such as Lotus 1-2-3) can load. There is no limit to the size of file ISQL can produce.
- SQL** The output is an ISQL INPUT command required to recreate the information in the table.
- TEXT** The output is a TEXT format file which prints the results in columns with the column names at the top and vertical lines separating the columns. This format is similar to that used to display data in the ISQL data window.
- WATFILE** The output is a WATFILE format file with the column definitions at the top of the file. WATFILE is a tabular file management tool available from WATCOM.

## OUTPUT\_LENGTH

Controls the length used when ISQL exports information to an external file (using output redirection or the OUTPUT command). The default for Output\_length is 0—no truncation.

## STATISTICS

Controls whether execution times, optimization strategies and other statistics will be displayed in the statistics window. This option can be set to 0, 3, 4, 5, or 6. When 0, the statistics window is not displayed. Otherwise, the value represents the height of the statistics window in lines.

## TRUNCATION\_LENGTH

When SELECT statement results are displayed on the screen, each column of output is limited to the width of the screen. The TRUNCATION\_LENGTH option is used to reduce the width of wide columns so that more than one column will fit on the screen. A value of 0 means that columns will not be truncated.

## ISQL examples

1. SET OPTION public.date\_format = 'Mmm dd yyyy'
2. SET TEMPORARY OPTION wait\_for\_commit = on

## SET variable

<b>Syntax</b>	SET identifier = expression
<b>Purpose</b>	To assign a value to a SQL variable.
<b>Usage</b>	Anywhere.
<b>Authorization</b>	None.
<b>Side effects</b>	None.
<b>See also</b>	CREATE VARIABLE, DROP VARIABLE, Expressions.
<b>Description</b>	<p>The SET command assigns a new value to a variable that was previously created using the CREATE VARIABLE command.</p> <p>A variable can be used in a SQL statement anywhere a column name is allowed. If there is no column name that matches the identifier, the database engine checks to see if there is a variable that matches and uses its value.</p> <p>Variables are local to the current connection, and disappear when you disconnect from the database or when you use the DROP VARIABLE command. They are not affected by COMMIT or ROLLBACK statements.</p> <p>Variables are necessary for creating large text or binary objects for INSERT or UPDATE statements from embedded SQL programs because embedded SQL host variables are limited to 32,767 bytes.</p>

# SIGNAL

<b>Syntax</b>	SIGNAL exception-name
<b>Purpose</b>	Signal an exception condition.
<b>Usage</b>	Procedures and triggers only.
<b>Authorization</b>	None.
<b>Side effects</b>	None.
<b>See also</b>	"Errors in procedures and triggers" on page 162, RESIGNAL, Compound Statements.
<b>Description</b>	SIGNAL allows you to raise an exception. See "Errors in procedures and triggers" on page 162 for a description of how exceptions are handled.

## Example

```
DECLARE COLUMN_NOT_FOUND EXCEPTION FOR SQLSTATE '52003';  
.  
.  
.  
SIGNAL COLUMN_NOT_FOUND;
```

## UNION

**Syntax**                   select-without-order-by UNION [ALL] select-without-order-by  
                              ... [ UNION [ALL] select-without-order-by ] ...  
                              ... [ ORDER BY integer [ ASC | DESC ], ... ]

**Purpose**                   To combine the results of two or more select commands.

**Usage**                    Anywhere.

**Authorization**        Must have SELECT permission for each of the component SELECT commands.

**Side effects**           None.

**See also**                SELECT.

**Description**           The results of several SELECT commands can be combined into a larger result using UNION. The component SELECT commands must each have the same number of items in the select list, and cannot contain an ORDER BY clause.

The results of UNION ALL is simply the combined results of the component SELECT commands. The results of UNION are the same as UNION ALL except that duplicate rows are eliminated. Since eliminating duplicates requires extra processing, UNION ALL should be used instead of UNION where possible.

If corresponding items in two select lists have different data types, Watcom SQL will choose a data type for the corresponding column in the result and automatically convert the columns in each component SELECT command appropriately.

If ORDER BY is used, only integers are allowed in the order by list. These integers specify the position of the columns to be sorted.

The column names displayed are the same column names which would be displayed for the first SELECT statement.

## Examples

This first example lists all distinct surnames of employees and customers.

```
SELECT emp_lname FROM Employee
UNION SELECT cus_lname FROM Customer
```

The following example produces a list of all employees having each skill combined with an extra line for each skill showing the number of employees with that skill.

```
SELECT 'Detail', skill, emp_id FROM Skill JOIN Expertise
UNION ALL
SELECT 'Total', skill, count(*) FROM Skill
GROUP BY skill
ORDER BY 2, 1
```

# UPDATE

<b>Syntax</b>	UPDATE table-list SET  ... column-name = expression, ...  ... [ WHERE search-condition ]  ... [ ORDER BY expression [ ASC   DESC ], ... ]
<b>Purpose</b>	To modify data in the database.
<b>Usage</b>	Anywhere.
<b>Authorization</b>	Must have UPDATE permission for the columns being modified.
<b>Side effects</b>	None.
<b>See also</b>	DELETE, INSERT.
<b>Description</b>	<p>The UPDATE command is used to modify rows of one or more tables. Each named column is set to the value of the expression on the right hand side of the equal sign. There are no restrictions on the <b>expression</b>. Even <b>column-name</b> can be used in the expression—the old value will be used. For a description of the table list and how to do joins, see "FROM" on page 243.</p> <p>If no WHERE clause is specified, every row will be updated. If a WHERE clause is specified, then only those rows which satisfy the search condition will be updated.</p> <p>Normally, the order that rows are updated doesn't matter. However, in conjunction with the NUMBER(*) function, an ordering can be useful to get increasing numbers added to the rows in some specified order. Also, if you wish to do something like add 1 to the primary key values of a table, it is necessary to do this in descending order by primary key, so that you do not get duplicate primary keys during the operation.</p> <p>Views can be updated provided the SELECT command defining the view does not contain a GROUP BY clause, an aggregate function, or involve a UNION operation.</p>



## Examples

1. UPDATE Employee SET emp\_lname = 'Brussel'  
WHERE emp\_id = 86004
2. UPDATE Employee KEY JOIN Expertise KEY JOIN Skill  
SET skill\_level = 5  
WHERE emp\_lname = 'Brooks' AND skill = 'COBOL'

## UPDATE (positioned)

<b>Syntax</b>	UPDATE table-list SET ... column-name = expression, ... ... WHERE CURRENT OF cursor-name
<b>Purpose</b>	To modify the data at the current location of a cursor.
<b>Usage</b>	Procedures and triggers only.
<b>Authorization</b>	Must have UPDATE permission on the columns being modified.
<b>Side effects</b>	None.
<b>See also</b>	UPDATE, DELETE, DELETE (positioned), INSERT, FETCH.
<b>Description</b>	<p>This form of the UPDATE statement updates the current row of the specified cursor. The current row is defined to be the last row FETCHed from the cursor and the last operation on the cursor must not have been a DELETE Positioned.</p> <p>The requested columns are set to the specified values for the row at the current row of the specified query. The columns do not need to be in the select list of the specified open cursor.</p>

### Example

```
UPDATE Employee SET emp_lname = 'Jones'  
WHERE CURRENT OF emp_cursor  
USING sqlda;
```

# VALIDATE TABLE

**Syntax** VALIDATE TABLE [creator.]table-name

**Purpose** To validate a table in the database.

**Usage** DBA notepad.

**Authorization** Must be the creator of the table or have DBA authority.

**Side effects** None.

**See also** DBVALID

**Description** The VALIDATE TABLE command will scan every row of a table and look each row up in each index on the table. If the database file is corrupt, an error will be reported. This should not happen. However, because DOS and Windows are unprotected operating environments, other software can corrupt memory used by the database engine. This problem may be detected through software errors or crashes, or the corrupt memory could get written to the database creating a corrupt database file. Also, in any operating system, hardware problems with the disk could cause the database file to get corrupted. If you cannot determine how a database file became corrupted, please report the problem to WATCOM.

If you do have errors reported, you can drop all of the indexes and keys on a table and recreate them. Any foreign keys to the table will also need to be recreated. Another solution to errors reported by VALIDATE TABLE is to unload and reload your entire database. You should use the -u option of DBUNLOAD so that it will not try to use a possibly corrupt index to order the data.



## CHAPTER 17

# Program Summary

- About this chapter** This chapter presents all of the executable files that comprise Watcom SQL and a description of each.
- The commands are presented with a syntax summary in a box at the top of each page. Below the syntax box, you will find a list of related commands, a detailed description of the command and a summary of the command line switches.
- Contents** The programs are listed alphabetically.

# DBBACKUP

## Syntax

Windows NT: DBBACKUP [switches] directory

Windows: DBBACKW [switches] directory

Switch	Description
-c "keyword=value; ..."	Supply database connection parameters
-d	Only back up main database file
-q	Quiet mode—do not print messages
-r	Rename and restart transaction log
-s "cmd"	Specify command line to start database
-t	Only back up transaction log
-w	Only back up write file
-x	Delete and restart transaction log
-y	Replace files without confirmation

## See also

DBWRITE

## Description

Make a backup copy of all the files making up a single database. A simple database consists of two files: the main database file and the transaction log. More complicated databases can store tables in multiple files, called **dbspaces**. All filenames will be the same as the filenames on the server.

Each file of a database is copied to an identical file of the same name in the specified output directory. In fact, the DBBACKUP command is equivalent to copying the database files when the database is not running. It is provided to allow a database to be backed up while other applications or users are using the database.

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

See "Backup and Recovery" on page 115 for a description of suggested backup procedures.

## Switches

- c "keyword=value; ..."**  
Specify connection parameters. See "Connection parameters" on page 53 for a description of the connection parameters. If this option is not specified, the environment variable `SQLCONNECT` is used.  
  
The **user ID** specified must have DBA authority.
- d**  
Back up the main database files only, without backing up the transaction log file.
- q**  
Operate quietly. Do not print messages.
- r**  
Rename and restart the transaction log after successful backup. Using `-r` will force the transaction log to be backed up. The renamed transaction log (on the server) will have a filename the same as the transaction log with an extension of **nnn**, where **nnn** is the first available number starting at 000. `DBBACKUP` will display the name of the renamed transaction log. This option will cause the backup to wait for a point when all transactions from all connections are committed.
- s "cmd"**  
Historical. The start command was added to the connection parameters as the **Start** parameter.
- t**  
Back up the transaction log file only. This can be used as a partial backup since the transaction log can be applied to the most recently backed up copy of the database file(s).
- w**  
Back up the database write file only. See `DBWRITE` for a description of database write files.
- x**  
Delete and restart the transaction log after successful backup. Using `-x` will force the transaction log to be backed up. This option will cause the backup to wait for a point when all transactions from all connections are committed.

**-y**

Operate without confirming actions. Normally, you will be prompted to confirm the creation of the backup directory or the replacement of a previous backup file in the directory.



# DBCOLLAT

## Syntax

Windows NT: DBCOLLAT [switches] output-file

Windows: DBCOLLW [switches] output-file

Switch	Description
-c "keyword=value; ..."	Supply database connection parameters
-e	Include empty mappings
-q	Quiet mode — do not print messages
-s "cmd"	Specify command line to start database
-x	Use hex for extended characters (7F-FF)
-y	Replace file without confirmation
-z col-seq	Specify collating sequence label

## See also

DBINIT

## Description

Extracts a collation (sorting sequence) from the SYS.SYSCOLLATION table of a database into a file suitable for use with DBINIT to create a database using a custom collation.

The file produced by DBCOLLAT can be modified and used with the `-z` option of DBINIT to create a new database with a custom collation. Ensure that the label is changed on the line that looks like:

```
Collation label (name)
```

Otherwise, DBINIT will reject the collation.

## Switches

**-c "keyword=value; ..."**

Specify connection parameters. See "Connection parameters" on page 53 for a description of the connection parameters. If this option is not specified, the environment variable SQLCONNECT is used.

- e** Normally, collations don't specify the actual value that a character is to sort to. Instead, each line of the collation sorts to one position higher than the previous line. However, older collations have gaps between some sort positions. Normally, DBCOLLAT will skip the gaps and write the next line with an explicit sort-position. The **-e** switch will cause DBCOLLAT to write empty mappings (consisting of just a colon(:)) for each line in the gap.
- q** Operate quietly. Do not print messages.
- s "cmd"** Historical. The start command was added to the connection parameters as the **Start** parameter.
- x** Extended characters (whose value is greater than hex 7F) may or may not appear correctly on your screen, depending on whether or not the code page in use on your computer is the same as the code page being used in the collation you are extracting. The **-x** switch causes DBCOLLAT to write all characters at hex 7F or above to be written as a two-digit hexadecimal number, in the form:

\xdd

(For example, \x80, \xFE). Normally, only characters from hex 00 to hex 1F, hex 7F and hex FF are written in hexadecimal form.
- y** Operate without confirming actions. Normally, you will be prompted to confirm replacing an existing collation file.
- z col-seq** The label of the collation to be extracted. The names of the collation sequences can be found in the collation\_label column of the SYS.SYSCOLLATION table. If no **-z** is specified, then DBCOLLAT will extract the collation being used by the database.

# DBERASE

## Syntax

Windows NT: DBERASE [switches] database-name

Windows: DBERASEW [switches] database-name

Switch	Description
-q	Operate quietly—do not print messages
-y	Erase database without confirmation

## See also

DBINIT

## Description

Erase a database file, log file, or write file. If -y is not specified, you will be prompted whether you really want to erase the file. Note that all Watcom SQL databases, write files, and log files are marked **read-only** to prevent accidental damage to the database or accidental deletion of the database files. Note that deletion of database file that references other dbspaces will not automatically delete the dbspace files.

## Switches

- q Operate quietly. Do not print informational messages. Confirmation prompts will still appear unless -y is specified.
- y Do not prompt whether you really want to erase the file.

# DBEXPAND

## Syntax

Windows NT: DBEXPAND [switches] compressed-database [database]

Windows: DBEXPANW [switches] compressed-database [database]

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

## See also

DBSHRINK

## Description

Expand a compressed database file created by DBSHRINK. DBEXPAND will read the given *compressed database* file and create an uncompressed database file. The uncompressed filename will default to the same name as the compressed filename with an extension of .db. The input filename extension will default to .cdb. The output filename (with extension) must not have the same name as the input filename (with extension).

## Switches

- q Operate quietly. Do not print messages.
- y Operate without confirming actions. Normally, you will be prompted to confirm replacing an existing database.

# DBINFO

## Syntax

Windows NT: DBINFO [switches] filename

Windows: DBINFOW [switches] filename

Switch	Description
-c "keyword=value; ..."	Supply database connection parameters
-q	Suppress any messages
-s "cmd"	Specify command line to start database
-u	Output page usage statistics

## See also

DBINIT

## Description

Display information about a database file or write file. The database engine should not be running when you run DBINFO. The information displayed includes the options specified when the database was created, the name of the transaction log file and other information.

## Switches

### **-c "keyword=value; ..."**

Specify connection parameters. See "Connection parameters" on page 53 for a description of the connection parameters. If this option is not specified, the environment variable SQLCONNECT is used.

### **-q**

Suppress any messages. The return code may still provide useful information (see "Software component return codes" on page 349).

### **-s "cmd"**

Historical. The start command was added to the connection parameters as the **Start** parameter.

### **-u**

Output page usage statistics. DBINFO must establish a connection with the database in order to get system table

information for the page usage statistics. DBINFO will start the database engine, get the system table information, and then stop the database engine. The connection parameters and start command switches are only used if this switch is specified.

# DBINIT

## Syntax

Windows NT: DBINIT [switches] new-database-file

Windows: DBINITW [switches] new-database-file

Switch	Description
-b	ANSI behavior of blanks
-c	Case sensitive database
-e	Encrypt database
-l	List available collating sequences
-n	No transaction log
-p page-size	Set page size
-q	Quiet mode—do not print messages
-s "cmd"	Specify command line to start database
-t log-name	Transaction log filename (default is database name with .log)
-z col-seq	Collation sequence used for comparisons

## See also

DBCOLLAT, DBERASE, DBSTART

## Description

DBINIT will initialize a database with the given database filename. The default filename extension is .db.

## Switches

**-b** ANSI behavior of blanks. Trailing blanks are ignored for comparison purposes, and embedded SQL programs pad strings fetched into character arrays. For example, "Smith" and "Smith " would be treated as equal using the -b option. This option is provided for compatibility with the ANSI SQL standard.

The default is that blanks are significant for comparisons, which was the behavior supported in Watcom SQL Version 3.0.

- c** Case sensitivity. All names (table, columns, and so on) and values are case sensitive, so that Studnum is not the same as STUDNUM. This option is provided for compatibility with the ANSI SQL standard.
- The default is that all comparisons are case insensitive, which was the only behavior supported in Watcom SQL Version 3.0.
- e** Encrypt database. Encryption will make it much more difficult for someone to decipher the data in your database by using a disk utility to look at the file. File compaction utilities will not work well on encrypted database files.
- l** List the available collating sequences.
- n** Do not use a transaction log.
- p page-size** The *page size* can be 512, 1024, 2048 or 4096 bytes, with 1024 being the default. Other values for the size will be changed to the next larger size. Large databases usually benefit from a larger page size.
- q** Operate quietly. Do not print messages.
- s "cmd"** Specifies the command string to start the database engine or network requestor (DBCLIENT). If this option is not specified, the environment variable SQLCONNECT is checked, and if it is not defined then the default is DBSTART -q %d (DBSTARTW -q %d in Windows).
- t log-name** Set the transaction log filename. The transaction log is a file where the database engine will log all changes made by all users no matter what application system is being used. The transaction log plays a key role in backup and recovery (see "Transaction log" on page 118). If the filename has no path, it will always go in the same directory as the database file. If you do not specify -t or -n, the default transaction log will be the same filename as the database file with the extension .log.
- z col-seq** Specify a collating sequence or filename. The collation sequence is used for all string comparisons in the database.



The database is created with all collations built into SYS.SYSCOLLATION. If -z is specified and names one of the collations, then that collation will be used by the database. If -z is specified but does not name one of the collations, then it is assumed that the name is a filename, and the file will be opened and the collation will be parsed from the file. The specified collation will then be used by the database.

If -z is not specified, then the default collation will be used. Normal ASCII (binary) ordering will be used for the lower 128 characters. For the upper 128 characters (also called the extended characters), characters which are accented forms of a letter in the lower 128 are sorted to the same position as the unaccented form. The determination of whether or not an extended character is a letter is based upon code page 850 (multilingual code page).

If the user wants a custom collation, then it is recommended that DBCOLLAT be used to extract the closest collation from the database, then the user may modify the collation and use the DBINIT -z switch to specify the new collation. It is important to change the collation label, otherwise DBINIT will not allow the insertion of the new collation, since it will be a duplicate label. See "Database Collations" on page 173 for a full description of custom collating sequences.

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

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

Label	Explanation
437EBCDIC	(Code Page 437, EBCDIC)
437LATIN1	(Code Page 437, Latin 1)
437ESP	(Code Page 437, Spanish)
437SVE	(Code Page 437, Swedish/Finnish)
850CYR	(Code Page 850, Cyrillic)
850DAN	(Code Page 850, Danish)
850ELL	(Code Page 850, Greek)
850ESP	(Code Page 850, Spanish)
850ISL	(Code Page 850, Icelandic)
850LATIN1	(Code Page 850, Latin 1)
850LATIN2	(Code Page 850, Latin 2)
850NOR	(Code Page 850, Norwegian)
850RUS	(Code Page 850, Russian)
850SVE	(Code Page 850, Swedish/Finnish)
850TRK	(Code Page 850, Turkish)
852LATIN2	(Code Page 852, Latin 2)
852CYR	(Code Page 852, Cyrillic)
855CYR	(Code Page 855, Cyrillic)
856HEB	(Code Page 856, Hebrew)
857TRK	(Code Page 857, Turkish)
860LATIN1	(Code Page 860, Latin 1)
861ISL	(Code Page 861, Icelandic)
862HEB	(Code Page 862, Hebrew)
863LATIN1	(Code Page 863, Latin 1)
865NOR	(Code Page 865, Norwegian)
866RUS	(Code Page 866, Russian)
869ELL	(Code Page 869, Greek)

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

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

**User ID and password**

All databases are created with one user ID: DBA with password SQL. Note that if you specify the -c command line switch for case-sensitive comparisons, the DBA user ID and its password must be entered in uppercase.

# DBLOG

## Syntax

Windows NT: DBLOG [switches] database-file

Windows: DBLOGW [switches] database-file

Switch	Description
-n	Remove transaction log name
-q	Quiet mode—do not print messages
-t log-name	Set transaction log name

## See also

DBINIT, DBLOG, DBSTART, DBTRAN, "Transaction log" on page 118

## Description

Display or change the name of the transaction log. The initial name of the transaction log is determined when the database is initialized by DBINIT. DBLOG will work with database files or with write files. The database engine must not be running on that database when the transaction log filename is changed (an error message will be displayed if it is).

## Switches

- n** Do not use a transaction log.
- q** Operate quietly. Do not print messages.
- t log-name** Change the name of the transaction log file.

# DBSHRINK

## Syntax

Windows NT: DBSHRINK [switches] database [compressed-database]

Windows: DBSHRINW [switches] database [compressed-database]

Switch	Description
-q	Quiet mode—do not print messages
-y	Erase existing output file without confirmation

## See also

DBEXPAND, DBWRITE

## Description

Compress a database file. This can be very useful in situations where there is limited disk space. Compressed databases are usually 40 to 60 percent of their original size. They can only be used in conjunction with write files. The database engine cannot update compressed database files.

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

## Switches

- q** Operate quietly. Do not print any messages.
- y** Operate without confirming actions. Normally, you will be prompted to confirm replacing an existing file.

# DBSTART

## Syntax

Windows NT: DBSTART [engine-switches] [database [database-switches]]\*  
 RTSTART [engine-switches] [database [database-switches]]\*  
 DB32 [engine-switches] [database [database-switches]]\*  
 RT32 [engine-switches] [database [database-switches]]\*

Windows: DBSTARTW [engine-switches] [database [database-switches]]\*  
 RTSTARTW [engine-switches] [database [database-switches]]\*  
 DB32W [engine-switches] [database [database-switches]]\*  
 RT32W [engine-switches] [database [database-switches]]\*

Engine switch	Description
-b	Run in bulk operations mode
-c cache-size	Set maximum cache size
-ga	Automatically shutdown after last database closed
-gc num	Set checkpoint timeout period
-gd level	Set database starting permission
-gf	Disable firing of triggers
-gi num	Set maximum number of concurrent i/o requests
-gn num	Set number of threads
-gp size	Set maximum page size
-gr num	Set maximum recovery time
-gs size	Set thread stack size
-n name	Name the database engine
-q	Quiet mode—suppress output
-v	Log old values of all columns on UPDATE

Recovery switch	Description
-a log-file	Apply named transaction log file
-f	Force database to start without transaction log

Database switch	Description
-n name	Name the database
-v	Log old values of all columns on UPDATE
Windows switch	Description
-d	Use DOS I/O instead of direct I/O

**See also**

DBINIT, DBSHRINK, DBSTOP, DBWRITE

**Description**

Start the Watcom SQL database engine. The database engine can be started with a number of database files or no database files. Command-line switches specified before any databases apply to the engine and all databases. Switches specified after a particular database apply only to that database file. Applications that use the Watcom SQL database engine can cause additional database files to be loaded by a database engine after it has started.

If a *database* is specified without a file extension, Watcom SQL first looks for *database* with extension .WRT, followed by *database* with extension .DB.

In Windows, the engines are programs that run as a separate task. DB32W is a 32-bit version of the Windows database engine. The 32-bit version has much better performance than the 16-bit version. It requires Windows running in enhanced mode.

RTSTART, RTSTARTW, RT32 and RT32W are runtime versions of the Watcom SQL database engine.

**Engine switches**

- b** Use bulk operation mode. This is useful when loading large quantities of data into a database. The database engine will only allow one connection by one application. It will not keep a rollback log or a transaction log, and the multiuser locking mechanism is turned off.
- c cache-size** Set the size of the cache. The database engine will use extra memory for caching database pages. Any cache-size less than 10000 will be assumed to be K-bytes (1K = 1024

- bytes). The cache-size may also be specified as nK or nM (1M = 1024K). By default, the database engine will use 2 megabytes of memory for caching.
- ga** Automatically shutdown the engine after the last database is closed. Applications can cause databases to be started and stopped by the engine. Specifying this switch will cause the engine to shutdown when the last database is stopped.
- gc num** Set the maximum desired length of time (in minutes) that the database engine will run without doing a checkpoint. See the description of the `CHECKPOINT_TIME` option in "SET OPTION" on page 283. When a database engine is running with multiple databases, the checkpoint time specified by the first database started will be used unless overridden by this switch.
- gd level** Set the database starting permission to *level*. This is the permission level required by a user to cause a new database file to be loaded by the engine. The level can be one of:
- ◆ **dba** Only users with DBA authority can start new databases.
  - ◆ **all** All users can start new databases.
  - ◆ **none** Starting new databases is not allowed.
- gf** Disable firing of triggers by the engine.
- gi num** Sets the maximum number of concurrent i/o requests that will be issued by the database engine. The default is one. Increasing the number of i/o requests can significantly improve performance on machines with appropriate hardware. This option only applies to the Windows NT version of the database engine.
- gn num** Sets the number of execution threads that will be used in the database engine while running with multiple users. See the description of the `THREAD_COUNT` in "SET OPTION" on page 283. When a database engine is running with multiple databases, the thread count specified by the



first database started will be used unless overridden by this switch.

**-gp size**

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

**-gr num**

Set the maximum desired length of time (in minutes) that the database engine will take to recover from system failure. See to the description of the RECOVERY\_TIME option in "SET OPTION" on page 283. When a database engine is running with multiple databases, the recovery time specified by the first database started will be used unless overridden by this switch.

**-gs size**

Sets the stack size of every thread in the engine (in bytes).

**-n name**

Set the name of the database engine. By default, the database engine receives the name of the database file with the path and extension removed. For example, if the engine is started on C:\WSQL40\SAMPLE.DB and no -n switch is specified, then the name of the engine will be **sample**.

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

**-q**

Operate quietly. Suppress all output.

**Recovery switches****-a log-file**

Apply the named transaction log. This is used to recover from media failure on the database file (see "Backup and Recovery" on page 115). When this option is specified, the

database engine will apply the log and then terminate—it will not continue to run.

- f force the database engine to start without a transaction log. This is only used for recovery when you would like to force the database engine to start after the transaction log has been lost (see "Backup and Recovery" on page 115). When this option is specified, the database engine will force a checkpoint recovery of the database and then terminate—it will not continue to run.

### Database switches

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

By default, the database file receives the name of the file with the path and extension removed. For example, if the engine is started on C:\WSQL40\SAMPLE.DB and no -n switch is specified, then the name of the file will be **sample**.

- v Cause the engine to record the previous values in the transaction log of each of the columns whenever a row of the database is updated. By default, the engine will only record enough information to uniquely identify the row (primary key values or values from a not null unique index). This switch is useful when you are working on a copy of a database file (see "Portable computers" on page 112).

### Windows switches

- d use normal DOS input and output instead of direct input and output. By default the database engine will read and write database pages with low-level direct calls that are usually much faster than normal DOS calls. When your database file is small (0-3Mb) and you have disk caching software running, DOS calls can be faster than direct calls. Do not use this switch when your database is large (>10Mb).

**Windows 3.11**

This switch may be necessary for some disk controllers and 32-bit disk access in Windows 3.11.

**Automatic loading**

DBBACKUP, DBINIT, DBUNLOAD, DBVALID, and ISQL will automatically load the database engine if it has not been previously loaded. The multiuser client can also be started automatically in this way. See "Environment variables" on page 346 and the respective commands for more details. If the database engine (or client) is started automatically, it will be unloaded automatically when the software is finished executing.

# DBSTOP

## Syntax

Windows NT: DBSTOP [switches] name

Windows: DBSTOPW [switches] name

Switch	Description
-c	Do not stop if there are active connections
-q	Quiet mode—do not print messages

## See also

DBSTART

## Description

Stop the database engine . All memory used by the database engine will be returned to the system for use by other applications. If -q is specified, no error message is printed if the Watcom SQL database engine was not running.

The *name* specifies the engine name.

This command is not necessary after the database engine has been started by ISQL ISQL will automatically unload the database engine when it disconnects.

## Switches

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

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

# DBTRAN

## Syntax

Windows NT: DBTRAN [switches] transaction-log [sql-log]

Windows: DBTRANW [switches] transaction-log [sql-log]

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

## See also

DBLOG, DBSTART

## Description

Translate a transaction log into a SQL command file. The *sql-log* filename will default to the transaction log name with a .sql extension.

## Switches

- a** Include transactions that were not committed.
- f** Only the transactions completed since the last checkpoint are translated.
- r** Remove any transactions that were not committed (this is the default).
- s** Generate ANSI standard SQL.
- t** Transactions generated by triggers will be included in the output file.

- u user,...** Translate transactions for listed users only.
- x user,...** Translate transactions except for listed users.
- y** Operate without confirming actions. Normally, you will be prompted to confirm replacing an existing SQL file.
- t** Transactions generated by triggers will be included only as comments in the output file.

# DBUNLOAD

## Syntax

Windows NT: DBUNLOAD [switches] directory

Windows: DBUNLOAW [switches] directory

Switch	Description
-c "keyword=value; ..."	Supply database connection parameters
-d	Unload data only
-n	No data—schema definition only
-q	Quiet mode—no windows or messages
-r reload-file	Specify name of generated reload ISQL command file (default RELOAD.SQL)
-s "cmd"	Specify command line to start database
-u	Unordered data
-v	Verbose messages
-y	Replace command file without confirmation

## Description

Unload a database and put data files into *directory*. DBUNLOAD will create the ISQL command file RELOAD.SQL to rebuild your database from scratch. It will unload all of the data in each of your tables in ASCII comma delimited format into files in the specified directory. Binary data will be properly represented with escape sequences.

### Note

DBUNLOAD should be run from a DBA user ID. It is the only way you can be assured to have enough privileges to unload all the data. Also, the RELOAD.SQL file should be run from the DBA user ID. (Usually it will be run on a new database where the only user ID is DBA with password SQL.)

## Switches

- c "keyword=value; ..."** Specify connection parameters. See "Connection parameters" on page 53 for a description of the connection parameters. If this option is not specified, the environment variable `SQLCONNECT` is used.
- d** Unload data only. None of the database definition commands will be generated (`CREATE TABLE`, `CREATE INDEX`, and so on); `RELOAD.SQL` will contain statements to reload the data only.
- n** Unload schema definition only. None of the data in the database will be unloaded; `RELOAD.SQL` will contain SQL statements to build the structure of the database only.
- q** Operate quietly. No messages except errors will be displayed.
- r reload-file** Modify the name and directory of the generated reload ISQL command file. The default is `RELOAD.SQL` in the current directory.
- s "cmd"** Historical. The start command was added to the connection parameters as the **Start** parameter.
- u** Output the data unordered. Normally the data in each table is ordered by the primary key. Use this option if you are unloading a database with a corrupt index so that the corrupt index is not used to order the data.
- v** Enable verbose mode. The table name of the table currently being unloaded and how many rows have been unloaded will be displayed.
- y** Operate without confirming actions. Normally, you will be prompted to confirm replacing an existing ISQL command file.

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



To reload a database, you need to DBINIT a new database and then run the generated RELOAD.SQL command file through ISQL. In Windows NT, there is a file (REBUILD.BAT, REBUILD.CMD, or REBUILD) with Watcom SQL that will automate the unload and reload. In Windows, this can be done from the DBTOOLS application.

# DBUPGRAD

## Syntax

Windows NT: DBUPGRAD [switches]

Windows: DBUPGRDW [switches]

Switch	Description
-c "keyword=value; ..."	Supply database connection parameters
-q	Quiet mode—no windows or messages

## See also

DBINIT

## Description

Watcom SQL Version 4.0 adds the ability to create procedures in a database. The DBUPGRAD utility upgrades a database created with Watcom SQL Version 3.2 so that it can contain procedures.

This utility has two major purposes.

- ◆ For people switching from version 3.2 to 4.0, this utility can be used to upgrade their databases without having to unload and reload their databases.
- ◆ For users who want to run both the 3.2 version of DBSTART and the 4.0 version, and want to add procedures to their database. Version 3.2 of DBSTART will not run with a database created with the 4.0 version of DBINIT. The 4.0 version of DBSTART will run against a 3.2 created database, but procedures can not be created.

Procedures can only be created and called using the 4.0 version of DBSTART. If triggers are added to a database, it will no longer be possible to run the 3.2 version of DBSTART.

### -c "keyword=value; ..."

Specify connection parameters. See "Connection parameters" on page 53 for a description of the connection parameters. If this option is not specified, the environment variable SQLCONNECT is used. The DBUPGRAD utility must be run by a user with DBA authority.

**-q**

Operate quietly. No messages except errors will be displayed.

# DBVALID

## Syntax

Windows NT: DBVALID [switches] [table-name]

Windows: DBVALIDW [switches] [table-name]

Switch	Description
-c "keyword=value; ..."	Supply database connection parameters
-q	Quiet mode—do not print messages
-s "cmd"	Specify command line to start database

## See also

DBSTART, "VALIDATE TABLE" on page 305

## Description

Validate all indexes on a table in the database. The *table-name* parameter can contain the name of a particular table to validate, or a list of table names separated by commas. If no *table-name* is specified, DBVALID will validate all tables in a database (including the system tables). Validation involves scanning the entire table, and looking up each record in every index and key defined on the table. See "VALIDATE TABLE" on page 305.

This utility can be used in combination with regular backups (see "Backup and Recovery" on page 115) to give you confidence in the security of the data in your database.

## Switches

### -c "keyword=value; ..."

Specify connection parameters. See "Connection parameters" on page 53 for a description of the connection parameters. If this option is not specified, the environment variable SQLCONNECT is used.

### -q

Operate quietly. Do not print messages.

### -s "cmd"

Historical. The start command was added to the connection parameters as the **Start** parameter.

# DBWRITE

## Syntax

Windows NT: DBWRITE [switches] db-name [write-name [log-name]]

Windows: DBWRITEW [switches] db-name [write-name [log-name]]

Switch	Description
-c	Create a new write file
-d	Point a write to a different database
-q	Quiet mode—do not print messages
-s	Report write file status (default)
-y	Erase/replace old files without confirmation

## See also

DBLOG, DBSHRINK, DBSTART

## Description

DBWRITE is used to manage database **write files**. A **write file** is a file attached to a particular database into which all changes will be written, leaving the original database unchanged. Write files can be used effectively for testing when you do not wish to modify the production database. They can also be used in network and student environments where read-only access to a database is desired. If you are using a compressed database, then you must use a write file. The write filename is then used in place of the database name in the ISQL database connection window or on the DBSTART command line. If any changes are made to the original database (not using the write file), the write file will no longer be valid. This happens if you start the engine using the original database file and make a modification to it. It can be made valid again by the command:

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

However, this will delete all changes that were recorded in the write file.

The log-name parameter is only used with the -c parameter. The write-name parameter is only used with the -c and -d parameters. Note that the db-name parameter must be specified before the write-name parameter.

**Switches**

- c** Create a new write file. If an existing write already exists, any information in the old write file will be lost. If no write filename is specified on the command line, the write filename will default to the database name with the extension **.WRT**. If no transaction log name is specified, the log filename will default to the database name with the extension **.WLG**.
- d** Change the database that an existing write file points to.
- q** Operate quietly. Do not print messages.
- s** Report write file status only. This displays the name of the database that the write file points to.
- y** Operate without confirming actions. Normally, you will be prompted to confirm the replacement of an existing write file.

# ISQL

## Syntax

Windows NT: ISQL [switches] [isql-command]  
 RTSQL [switches] [isql-command]

Windows: ISQLW [switches] [isql-command]  
 RTSQLW [switches] [isql-command]

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

## See also

DBSTART

## Description

Starts the Interactive SQL environment. ISQL provides the user with an interactive environment for database administration:

- ◆ Querying data in a database,
- ◆ Modifying data in a database,
- ◆ Creating, altering, and dropping tables,
- ◆ Creating and dropping indexes,
- ◆ Managing userids and permissions,
- ◆ Setting database options,
- ◆ Transferring data to and from Watcom SQL.

Most administration is done by sending SQL commands to the database engine. ISQL allows you to type the SQL commands, or run ISQL command files. It will also provide feedback about the number of rows affected, the time required for each command, and any error messages.

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

In Windows, the executable name is ISQLW. You can set up program manager icons to start ISQLW with the same command line syntax as specified above.

ISQL will start the database engine automatically if it is not already started. It will then unload the engine on exit.

RTSQL and RTSQLW are the runtime SQL command processors. They are like ISQL except there is no interactive part, and they can only run command files. RTSQL will automatically load the database engine if it is not already loaded. It will then unload the database engine on exit. The command line switches are the same as those of ISQL.

## Switches

- b** Do not print the identification banner when ISQL starts up.
- c "keyword=value; ..."** Specify connection parameters. See "Connection parameters" on page 53 for a description of the connection parameters. If this option is not specified, the environment variable SQLCONNECT is used. If required connection parameters are not specified, then you will be presented with a dialog to enter the connection parameters.
- q** Operate quietly. ISQL will not display any information on the screen. This is only useful if a command or command file is executed when starting ISQL.
- s "cmd"** Historical. The start command was added to the connection parameters as the **Start** parameter.
- v** Verbose output. Information about each command is displayed as it is executed (RTSQL only).
- x** Scan commands but do not execute them. This is useful for checking long command files for syntax errors.



## Commands

ISQL commands are broken into three 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. Detailed descriptions of these SQL commands can be found in "Command Syntax" on page 181.

Data Manipulation Commands:

- CALL** Invoke a database procedure
- CHECKPOINT** Perform a disaster recovery checkpoint
- 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 within a transaction
- ROLLBACK** Undo changes to database
- SAVEPOINT** Establish a savepoint within a transaction
- SELECT** Retrieve information from the database
- UPDATE** Update rows in the database

Data Definition Commands:

- ALTER TABLE**  
Modify a database table definition
- COMMENT** Add comments to various database entities
- CREATE** Create database files, tables, indexes, views, procedures, and triggers

**DROP** Remove files, tables, indexes, views, procedures, and triggers from the database

**GRANT** Create user IDs, grant privileges to users

**REVOKE** Delete user IDs, revoke privileges from users

**SET OPTION** Set database options

**VALIDATE TABLE**

Check the validity of a database table and its indexes

- 2 Commands that are particular to ISQL and manipulate the ISQL environment. Detailed descriptions of these ISQL commands can be found in the "Command Syntax" on page 181.

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

**CONNECT** Reconnect to the database engine with a different userid and password

**DBTOOL** Invoke one of the database tools

**DISCONNECT** Disconnect from the database engine

**EXIT** Leave ISQL

**HELP** Enter ISQL on-line help facility

**INPUT** Do mass input into database tables

**OUTPUT** Output the current query results to a file

**PARAMETERS**

Specify the parameters to a command file

**QUIT** Leave ISQL

**READ** Execute ISQL command files

**SET CONNECTION**

Change the active database connection

- SET OPTION** Set options that control the ISQL environment
- 3 Simple commands that manipulate the data window. These commands are not described in the command summary section.
- CLEAR** Clear the data window
- DOWN [n]** Move the current query results down **n** lines. The default is one line.
- UP [n]** Move the current query results up **n** lines. The default is one line.

## Environment variables

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

### SQLCONNECT

#### Syntax

SQLCONNECT = keyword=value ; ...

SLQCONNCTCT = keyword#value ; ...

#### Description

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

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

## SQLSTART

### Syntax

SQLSTART = start-line  
SQLSTARTW = start-line

### Description

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

## TMP

### Syntax

TMP = directory  
TMPDIR = directory  
TEMP = directory

### Description

The database engine creates temporary files for various operations such as sorting and performing unions. These temporary files will be placed in the directory specified by the TMP, TMPDIR, or TEMP environment variables. (There is no standard name for this environment variable, so the database engine takes the first one of the three that it finds.)

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

## **WSQL**

### **Syntax**

WSQL = path

### **Description**

This environment variable is used to contain the directory where Watcom SQL is installed. (The default installation directory is `c:\wsq140`). The install procedure will automatically add a SET command for WSQL to your startup environment. The variable is used by the batch files or command files that build the Embedded SQL examples. It is also used by the Watcom SQL patch system to locate the installed software on your hard drive.

## Software component return codes

All database components use the following executable return codes. The header file SQLDEF.H has constants defined for these codes.

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





## APPENDIX A

# Watcom SQL Features

About this appendix    Watcom SQL conforms to the ANSI SQL89 standard but has many additional features defined in IBM's DB2 and SAA specification, and in ANSI SQL92.

This appendix describes those features of Watcom SQL that are not commonly found in SQL implementations.

## Differences from other SQLs

The following Watcom SQL features are not found in many other SQL implementations.

- ◆ Full type conversion is implemented. Any data type can be compared with or used in any expression with any other data type.
- ◆ Watcom SQL has date, time and timestamp types that includes a year, month and day, hour, minutes, seconds and fraction of a second. For insertions or updates to date fields, or comparisons with date fields, a free format date is supported.

In addition, the following operations are allowed on dates:

### **date + integer**

Add the specified number of days to a date.

### **date - integer**

Subtract the specified number of days from a date.

### **date - date**

Compute the number of days between two dates.

### **date + time**

Make a timestamp out of a date and time.

Also, many functions are provided for manipulating dates and times. See "Functions" on page 71 for a description of these.

- ◆ Watcom SQL supports both entity and referential integrity. This has been implemented via the following two extensions to the CREATE TABLE and ALTER TABLE commands.

```
PRIMARY KEY ( column-name, ... )

[NOT NULL] FOREIGN KEY [role-name] [(column-name, ...)]
REFERENCES table-name [(column-name, ...)]
[ CHECK ON COMMIT ]
```

The PRIMARY KEY clause declares the primary key for the relation. Watcom SQL will then enforce the uniqueness of the primary key, and ensure that no column in the primary key contains the NULL value.

The FOREIGN KEY clause defines a relationship between this table and another table. This relationship is represented by a column (or columns) in this table which must contain values in the primary key of another

table. The system will then ensure referential integrity for these columns — whenever these columns are modified or a row is inserted into this table, these columns will be checked to ensure that either they are all NULL or the values match the corresponding columns for some row in the primary key of the other table. For more information, see "CREATE TABLE" on page 209.

- ◆ Watcom SQL allows **automatic joins** between tables. In addition to the NATURAL and OUTER join operators supported in other implementations, Watcom SQL allows KEY joins between tables based on foreign key relationships. This reduces the complexity of the WHERE clause when performing joins.
- ◆ Watcom SQL allows more than one table to be referenced by the UPDATE command. Views defined on more than one table can also be updated. Many SQL implementations will not allow updates on joined tables.
- ◆ The ALTER TABLE command has been extended. In addition to changes for entity and referential integrity, the following types of alterations are allowed:

```

ADD      column data-type
MODIFY   column data-type
DELETE   column
RENAME   new-table-name
RENAME   old-column TO new-column
    
```

The MODIFY can be used to change the maximum length of a character column as well as converting from one data type to another. For more information, see "ALTER TABLE" on page 187.

- ◆ Watcom SQL allows subqueries to appear wherever expressions are allowed. Many SQL implementations only allow subqueries on the right side of a comparison operator. For example, the following command is valid in Watcom SQL but not valid in most other SQL implementations.

```

SELECT emp_lname, emp_birthdate,
       (SELECT Skill from Department
        WHERE emp_ID = Employee.emp_ID
        AND Department = 200)
FROM Employee
    
```

- ◆ Watcom SQL supports several functions not in the ANSI SQL definition. See "Functions" on page 71 for a full list of available functions.
- ◆ When using Embedded SQL, cursor positions can be moved arbitrarily on the FETCH statement. Cursors can be moved forward or backward

relative to the current position or a given number of records from the beginning or end of the cursor.

## APPENDIX B

# Limitations

Watcom SQL has been designed with as few limitations as possible. The following limitations do exist, but the memory and disk drive of the microcomputer are more limiting factors in most cases.

Item	Limitation
Database size	2 GB per file, 12 files per database
Number of tables per database	32,767
Number of tables referenced per transaction	No limit
Table size	2 GB—must be in one file
Number of columns per table	999 (using 4K pages)
Number of rows per table	Limited by table size
Row size	Limited by table size
Number of rows per database	Limited by file size
Field size	2 GB
Number of indexes	32,767 per table
Maximum index entry size	No limit



## APPENDIX C

# Database Error Messages

- About this appendix    This appendix lists all of the database error messages that are reported by Watcom SQL. Many of the errors contain the characters %s. These values will be replaced by the parameters to the error message.
- First is an index sorted alphabetically by error message. If you know the error message text but not the number, you can use this list to find the error number and look it up in the list of descriptions. Second is an index of error numbers sorted by SQLSTATE. Following is a complete set of error message descriptions.
- Contents
- ◆ "Alphabetic by error message" on the next page
  - ◆ "Alphabetic by SQLSTATE" on page 363
  - ◆ "Error message descriptions" on page 368

## Alphabetic by error message

Each error has a numeric error code, called the SQLCODE. Negative codes are considered errors while positive codes are warnings. The SQLCODE 0 indicates successful completion.

### SQL Code Error message

-210	%s has the row in %s locked
-134	%s not implemented
-110	'%s' already exists
-120	'%s' already has grant permission
-140	'%s' is an unknown userid
-123	'%s' is not a user group
0	(no message)
-150	aggregate functions not allowed on this statement
-307	all threads are blocked
-125	ALTER clause conflict
-407	an argument passed to a WSQL HLI function was invalid
-298	attempted two active database requests
-98	authentication violation
-160	can only describe a SELECT statement
-127	cannot alter a column in an index
-105	cannot be started -- %s
-157	cannot convert %s to a %s
-269	cannot delete a column referenced in a trigger definition
-270	cannot drop a user that owns procedures in runtime engine
-128	cannot drop a user that owns tables in runtime engine
-183	cannot find index named '%s'
-191	cannot modify column '%s' in table '%s'
-106	cannot open log file %s
-190	cannot update an expression
-212	CHECKPOINT command requires a rollback log
-88	client/server communications protocol mismatch
-113	column %s in foreign key has a different definition than primary key
-149	column '%s' cannot be used unless it is in a GROUP BY
-144	column '%s' found in more than one table -- need a correlation name
-195	column '%s' in table '%s' cannot be NULL
-143	column '%s' not found
-267	COMMIT/ROLLBACK not allowed within atomic operation
-273	COMMIT/ROLLBACK not allowed within trigger actions



-85	communication error
-108	connection not found
-99	connections to database have been disabled
-142	correlation name '%s' not found
-172	cursor already open
-170	cursor has not been declared
-180	cursor not open
-241	database backup not started
-96	database engine already running
-89	database engine not running in multi-user mode
-100	database engine not running
-77	database name not unique
-87	database name required to start engine
-266	database was initialized with an older version of the software
-97	database's page size too big
-231	dblib/database engine version mismatch
-138	dbspace '%s' not found
-306	deadlock detected
-304	disk full transaction rolled back
-121	do not have permission to %s
-78	Dynamic memory exhausted!
-184	error inserting into cursor
-107	error writing to log file
-251	foreign key '%s' for table '%s' duplicates an existing foreign key
-145	foreign key name '%s' not found
-305	I/O error %s transaction rolled back
-250	identifier '%s' too long
-242	incomplete transactions prevent transaction log renaming
-196	index '%s' for table '%s' would not be unique
-111	index name '%s' not unique
-199	INSERT/DELETE on cursor can modify only one table
-301	internal database error %s transaction rolled back
-263	invalid absolute or relative offset in FETCH
-159	invalid column number
103	invalid data conversion
-81	invalid database engine command line
-156	invalid expression near '%s'
-155	invalid host variable
-79	invalid local database switch
-187	invalid operation for this cursor
-192	invalid operation on joined tables

-200	invalid option '%s' -- no PUBLIC setting exists
-95	invalid parameter
-133	invalid prepared statement type
-272	invalid REFERENCES clause in trigger definition
-201	invalid setting for option '%s'
-130	invalid statement
-161	invalid type on DESCRIBE statement
-104	invalid userid and password on preprocessed module
-103	invalid userid or password
-209	invalid value for column '%s' in table '%s'
-405	invalid WSQL HLI callback function
-400	invalid WSQL HLI command syntax
-401	invalid WSQL HLI cursor name
-403	invalid WSQL HLI host variable name
-404	invalid WSQL HLI host variable value
-402	invalid WSQL HLI statement name
-262	label '%s' not found
-135	language extension
-139	more than one table is identified as '%s'
-197	no current row of cursor
-181	no indicator variable provided for NULL result
-194	no primary key value for foreign key '%s' in table '%s'
-211	not allowed while %s is using the database
-101	not connected to SQL database
-182	not enough fields allocated in SQLDA
-86	not enough memory to start
-188	not enough values for host variables
-152	number in ORDER BY is too large
-114	number of columns does not match SELECT
-122	operation would cause a group cycle
-119	primary key column '%s' already defined
-198	primary key for row in table '%s' is referenced in another table
-193	primary key for table '%s' is not unique
-265	procedure '%s' not found
105	procedure has completed
-215	procedure in use
-274	procedure or trigger calls have nested too deeply
-76	request denied -- no active databases
-75	request to start/stop database denied
-222	result set not allowed from within an atomic compound statement
-221	ROLLBACK TO SAVEPOINT not allowed

104	row has been updated since last time read
-208	row has changed since last read -- operation cancelled
100	row not found
-300	run time SQL error -- %s
-220	savepoint '%s' not found
-213	savepoints require a rollback log
-153	SELECT lists in UNION do not match in length
-185	SELECT returns more than one row
-232	server/database engine version mismatch
-84	specified database is invalid
-83	specified database not found
-132	SQL statement error
-230	sqlpp/dblib version mismatch
-299	statement interrupted by user
-151	subquery allowed only one select list item
-186	subquery cannot return more than one result
-131	syntax error near '%s'
-118	table '%s' has no primary key
-141	table '%s' not found
-112	table already has a primary key
-126	table cannot have two primary keys
-214	table in use
-116	table must be empty
-302	terminated by user transaction rolled back
-74	the selected database is currently inactive
400	the supplied buffer was too small to hold all requested query results
-109	there are still active database connections
-261	there is already a variable named '%s'
-147	there is more than one way to join '%s' to '%s'
-146	there is no way to join '%s' to '%s'
-102	too many connections to database
-268	trigger '%s' not found
-271	trigger definition conflicts with existing triggers
-243	unable to delete database file
-189	unable to find in index '%s' for table '%s'
-80	unable to start database engine
-82	unable to start specified database
-240	unknown backup operation
-148	unknown function '%s'
-297	user-defined exception signalled
102	using temporary table

*Alphabetic by error message*

---

<b>-158</b>	value %s too large for destination
<b>101</b>	value truncated
<b>-260</b>	variable '%s' not found
<b>200</b>	warning
<b>-154</b>	wrong number of parameters to function '%s'
<b>-207</b>	wrong number of values for INSERT
<b>-264</b>	wrong number of variables in FETCH
<b>-406</b>	WSQL HLI internal error

## Alphabetic by SQLSTATE

Watcom SQL supports a second error code, called SQLSTATE, defined by ANSI SQL-92. Each SQLSTATE value is a five character string containing a two character class followed by a three character subclass. Each character can be one of the uppercase letters A through Z or the digits 0 through 9. A class that begins with A through H or 0 through 4 has been defined by the ANSI standard; other classes are implementation defined. Similarly, subclasses of standard classes that start with the same characters (A-H, 0-4) are standard. The subclass 000 always means that no subclass code is defined. The most common SQLSTATE value is 00000 which indicates successful completion.

In Watcom SQL, there are two ways to get SQLSTATE values. In Embedded SQL, the SQLSTATE is returned in the SQLCA. In ODBC, the *SQLError* function returns SQLSTATE values. Since the ODBC definition explicitly specifies which SQLSTATE values are to be returned, some ODBC SQLSTATE values correspond to more than one Watcom SQL error condition. Consequently, many error conditions return one SQLSTATE value in Embedded SQL which precisely indicates the type of error, and return another less specific SQLSTATE value in ODBC. If the ODBC SQLSTATE value is different than that in Embedded SQL, it is explicitly specified in the error list.

In Embedded SQL, the SQLSTATE values are defined as constants in the file SQLSTATE.H in the H subdirectory. The descriptions below indicate the constant name for the SQLCODE for each state which begins with the characters 'SQLE\_'. The corresponding SQLSTATE values begin with the characters 'SQLSTATE\_'.

The SQLSTATE values used by Watcom SQL have been derived from a number of sources. The class of the error is determined by:

- 1 Using a class defined by SQL-92 if it exists
- 2 Otherwise, using a class defined by IBM's SAA (starts with 5)
- 3 Otherwise, using a class defined by WATCOM (starts with W)

The same rules are used for defining subclasses. In other words, a class or subclass starting with 0-4 or A-H are defined by ANSI, starting with 5 are defined by IBM, and starting with W are defined by WATCOM. Similarly, the ODBC SQLSTATE classes and subclasses starting with S have been defined by Microsoft in the ODBC specification.

**SQL Code SQLSTATE**

<b>0</b>	00000
<b>200</b>	01000
<b>101</b>	01004
<b>102</b>	01W02
<b>103</b>	01W03
<b>104</b>	01W04
<b>400</b>	01W05
<b>105</b>	01W06
<b>100</b>	02000
<b>-188</b>	07001
<b>-182</b>	07002
<b>-160</b>	07005
<b>-161</b>	07W01
<b>-130</b>	07W02
<b>-133</b>	07W03
<b>-105</b>	08001
<b>-101</b>	08003
<b>-140</b>	08004
<b>-100</b>	08W01
<b>-108</b>	08W02
<b>-102</b>	08W03
<b>-99</b>	08W04
<b>-106</b>	08W05
<b>-109</b>	08W06
<b>-80</b>	08W07
<b>-81</b>	08W08
<b>-82</b>	08W09
<b>-83</b>	08W10
<b>-84</b>	08W11
<b>-85</b>	08W12
<b>-86</b>	08W13
<b>-87</b>	08W14
<b>-88</b>	08W15
<b>-89</b>	08W16
<b>-107</b>	08W17
<b>-230</b>	08W18
<b>-231</b>	08W19
<b>-232</b>	08W20
<b>-98</b>	08W21
<b>-97</b>	08W22

<b>-96</b>	08W23
<b>-95</b>	08W24
<b>-79</b>	08W25
<b>-78</b>	08W26
<b>-77</b>	08W27
<b>-76</b>	08W28
<b>-75</b>	08W29
<b>-74</b>	08W30
<b>-184</b>	09W01
<b>-187</b>	09W02
<b>-197</b>	09W03
<b>-199</b>	09W04
<b>-134</b>	0A000
<b>-135</b>	0AW01
<b>-185</b>	21000
<b>-186</b>	21W01
<b>-181</b>	22002
<b>-158</b>	22003
<b>-407</b>	22W01
<b>-208</b>	22W02
<b>-195</b>	23502
<b>-194</b>	23503
<b>-196</b>	23505
<b>-209</b>	23506
<b>-193</b>	23W01
<b>-198</b>	23W05
<b>-180</b>	24501
<b>-172</b>	24502
<b>-170</b>	24W01
<b>-132</b>	26501
<b>-402</b>	26W01
<b>-103</b>	28000
<b>-104</b>	28W01
<b>-273</b>	2D501
<b>-401</b>	34W01
<b>-154</b>	37505
<b>-220</b>	3B001
<b>-221</b>	3B002
<b>-213</b>	3BW01
<b>-222</b>	3BW02
<b>-300</b>	40000
<b>-306</b>	40001

<b>-301</b>	40W01
<b>-302</b>	40W02
<b>-304</b>	40W03
<b>-305</b>	40W04
<b>-307</b>	40W06
<b>-121</b>	42501
<b>-120</b>	42W01
<b>-122</b>	42W02
<b>-123</b>	42W03
<b>-131</b>	42W04
<b>-148</b>	42W05
<b>-150</b>	42W06
<b>-155</b>	42W07
<b>-156</b>	42W08
<b>-403</b>	42W09
<b>-404</b>	42W10
<b>-400</b>	42W11
<b>-405</b>	42W12
<b>-159</b>	42W13
<b>-260</b>	42W14
<b>-261</b>	42W15
<b>-200</b>	42W16
<b>-201</b>	42W17
<b>-210</b>	42W18
<b>-211</b>	42W19
<b>-212</b>	42W20
<b>-214</b>	42W21
<b>-298</b>	42W22
<b>-215</b>	42W23
<b>-262</b>	42W24
<b>-263</b>	42W25
<b>-264</b>	42W26
<b>-266</b>	42W27
<b>-267</b>	42W28
<b>-274</b>	42W29
<b>-144</b>	52002
<b>-143</b>	52003
<b>-119</b>	52009
<b>-110</b>	52010
<b>-139</b>	52012
<b>-141</b>	52W01
<b>-142</b>	52W02



<b>-183</b>	52W03
<b>-111</b>	52W04
<b>-126</b>	52W05
<b>-251</b>	52W06
<b>-145</b>	52W07
<b>-147</b>	52W08
<b>-265</b>	52W09
<b>-268</b>	52W10
<b>-271</b>	52W11
<b>-272</b>	52W12
<b>-138</b>	52W13
<b>-207</b>	53002
<b>-149</b>	53003
<b>-152</b>	53005
<b>-191</b>	53008
<b>-114</b>	53011
<b>-157</b>	53018
<b>-151</b>	53023
<b>-153</b>	53026
<b>-113</b>	53030
<b>-125</b>	53W01
<b>-190</b>	53W02
<b>-192</b>	53W03
<b>-146</b>	53W04
<b>-127</b>	53W05
<b>-269</b>	53W06
<b>-250</b>	54003
<b>-118</b>	55008
<b>-112</b>	55013
<b>-116</b>	55W02
<b>-128</b>	55W03
<b>-270</b>	55W04
<b>-299</b>	57014
<b>-297</b>	99999
<b>-240</b>	WB001
<b>-241</b>	WB002
<b>-242</b>	WB003
<b>-243</b>	WB004
<b>-189</b>	WI005
<b>-406</b>	WI007

## Error message descriptions

### Warnings

(no message)

**SQL Code** 0  
**SQL State** 00000  
**Probable cause**

This code indicates that there was no error or warning.

row not found

**SQL Code** 100  
**SQL State** 02000  
**ODBC state** (handled by ODBC driver)  
**Probable cause**

You have positioned a cursor beyond the beginning or past the end of the query. There is no row at that position.

value truncated

**SQL Code** 101  
**SQL State** 01004  
**Probable cause**

You have tried to insert, update, or select a value in the database which is too large to fit in the destination. This warning is also produced if you do a fetch, and the host variable or SQLDA variable is not large enough to receive the value.

using temporary table

**SQL Code** 102  
**SQL State** 01W02  
**ODBC state** (handled by ODBC driver)  
**Probable cause**

A temporary table has been created in order to satisfy the query. It can only occur on an OPEN statement.

**invalid data conversion**

**SQL Code** 103  
**SQL State** 01W03  
**ODBC state** 07006

**Probable cause**

The database could not convert a value to the required type. This is either a value supplied to the database on an insert, update or as a host bind variable, or a value retrieved from the database into a host variable or SQLDA.

**row has been updated since last time read**

**SQL Code** 104  
**SQL State** 01W04  
**ODBC state** (handled by ODBC driver)

**Probable cause**

A FETCH has retrieved a row from a cursor declared as a SCROLL cursor, and the row was previously fetched from the same cursor, and one or more columns in the row has been updated since the previous fetch. Note that the column(s) updated may or may not be fetched by the cursor; this warning just indicates that the row from the table has been updated. If the cursor involves more than one table, a row from one or more of the tables has been updated.

**procedure has completed**

**SQL Code** 105  
**SQL State** 01W06  
**ODBC state** (handled by ODBC driver)

**Probable cause**

An OPEN or a RESUME has caused a procedure to execute to completion. There are no more result sets available from this procedure. This warning will also be returned if you attempt to RESUME a cursor on a SELECT statement.

**warning**

**SQL Code** 200  
**SQL State** 01000  
**ODBC state** (handled by ODBC driver)

**Probable cause**

A warning has occurred. The warning message will indicate the condition that caused the warning.

**the supplied buffer was too small to hold all requested query results**

**SQL Code** 400  
**SQL State** 01W05  
**ODBC state** (handled by ODBC driver)  
**Probable cause**

You attempted to get a query result set using the WSQL HLI function *wsqquerytomem*. The buffer supplied by the calling application was too small to contain the entire query. The buffer will contain as many rows of the result set as possible, and the cursor will be positioned on the next row of the result set.

## Environment errors

**unable to start database engine**

**SQL Code** -80  
**SQL State** 08W07  
**ODBC state** 08001  
**Probable cause**

It was not possible to start the database engine or multi-user client. Either there is not enough memory to run the database engine, or the executable cannot be found. See "Environment variables" on page 346 for a description of how the engine is started.

**invalid database engine command line**

**SQL Code** -81  
**SQL State** 08W08  
**ODBC state** 08001  
**Probable cause**

It was not possible to start the database engine or multi-user client because the command line was invalid. See "Environment variables" on page 346 for a description of how the engine is started.

**the selected database is currently inactive**

**SQL Code** -74  
**SQL State** 08W30  
**ODBC state** 08001  
**Probable cause**

The selected database is in an inactive state. This state occurs during database initialization and shutdown.

**request to start/stop database denied**

**SQL Code** -75  
**SQL State** 08W29  
**ODBC state** 08001  
**Probable cause**

The engine has denied permission to start/stop a database.

**request denied -- no active databases**

**SQL Code** -76  
**SQL State** 08W28  
**ODBC state** 08001  
**Probable cause**

The engine has denied the request as there are currently no loaded databases.

**database name not unique**

**SQL Code** -77  
**SQL State** 08W27  
**ODBC state** 08001  
**Probable cause**

The database cannot be loaded as its name is conflicting with a previously loaded database.

**invalid local database switch**

**SQL Code** -79  
**SQL State** 08W25  
**ODBC state** 08001  
**Probable cause**

An invalid local database switch was found in the DBS option.

**unable to start specified database**

**SQL Code** -82  
**SQL State** 08W09  
**ODBC state** 08001

**Probable cause**

The database engine or multi-user client was started but was unable to find the specified database or server name. No specific reason is known.

**specified database not found**

**SQL Code** -83  
**SQL State** 08W10  
**ODBC state** 08001

**Probable cause**

The database engine or multi-user client was started but was unable to find the specified database or server name. The database file cannot be opened or the specified server cannot be found on the network. The database engine or client is stopped.

**specified database is invalid**

**SQL Code** -84  
**SQL State** 08W11  
**ODBC state** 08001

**Probable cause**

The database engine was started but the specified database file is invalid. The engine is stopped.

**communication error**

**SQL Code** -85  
**SQL State** 08W12  
**ODBC state** 08S01

**Probable cause**

There is a communication problem between the multi-user client and server. This happens most frequently when the multi-user client was unable to start because a communication error occurred while trying to locate the server.

**Dynamic memory exhausted!**

**SQL Code** -78  
**SQL State** 08W26  
**ODBC state** S1001  
**Probable cause**

A failure occurred when trying to allocate dynamic memory.

**not enough memory to start**

**SQL Code** -86  
**SQL State** 08W13  
**ODBC state** S1001  
**Probable cause**

The database engine or multi-user client executable was loaded but was unable to start because there is not enough memory to run properly.

**database name required to start engine**

**SQL Code** -87  
**SQL State** 08W14  
**ODBC state** 08001  
**Probable cause**

A database name is required to start the database engine or the multi-user client, but it was not specified.

**client/server communications protocol mismatch**

**SQL Code** -88  
**SQL State** 08W15  
**ODBC state** 08S01  
**Probable cause**

The multi-user client was unable to start because the protocol versions of the client and the running server do not match. Make sure the client and server software are the same version.

**database engine already running**

**SQL Code** -96  
**SQL State** 08W23  
**ODBC state** S1000

### **database engine not running in multi-user mode**

**SQL Code** -89  
**SQL State** 08W16  
**ODBC state** 08001  
**Probable cause**

The database was started for bulk loading (the -b switch) and cannot be used as a multi-user engine. Stop the database, and start again without -b. In the DOS version of Watcom SQL Version 3.0, the database engine was not started in multi-user mode.

### **Connection errors**

#### **invalid parameter**

**SQL Code** -95  
**SQL State** 08W24  
**ODBC state** 08004  
**Probable cause**

An error occurred while parsing the string parameter associated with one of the entry points: db\_start\_engine(), db\_start\_database(), db\_stop\_engine(), db\_stop\_database(), db\_string\_connect().

#### **database's page size too big**

**SQL Code** -97  
**SQL State** 08W22  
**ODBC state** 08004  
**Probable cause**

You have attempted to start a database with a page size that exceeds the maximum page size of the running engine. Restart the engine with this database named on the commandline.

#### **authentication violation**

**SQL Code** -98



**SQL State** 08W21  
**ODBC state** 08001  
**Probable cause**

You have attempted to connect to an engine or server which has been authenticated for exclusive use with a specific application.

**connections to database have been disabled**

**SQL Code** -99  
**SQL State** 08W04  
**ODBC state** 08005  
**Probable cause**

Connections to the multi-user server have been disabled on the server console. You will receive this error until they have been reenabled on the server console.

**database engine not running**

**SQL Code** -100  
**SQL State** 08W01  
**ODBC state** 08001  
**Probable cause**

You have not run the database engine or network requestor or the interface library is unable to find it.

**not connected to SQL database**

**SQL Code** -101  
**SQL State** 08003  
**Probable cause**

You have not connected to the database, or you have executed the DISCONNECT command and have not connected to the database again.

**too many connections to database**

**SQL Code** -102  
**SQL State** 08W03  
**ODBC state** 08004  
**Probable cause**

If you are running the multi-user client, you have exceeded the number of computers allowed to connect to the server by your license agreement. Otherwise, the single user DOS

engine is limited to 2 connections, and the Windows engine is restricted to 10 connections.

**invalid userid or password**

**SQL Code** -103  
**SQL State** 28000  
**Probable cause**

The user has supplied an invalid userid or an incorrect password. ISQL will handle this error by presenting a connection dialog to the user.

**invalid userid and password on preprocessed module**

**SQL Code** -104  
**SQL State** 28W01  
**ODBC state** 28000  
**Probable cause**

A userid and password were specified when a module was preprocessed but the userid or password is invalid.

**cannot be started -- %s**

**SQL Code** -105  
**SQL State** 08001  
**Parameter** Name of database.  
**Probable cause**

The specified database environment cannot be found. If it is a database name, then it does not exist, it is not a database, it is corrupt, or it is an older format. If it is a server name, then the server cannot be found.

**cannot open log file %s**

**SQL Code** -106  
**SQL State** 08W05  
**ODBC state** 08003  
**Parameter** Name of log file.  
**Probable cause**

The database engine was unable to open the transaction log file. Perhaps the log file name specifies an invalid device or directory. If this is the case, you can use the DBLOG utility to find out where the transaction log should be and perhaps change it.

**error writing to log file**

**SQL Code** -107  
**SQL State** 08W17  
**ODBC state** S1000  
**Probable cause**

The database engine got an I/O error writing the log file. Perhaps the disk is full or the log file name is invalid.

**connection not found**

**SQL Code** -108  
**SQL State** 08W02  
**ODBC state** 08003  
**Probable cause**

The specified connection name on a DISCONNECT or SET CONNECTION statement is invalid.

**there are still active database connections**

**SQL Code** -109  
**SQL State** 08W06  
**ODBC state** S1000  
**Probable cause**

An application has requested Watcom SQL to shutdown the database using the *db\_stop()* function when there are still active connections to the database.

**Creation errors**

**'%s' already exists**

**SQL Code** -110  
**SQL State** 52010  
**ODBC state** S0001  
**Parameter** Name of the item that already exists.  
**Probable cause**

You have tried to create a table, view, column, or foreign key with the same name as an existing one.

**index name '%s' not unique**

**SQL Code** -111  
**SQL State** 52W04  
**ODBC state** S0011  
**Parameter** Name of the invalid index.  
**Probable cause**

You have attempted to create an index with a name of an existing index.

**table already has a primary key**

**SQL Code** -112  
**SQL State** 55013  
**ODBC state** 23000  
**Probable cause**

You have tried to add a primary key on a table that already has a primary key defined. You must delete the current primary key before adding a new one.

**column %s in foreign key has a different definition than primary key**

**SQL Code** -113  
**SQL State** 53030  
**ODBC state** 23000  
**Parameter** Name of the problem column.  
**Probable cause**

The data type of the column in the foreign key is not the same as the data type of the column in the primary key. Change the definition of one of the columns using ALTER TABLE.

**number of columns does not match SELECT**

**SQL Code** -114  
**SQL State** 53011  
**ODBC state** 21S01  
**Probable cause**

An INSERT command contains a SELECT with a different number of columns than the INSERT.

**table must be empty**

**SQL Code** -116

**SQL State** 55W02

**ODBC state** S1000

**Probable cause**

You have attempted to modify a table, and Watcom SQL can only perform the change if there are no rows in the table.

**table '%s' has no primary key**

**SQL Code** -118

**SQL State** 55008

**ODBC state** 23000

**Parameter** Name of the table that does not have a primary key.

**Probable cause**

You have attempted to add a foreign key referring to a table that does not have a primary key. You will need to add a primary key to the named table.

**primary key column '%s' already defined**

**SQL Code** -119

**SQL State** 52009

**ODBC state** 23000

**Parameter** Name of the column that is already in the primary key.

**Probable cause**

You have listed the same column name twice in the definition of a primary key.

**ALTER clause conflict**

**SQL Code** -125

**SQL State** 53W01

**ODBC state** S1000

**Probable cause**

A primary key clause, foreign key clause, or a uniqueness clause must be the only clause of an ALTER TABLE command.

**table cannot have two primary keys**

**SQL Code** -126

**SQL State** 52W05

**ODBC state** 23000

**Probable cause**

You have specified the primary key twice in a CREATE TABLE command.

**cannot alter a column in an index**

**SQL Code** -127  
**SQL State** 53W05  
**ODBC state** S1000  
**Probable cause**

This error is reported if you attempt to delete or modify the definition of a column that is part of a primary or foreign key. This error will also be reported if you attempt to delete a column that has an index on it. DROP the index or key, perform the ALTER command, and then add the index or key again.

**cannot drop a user that owns tables in runtime engine**

**SQL Code** -128  
**SQL State** 55W03  
**ODBC state** 37000  
**Probable cause**

This error is reported by the runtime engine if you attempt to drop a user that owns tables. Because this operation would result in dropping tables, and the runtime engine cannot drop tables, it is not allowed. Use the development engine.

**identifier '%s' too long**

**SQL Code** -250  
**SQL State** 54003  
**ODBC state** 37000  
**Parameter** The identifier in error.  
**Probable cause**

An identifier is longer than 128 characters.

**foreign key '%s' for table '%s' duplicates an existing foreign key**

**SQL Code** -251  
**SQL State** 52W06  
**ODBC state** 23000  
**Parameter** The role name of the new foreign key.

**Parameter** The table containing the foreign key.  
**Probable cause**

You have attempted to define a foreign key that already exists.

**cannot find index named '%s'**

**SQL Code** -183  
**SQL State** 52W03  
**ODBC state** S0012  
**Parameter** Name of the index that cannot be found.  
**Probable cause**

A DROP INDEX command has named an index that does not exist. Check for spelling errors or whether the index name must be qualified by a userid.

## Permission errors

**'%s' already has grant permission**

**SQL Code** -120  
**SQL State** 42W01  
**ODBC state** 37000  
**Parameter** Name of the userid that already has GRANT permission.  
**Probable cause**

The SQL GRANT command is attempting to give a user GRANT OPTION and that user already has GRANT OPTION.

**do not have permission to %s**

**SQL Code** -121  
**SQL State** 42501  
**ODBC state** 42001  
**Parameter** Description of the type of permission lacking.  
**Probable cause**

You have not been granted permission to use a table belonging to another userid.

**operation would cause a group cycle**

**SQL Code** -122

**SQL State** 42W02

**ODBC state** 37000

**Probable cause**

You have tried to add a member to group that would result in a member belonging to itself (perhaps indirectly).

**'%s' is not a user group**

**SQL Code** -123

**SQL State** 42W03

**ODBC state** 37000

**Parameter** Name of user you thought was a group.

**Probable cause**

You have tried to add a member to group, but the group specified has not been granted the GROUP special privilege.

## **Prepare errors**

### **invalid statement**

**SQL Code** -130

**SQL State** 07W02

**ODBC state** S1000

**Probable cause**

The statement identifier (generated by PREPARE) passed to the database for a further operation is invalid.

### **syntax error near '%s'**

**SQL Code** -131

**SQL State** 42W04

**ODBC state** 37000

**Parameter** The word or symbol where the syntax error has been detected.

**Probable cause**

The database engine cannot understand the command you are trying to execute. If you have used a keyword (such as DATE) for a column name, try enclosing the keyword in quotation marks ("DATE").



**SQL statement error**

**SQL Code** -132  
**SQL State** 26501  
**ODBC state** S1000  
**Probable cause**

The statement identifier (generated by PREPARE) passed to the database for a further operation is invalid.

**invalid prepared statement type**

**SQL Code** -133  
**SQL State** 07W03  
**ODBC state** S1000  
**Probable cause**

This is an internal C language interface error. If it occurs, it should be reported to Watcom.

**%s not implemented**

**SQL Code** -134  
**SQL State** 0A000  
**ODBC state** S1000  
**Parameter** The unimplemented feature.  
**Probable cause**

The requested operation or feature is not implemented in Watcom SQL.

**language extension**

**SQL Code** -135  
**SQL State** 0AW01  
**ODBC state** S1000  
**Probable cause**

The requested operation is valid in some versions of SQL, but not in Watcom SQL.

**Semantic errors**

**dbspace '%s' not found**

**SQL Code** -138

**SQL State** 52W13  
**ODBC state** S0002  
**Parameter** Name of the dbspace that could not be found.  
**Probable cause**  
The named dbspace was not found.

**more than one table is identified as '%s'**

**SQL Code** -139  
**SQL State** 52012  
**ODBC state** SG001  
**Parameter** Ambiguous correlation name.  
**Probable cause**  
You have identified two tables in the same FROM clause with the same correlation name.

**'%s' is an unknown userid**

**SQL Code** -140  
**SQL State** 08004  
**ODBC state** 28000  
**Parameter** Name of the userid that could not be found.  
**Probable cause**  
The specified userid does not exist.

**table '%s' not found**

**SQL Code** -141  
**SQL State** 52W01  
**ODBC state** S0002  
**Parameter** Name of the table that could not be found.  
**Probable cause**  
You have misspelled the name of a table, or you have connected with a different userid and forgotten to qualify a table name with a user name. For example, you might have connected to userid 'Teacher' and tried to refer to **Student** instead of **admin.Student**.

**correlation name '%s' not found**

**SQL Code** -142  
**SQL State** 52W02  
**ODBC state** S0002  
**Parameter** Name of the invalid correlation name.

**Probable cause**

You have misspelled a correlation name, or you have used a table name instead of the correlation name.

**column '%s' not found**

**SQL Code** -143  
**SQL State** 52003  
**ODBC state** S0022  
**Parameter** Name of the column that could not be found.

**Probable cause**

You have misspelled the name of a column, or the column you are looking for is in a different table.

**column '%s' found in more than one table -- need a correlation name**

**SQL Code** -144  
**SQL State** 52002  
**ODBC state** SJS01  
**Parameter** Name of the ambiguous column.

**Probable cause**

You have not put a correlation name on a column that is found in more than one of the tables referenced in a query. You need to add a correlation name to the reference.

**foreign key name '%s' not found**

**SQL Code** -145  
**SQL State** 52W07  
**ODBC state** 37000  
**Parameter** Name of the non-existing foreign key.

**Probable cause**

You have misspelled the name of a foreign key or the foreign key does not exist.

**there is no way to join '%s' to '%s'**

**SQL Code** -146  
**SQL State** 53W04  
**ODBC state** 37000  
**Parameter** Name of first table that cannot be joined.  
**Parameter** Name of second table that cannot be joined.

**Probable cause**

You have attempted a KEY JOIN between two tables and

there is no foreign key on one of the tables that references the primary key of the other table; or you have attempted a NATURAL JOIN between two tables and the tables have no common column names.

**there is more than one way to join '%s' to '%s'**

**SQL Code** -147  
**SQL State** 52W08  
**ODBC state** 37000  
**Parameter** Name of first table that cannot be joined.  
**Parameter** Name of second table that cannot be joined.  
**Probable cause**

There are two or more foreign keys relating the two tables and you are attempting to KEY JOIN the two tables. Either there are two foreign keys from the first table to the second table, or each table has a foreign key to the other table. You must use a correlation name for the primary key table which is the same as the role name of the desired foreign key relationship.

**unknown function '%s'**

**SQL Code** -148  
**SQL State** 42W05  
**ODBC state** 37000  
**Parameter** Function name that is not a database function.  
**Probable cause**

You have misspelled the name of a database function (such as MAXIMUM instead of MAX) in a query definition or in a query column name. Refer to "Functions" on page 71 for the correct function name.

**column '%s' cannot be used unless it is in a GROUP BY**

**SQL Code** -149  
**SQL State** 53003  
**ODBC state** 37000  
**Parameter** Name of the column that must be in the GROUP BY.  
**Probable cause**

You have forgotten to put a column name in a GROUP BY. In some cases, you may want to use the MAX function on the column name instead of adding the column to the GROUP BY list.

**aggregate functions not allowed on this statement**

**SQL Code** -150  
**SQL State** 42W06  
**ODBC state** 37000  
**Probable cause**

An UPDATE statement has used an aggregate function (MIN, MAX, SUM, AVG or COUNT).

**subquery allowed only one select list item**

**SQL Code** -151  
**SQL State** 53023  
**ODBC state** 37000  
**Probable cause**

You have entered a subquery which has more than one column in the select list. Change the select list to have only one column.

**number in ORDER BY is too large**

**SQL Code** -152  
**SQL State** 53005  
**ODBC state** 37000  
**Probable cause**

You have used an integer in an ORDER BY list and the integer is larger than the number of columns in the select list.

**SELECT lists in UNION do not match in length**

**SQL Code** -153  
**SQL State** 53026  
**ODBC state** 37000  
**Probable cause**

You have specified a UNION but the SELECT statements involved in the union do not have the same number of columns in the select list.

## Expression and function errors

### wrong number of parameters to function '%s'

**SQL Code** -154  
**SQL State** 37505  
**ODBC state** 37000  
**Parameter** Name of the function.  
**Probable cause**

You have supplied an incorrect number of parameters to a database function. Refer to "Functions" on page 71 for the correct number of parameters.

### invalid host variable

**SQL Code** -155  
**SQL State** 42W07  
**ODBC state** 37000  
**Probable cause**

A host variable supplied to the database using the C language interface as either a host variable or through an SQLDA is invalid.

### invalid expression near '%s'

**SQL Code** -156  
**SQL State** 42W08  
**ODBC state** 37000  
**Parameter** The invalid expression.  
**Probable cause**

You have an expression which the database engine cannot understand. For example, you might have tried to add two dates.

### cannot convert %s to a %s

**SQL Code** -157  
**SQL State** 53018  
**ODBC state** 07006  
**Parameter** The value that could not be converted.  
**Parameter** The name of the type for the conversion.  
**Probable cause**

An invalid value has been supplied to or fetched from the

database. For example, the value '12X' might have been supplied where a number was required.

**value %s too large for destination**

**SQL Code** -158  
**SQL State** 22003  
**Parameter** The value that caused the overflow.  
**Probable cause**

A value has been supplied to the database or retrieved from the database that is too large for the destination column or host variable. For example, the value '10' may have been supplied for a DECIMAL(3,2) field.

**invalid column number**

**SQL Code** -159  
**SQL State** 42W13  
**ODBC state** S1000  
**Probable cause**

The column number in a GET DATA command is invalid.

## Describe errors

**can only describe a SELECT statement**

**SQL Code** -160  
**SQL State** 07005  
**ODBC state** (handled by ODBC driver)  
**Probable cause**

In the C language interface, you attempted to describe the select list of a statement other than a SELECT statement.

**invalid type on DESCRIBE statement**

**SQL Code** -161  
**SQL State** 07W01  
**ODBC state** (handled by ODBC driver)  
**Probable cause**

This is an internal C language interface error. If it occurs, it should be reported to Watcom.

## Open errors

### cursor has not been declared

**SQL Code** -170  
**SQL State** 24W01  
**ODBC state** 24000  
**Probable cause**

You attempted to OPEN a cursor that has not been declared.

### cursor already open

**SQL Code** -172  
**SQL State** 24502  
**ODBC state** 24000  
**Probable cause**

You attempted to OPEN a cursor that is already open.

## Fetch errors

### cursor not open

**SQL Code** -180  
**SQL State** 24501  
**ODBC state** 34000  
**Probable cause**

You attempted to OPEN a cursor that has not been declared.

### no indicator variable provided for NULL result

**SQL Code** -181  
**SQL State** 22002  
**ODBC state** S1000  
**Probable cause**

You tried to retrieve a value from the database that was the NULL value but you did not provide an indicator variable for that value.



**not enough fields allocated in SQLDA**

**SQL Code** -182  
**SQL State** 07002  
**Probable cause**

There are not enough fields in the SQLDA to retrieve all of the values requested.

**error inserting into cursor**

**SQL Code** -184  
**SQL State** 09W01  
**ODBC state** S1000  
**Probable cause**

An error has occurred while inserting into a cursor.

**SELECT returns more than one row**

**SQL Code** -185  
**SQL State** 21000  
**ODBC state** S1000  
**Probable cause**

An Embedded SELECT statement that does not use a cursor returns more than one result.

**subquery cannot return more than one result**

**SQL Code** -186  
**SQL State** 21W01  
**ODBC state** 37000  
**Probable cause**

The result of a subquery contains more than one row. If the subquery is in the WHERE clause, you might be able to use IN.

**invalid operation for this cursor**

**SQL Code** -187  
**SQL State** 09W02  
**ODBC state** 24000  
**Probable cause**

An operation that is not allowed was attempted on a cursor.

### **not enough values for host variables**

**SQL Code** -188  
**SQL State** 07001  
**Probable cause**

You have not provided enough host variables for either the number of bind variables, or the command, or the number of select list items.

### **unable to find in index '%s' for table '%s'**

**SQL Code** -189  
**SQL State** WI005  
**ODBC state** S1000  
**Parameter** Name of invalid index.  
**Parameter** Name of table containing the invalid index.  
**Probable cause**

This is a Watcom SQL internal error and should be reported to Watcom. You should be able to work around the error by dropping and recreating the index.

## **Update and insert errors**

### **cannot update an expression**

**SQL Code** -190  
**SQL State** 53W02  
**ODBC state** 37000  
**Probable cause**

You have tried to update a column in a query that is a database expression rather than a column in a table.

### **cannot modify column '%s' in table '%s'**

**SQL Code** -191  
**SQL State** 53008  
**ODBC state** 37000  
**Parameter** Name of the column that cannot be changed.  
**Parameter** Name of the table containing the column.  
**Probable cause**

You do not have permission to modify the column, or the table is actually a view and the column in the view is

defined as an expression (such as  
'column1+column2') that cannot be modified.

**invalid operation on joined tables**

**SQL Code** -192  
**SQL State** 53W03  
**ODBC state** 37000  
**Probable cause**

You have tried to delete from a query involving more than one table.

**primary key for table '%s' is not unique**

**SQL Code** -193  
**SQL State** 23W01  
**ODBC state** 23000  
**Parameter** Name of the table where the problem was detected.  
**Probable cause**

You have tried to add a new row to a table where the new row has the same primary key as an existing row. The database has not added the incorrect row to the database. For example, you might have added a student with student number 86004 and there is already a row for a student with that number.

**no primary key value for foreign key '%s' in table '%s'**

**SQL Code** -194  
**SQL State** 23503  
**ODBC state** 23000  
**Parameter** Name of the foreign key.  
**Parameter** Name of the table with the foreign key.  
**Probable cause**

You have tried to insert or update a row that has a foreign key for another table, and the value for the foreign key is not NULL and there is not a corresponding value in the primary key

**column '%s' in table '%s' cannot be NULL**

**SQL Code** -195  
**SQL State** 23502  
**ODBC state** 23000

**Parameter** Name of the column that cannot be NULL.  
**Parameter** Name of the table containing the column.  
**Probable cause**  
You have not supplied a value where a value is required.

**index '%s' for table '%s' would not be unique**

**SQL Code** -196  
**SQL State** 23505  
**ODBC state** 23000  
**Parameter** Name of the index that would not be unique.  
**Parameter** Name of the table that contains the index.  
**Probable cause**  
You have inserted or updated a row that has the same value as another row in some column, and there is a constraint that does not allow two rows to have the same value in that column.

**no current row of cursor**

**SQL Code** -197  
**SQL State** 09W03  
**ODBC state** 24000  
**Probable cause**  
You have attempted to perform an operation on the current row of a cursor, but there is no current row. The cursor is before the first row of the cursor, after the last row or is on a row that has since been deleted.

**primary key for row in table '%s' is referenced in another table**

**SQL Code** -198  
**SQL State** 23W05  
**ODBC state** 23000  
**Parameter** The name of the table with a primary key that is referenced.  
**Probable cause**  
You have attempted to delete or modify a primary key that is referenced elsewhere in the database.

**INSERT/DELETE on cursor can modify only one table**

**SQL Code** -199  
**SQL State** 09W04  
**ODBC state** 37000

**Probable cause**

You have attempted to INSERT into a cursor and have specified values for more than one table; or you have tried to DELETE from a cursor that involves a join. INSERT into one table at a time. For DELETE, use the FROM clause to specify which table you wish to delete from.

**invalid value for column '%s' in table '%s'**

**SQL Code** -209  
**SQL State** 23506  
**ODBC state** 23000  
**Parameter** Name of the column that was assigned an invalid value.  
**Parameter** Name of the table containing the column.

**Probable cause**

An INSERT or UPDATE has specified a value for a column that violates a CHECK constraint, and the INSERT or UPDATE were not done because of the error. Note that a CHECK constraint is violated if it evaluates to FALSE; it is okay if it evaluates to TRUE or UNKNOWN.

**row has changed since last read -- operation cancelled**

**SQL Code** -208  
**SQL State** 22W02  
**ODBC state** (handled by ODBC driver)  
**Probable cause**

You have done a UPDATE (positioned) or DELETE (positioned) on a cursor declared as a SCROLL cursor, and the row you are changing has been updated since you read it. This prevents the 'lost update' problem.

**wrong number of values for INSERT**

**SQL Code** -207  
**SQL State** 53002  
**ODBC state** 37000  
**Probable cause**

The number of values you are trying to insert does not match the number of columns specified in the INSERT command, or the number of columns in the table if no columns are specified.

## Variable errors

### variable '%s' not found

**SQL Code** -260  
**SQL State** 42W14  
**ODBC state** 37000  
**Probable cause**

You have tried to DROP or SET the value of a SQL variable that was not created or was previously dropped.

### there is already a variable named '%s'

**SQL Code** -261  
**SQL State** 42W15  
**ODBC state** 37000  
**Probable cause**

You have tried to CREATE a variable with the name of another variable that already exists.

## Procedure errors

### label '%s' not found

**SQL Code** -262  
**SQL State** 42W24  
**ODBC state** 37000  
**Parameter** Name of the label that could not be found.  
**Probable cause**

The label referenced in a LEAVE statement was not found.

### invalid absolute or relative offset in FETCH

**SQL Code** -263  
**SQL State** 42W25  
**ODBC state** 37000  
**Probable cause**

The offset specified in a FETCH was invalid or NULL.

### wrong number of variables in FETCH

**SQL Code** -264

**SQL State** 42W26

**ODBC state** 37000

**Probable cause**

The number of variables specified in the FETCH statement does not match the number of select list items.

**procedure '%s' not found**

**SQL Code** -265

**SQL State** 52W09

**ODBC state** S0002

**Parameter** Name of the procedure that could not be found.

**Probable cause**

You have misspelled the name of a procedure, or you have connected with a different userid and forgotten to qualify a procedure name with a user name.

**database was initialized with an older version of the software**

**SQL Code** -266

**SQL State** 42W27

**ODBC state** 37000

**Probable cause**

The database is missing some system table definitions required for this statement. These system table definitions are normally created when a database is initialized. The database should be unloaded and reloaded into a database that has been initialized with a newer version of Watcom SQL.

**COMMIT/ROLLBACK not allowed within atomic operation**

**SQL Code** -267

**SQL State** 42W28

**ODBC state** 37000

**Probable cause**

A COMMIT or ROLLBACK statement was encountered while executing within an atomic operation.

**trigger '%s' not found**

**SQL Code** -268

**SQL State** 52W10

**ODBC state** S0002

**Parameter** Name of the trigger that could not be found.  
**Probable cause** You have misspelled the name of a trigger, or you have connected with a different userid and forgotten to qualify a trigger name with a user name.

**cannot delete a column referenced in a trigger definition**

**SQL Code** -269  
**SQL State** 53W06  
**ODBC state** S1000  
**Probable cause**

This error is reported if you attempt to delete a column that is referenced in a trigger definition. DROP the trigger before performing the ALTER command.

**cannot drop a user that owns procedures in runtime engine**

**SQL Code** -270  
**SQL State** 55W04  
**ODBC state** 37000  
**Probable cause**

This error is reported by the runtime engine if you attempt to drop a user that owns procedures. Because this operation would result in dropping procedures, and the runtime engine cannot drop procedures, it is not allowed. Use the development engine.

**trigger definition conflicts with existing triggers**

**SQL Code** -271  
**SQL State** 52W11  
**ODBC state** S0001  
**Probable cause**

A trigger definition could not be created because it conflicts with an existing trigger definition. A trigger with the same name may already exist.

**invalid REFERENCES clause in trigger definition**

**SQL Code** -272  
**SQL State** 52W12  
**ODBC state** 37000



**Probable cause**

The REFERENCES clause in a trigger definition is invalid. An OLD correlation name may have been specified in a BEFORE INSERT trigger, or a NEW correlation name may have been specified in an AFTER DELETE trigger. In both cases, the values do not exist and cannot be referenced.

**COMMIT/ROLLBACK not allowed within trigger actions**

**SQL Code** -273  
**SQL State** 2D501  
**ODBC state** 37000  
**Probable cause**

An attempt was made to execute a statement that is not allowed while performing a trigger action. COMMIT and ROLLBACK statements cannot be executed from a trigger.

**procedure or trigger calls have nested too deeply**

**SQL Code** -274  
**SQL State** 42W29  
**ODBC state** 37000  
**Probable cause**

You have probably defined a procedure or trigger that causes unlimited recursion.

## Option errors

**invalid option '%s' -- no PUBLIC setting exists**

**SQL Code** -200  
**SQL State** 42W16  
**ODBC state** 37000  
**Parameter** Name of the invalid option.  
**Probable cause**

You have probably misspelled the name of an option in the SET OPTION command. You can only define an option for a user if the database administrator has supplied a PUBLIC value for that option.

### **invalid setting for option '%s'**

**SQL Code** -201  
**SQL State** 42W17  
**ODBC state** 37000  
**Parameter** Name of the invalid option.  
**Probable cause**

You have supplied an invalid value for an option in the SET command. Some options only allow numeric values, while other options only allow the values 'on' and 'off'.

### **Concurrency errors**

#### **%s has the row in %s locked**

**SQL Code** -210  
**SQL State** 42W18  
**ODBC state** 40001  
**Parameter** Name of another user.  
**Parameter** Table which generates the error.  
**Probable cause**

You have attempted to read or write a row and it is locked by another user. Note that this error will only be received if the database option BLOCKING is set to OFF. Otherwise, the requesting transaction will block until the row lock is released.

#### **not allowed while %s is using the database**

**SQL Code** -211  
**SQL State** 42W19  
**ODBC state** 40001  
**Probable cause**

You have attempted to CREATE or DROP a dbspace and there are other active users of the database. You must be the only connection for these commands.

#### **CHECKPOINT command requires a rollback log**

**SQL Code** -212  
**SQL State** 42W20

**ODBC state** 40001

**Probable cause**

You cannot use a CHECKPOINT command when the database engine is running in bulk mode without a rollback log.

**table in use**

**SQL Code** -214

**SQL State** 42W21

**ODBC state** 40001

**Probable cause**

You have attempted to ALTER or DROP a table that is being used by other active users of the database.

**procedure in use**

**SQL Code** -215

**SQL State** 42W23

**ODBC state** 40001

**Probable cause**

You have attempted to DROP a procedure that is being used by other active users of the database.

## Savepoint errors

**savepoint '%s' not found**

**SQL Code** -220

**SQL State** 3B001

**ODBC state** S1000

**Parameter** Name of savepoint.

**Probable cause**

You attempted to rollback to a savepoint that does not exist.

**ROLLBACK TO SAVEPOINT not allowed**

**SQL Code** -221

**SQL State** 3B002

**ODBC state** S1000

**Probable cause**

A ROLLBACK TO SAVEPOINT within an atomic operation is not allowed to a savepoint established before the atomic operation.

**result set not allowed from within an atomic compound statement**

**SQL Code** -222  
**SQL State** 3BW02  
**ODBC state** S1000

**Probable cause**

A SELECT statement with no INTO clause or a RESULT CURSOR statement are not allowed within an atomic compound statement.

**savepoints require a rollback log**

**SQL Code** -213  
**SQL State** 3BW01  
**ODBC state** S1000

**Probable cause**

You cannot use savepoints when the database engine is running in bulk mode without a rollback log.

## **Version checking errors**

**sqlpp/dblib version mismatch**

**SQL Code** -230  
**SQL State** 08W18  
**ODBC state** 08001

**Probable cause**

Your executable has source files with Embedded SQL that were preprocessed with a preprocessor that does not match the database interface library.

**dblib/database engine version mismatch**

**SQL Code** -231  
**SQL State** 08W19  
**ODBC state** 08001

**Probable cause**

Your executable uses a database interface library that does not match the version number of the database engine.

**server/database engine version mismatch**

**SQL Code** -232  
**SQL State** 08W20  
**ODBC state** 08001

**Probable cause**

Your version of the database server software is not compatible with your version of the database engine.

## Backup errors

**unknown backup operation**

**SQL Code** -240  
**SQL State** WB001  
**ODBC state** S1000

**Probable cause**

An invalid backup command operation was specified in a call to *db\_backup*.

**database backup not started**

**SQL Code** -241  
**SQL State** WB002  
**ODBC state** S1000

**Probable cause**

A database backup could not be started. Either you do not have DBA authority, or another backup has started and not completed.

**incomplete transactions prevent transaction log renaming**

**SQL Code** -242  
**SQL State** WB003  
**ODBC state** S1000

**Probable cause**

The last page in the transaction log was read by a call to *db\_backup*. One or more currently active connections have

partially completed transactions, preventing the transaction log file from being renamed. The *db\_backup* call should be reissued.

#### unable to delete database file

**SQL Code** -243  
**SQL State** WB004  
**ODBC state** S1000  
**Probable cause**

The specified file could not be deleted. The filename should not be the same as any database file that is currently in use.

## Miscellaneous errors

#### user-defined exception signalled

**SQL Code** -297  
**SQL State** 99999  
**ODBC state** S1000  
**Probable cause**

A stored procedure or trigger signalled a user-defined exception. This error state is reserved for use within stored procedures or triggers which contain exception handlers, as a way of signalling an exception which can be guaranteed to not have been caused by the database engine.

#### attempted two active database requests

**SQL Code** -298  
**SQL State** 42W22  
**ODBC state** S1000  
**Probable cause**

In Embedded SQL, you have attempted to submit a database request while you have another request in process. This often occurs in Windows when processing the WM\_PAINT message causes a database request, and you get a second WM\_PAINT before the database request has completed.

## User interruption

### statement interrupted by user

**SQL Code** -299  
**SQL State** 57014  
**ODBC state** S1000  
**Probable cause**

The user has aborted a statement during its execution. The database was able to stop the operation without doing a rollback. If the statement is INSERT, UPDATE, or DELETE, any changes made by the statement will be canceled.

If the statement is a data definition command (for example CREATE TABLE), the command will be canceled, but the COMMIT that was done as a side effect will not be canceled.

## Errors that cause a rollback

### run time SQL error -- %s

**SQL Code** -300  
**SQL State** 40000  
**ODBC state** S1000  
**Parameter** Identification of the error.  
**Probable cause**

This error indicates an internal database error, and should be reported to Watcom.

### internal database error %s transaction rolled back

**SQL Code** -301  
**SQL State** 40W01  
**ODBC state** S1000  
**Parameter** Identification of the error.  
**Probable cause**

This error indicates an internal database error, and should be reported to Watcom. A ROLLBACK WORK command has been automatically executed.

**terminated by user transaction rolled back**

**SQL Code** -302  
**SQL State** 40W02  
**ODBC state** S1000  
**Probable cause**

The user has aborted a command while the database was executing. A ROLLBACK WORK command has been automatically executed. This will happen when the engine is running in bulk mode and the user aborts an INSERT, UPDATE, or DELETE operation.

**disk full transaction rolled back**

**SQL Code** -304  
**SQL State** 40W03  
**ODBC state** S1000  
**Probable cause**

Your hard disk is out of free space. A ROLLBACK WORK command has been automatically executed.

**I/O error %s transaction rolled back**

**SQL Code** -305  
**SQL State** 40W04  
**ODBC state** S1000  
**Probable cause**

Watcom SQL has detected a problem with your hard disk. If you cannot find a hardware error using the operating system disk check utility (eg. in DOS, CHKDSK, and in QNX, CHKFSYS), report the problem to Watcom. A ROLLBACK WORK command has been automatically executed.

**deadlock detected**

**SQL Code** -306  
**SQL State** 40001  
**Probable cause**

You have attempted to read or write a row and it is locked by another user. Also, the other user is blocked directly or indirectly on your own transaction. This is a deadlock situation and your transaction has been chosen as the one to rollback.



**all threads are blocked**

**SQL Code** -307  
**SQL State** 40W06  
**ODBC state** 40001  
**Probable cause**

You have attempted to read or write a row and it is locked by another user. Also, all other threads (see database option `THREAD_COUNT`) are blocked waiting for a lock to be released. This is a deadlock situation and your transaction has been chosen as the one to rollback.

## Errors specific to WSQL HLI

**invalid WSQL HLI command syntax**

**SQL Code** -400  
**SQL State** 42W11  
**Probable cause**

The command string that you sent to *wsqlexec* cannot be understood. Make sure that all of the keywords in the command string are spelled properly, and that variable names (such as host variable, cursor or statement names) are not too long.

**invalid WSQL HLI cursor name**

**SQL Code** -401  
**SQL State** 34W01  
**Probable cause**

The cursor name indicated in your command is not a valid one. For instance, this error would occur if you tried to close a cursor that had never even been declared.

**invalid WSQL HLI statement name**

**SQL Code** -402  
**SQL State** 26W01  
**Probable cause**

The statement name indicated in your command is not a valid one. This typically indicates that you have failed to prepare the statement.

**invalid WSQL HLI host variable name**

**SQL Code** -403

**SQL State** 42W09

**Probable cause**

You have used a host variable, and the host variable callback function does not recognize it.

**invalid WSQL HLI host variable value**

**SQL Code** -404

**SQL State** 42W10

**Probable cause**

You have used a host variable, and the host variable value is too long.

**invalid WSQL HLI callback function**

**SQL Code** -405

**SQL State** 42W12

**Probable cause**

WSQL HLI needed to use a callback function, but the function has not been registered using the *wsqlregisterfuncs* entry point.

**WSQL HLI internal error**

**SQL Code** -406

**SQL State** WI007

**Probable cause**

This is a Watcom SQL internal error and should be reported to Watcom.

**an argument passed to a WSQL HLI function was invalid**

**SQL Code** -407

**SQL State** 22W01

**Probable cause**

One of the arguments passed to a WSQL HLI function was invalid. This may indicate that a pointer to a command string or result buffer is the null pointer.

## **Internal errors (assertion failed)**

The Watcom SQL engine has many internal checks that have been designed to detect possible database corruption as soon as possible. If the database engine prints an Assertion Failed message, you should not continue to use it before attempting to determine the cause. Record the assertion number displayed on the screen and report the error to WATCOM. The DBVALID utility is useful for determining if your database file is corrupt. You may find it necessary to reconstruct your data from backups and transaction logs (see "Backup and Recovery" on page 115).



## APPENDIX D

# Watcom SQL Keywords

About this appendix    This Appendix lists the SQL keywords.

If any of these keywords are to be used as an identifier they should be enclosed in quotation marks. A number of these keywords are not used in the current implementation of Watcom SQL but are reserved for future enhancements.

---

absolute	dynamic	long	revoke
add	each	loop	right
after	else	match	rollback
all	elseif	membership	row
alter	end	mode	rows
and	endif	modify	savepoint
any	escape	named	schedule
as	exception	natural	scroll
asc	exclusive	new	select
atomic	execute	next	set
autoincrement	exists	no	share
before	false	not	signal
begin	fetch	null	smallint
between	first	numeric	some
binary	float	of	sqlcode
by	for	old	sqlstate
cache	foreign	on	statement
call	from	only	statistics
cascade	full	open	subtrans
case	global	option	subtransaction
cast	grant	options	table
char	group	or	temporary
character	having	order	then
check	hold	others	time
checkpoint	identified	out	timestamp
close	if	outer	to
column	in	precision	trigger
comment	index	prepare	true
commit	inner	preserve	union
connect	inout	primary	unique
create	insert	prior	unknown
cross	instead	privileges	update
current	int	procedure	user
cursor	integer	read	using
date	into	real	validate
dba	is	reference	value
dbspace	isolation	references	values
dec	join	referencing	varchar
decimal	key	relative	variable
declare	last	release	varying
default	leave	rename	view
delete	left	resignal	when
desc	level	resource	where
distinct	like	restrict	while
do	local	result	with
double	lock	resume	work
drop			

## APPENDIX E

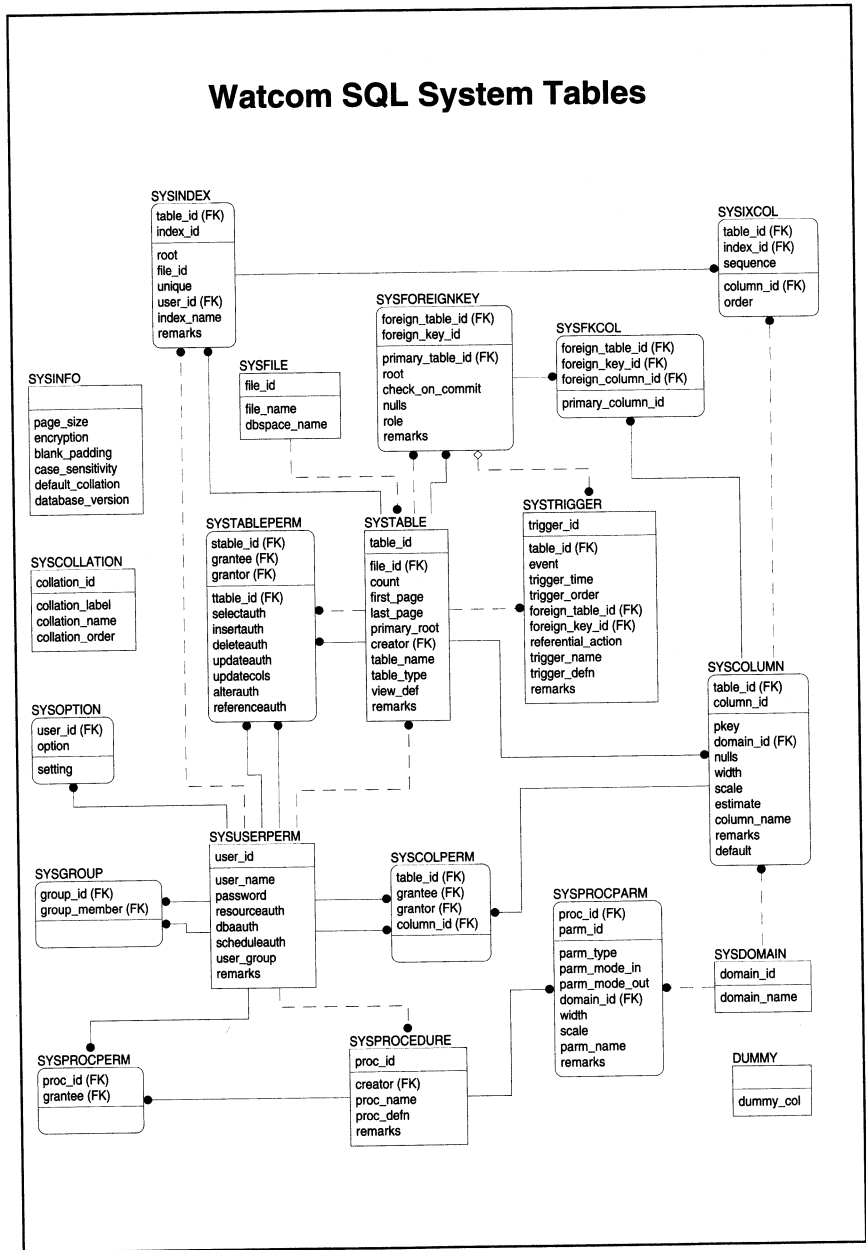
# Watcom SQL System Tables

**About this appendix** The structure of every Watcom SQL database is described in a number of **system tables**. The Entity-Relationship diagram on the next page shows all the system tables and the foreign keys connecting the system tables. These tables have been created by the **SYS** user ID. The contents of these tables can only be changed by the database system. Thus, the **UPDATE**, **DELETE**, and **INSERT** commands cannot be used to modify the contents of these tables. Further, the structure of these tables cannot be changed using the **ALTER TABLE** and **DROP** commands.

**Contents** These system tables will be described via the **CREATE TABLE** commands used to create them. They serve as good examples of how tables are created in SQL. Following the **CREATE TABLE** command, each column will be briefly described.

Several of the columns have only two possible values. Usually these values are "Y" and "N" for "yes" and "no" respectively. These columns are designated by "(Y/N)".

# Watcom SQL System Tables





## SYS.SYSUSERPERM

```

CREATE TABLE SYS.SYSUSERPERM (
    user_id          SMALLINT NOT NULL,
    user_name       CHAR(128) NOT NULL UNIQUE,
    password        CHAR(128),
    resourceauth    CHAR(1) NOT NULL,
    dbaauth         CHAR(1) NOT NULL,
    scheduleauth    CHAR(1) NOT NULL,
    user_group      CHAR(1) NOT NULL,
    remarks         LONG VARCHAR,
    PRIMARY KEY    ( user_id )
)

```

Each row of SYSUSERPERM describes one user ID.

### **user\_id**

Each new user ID is assigned a unique number (the **user number**) that is the primary key for SYSUSERPERM.

### **user\_name**

A string containing the name for the user ID. Each userid must have a unique name.

### **password**

The password for the user ID. The password contains the NULL value for the special userids SYS and PUBLIC, preventing anyone from connecting to these user IDs.

### **resourceauth (Y/N)**

Indicate whether the user has RESOURCE authority. Resource authority is required to create tables.

### **dbaauth (Y/N)**

Indicate whether the user has DBA (database administrator) authority. DBA authority is very powerful, and should be restricted to as few user IDs as possible for security purposes.

### **scheduleauth (Y/N)**

Indicate whether the user has SCHEDULE authority. This is currently not used by Watcom SQL.

### **user\_group (Y/N)**

Indicate whether the user is a group.

**remarks**

A comment string.

When a database is initialized, three user IDs are created:

- ◆ **SYS** The creator of all the system tables.
- ◆ **PUBLIC** A special user ID used to record PUBLIC permissions.
- ◆ **DBA** The database administrator user ID is the only usable user ID in an initialized system. The initial password is SQL.

There is no way to connect to the SYS or PUBLIC user IDs.

## SYS.SYSGROUP

```
CREATE TABLE SYS.SYSGROUP (  
    group_id          SMALLINT NOT NULL,  
    group_member     SMALLINT NOT NULL,  
    PRIMARY KEY      ( group_id, group_member ),  
    FOREIGN KEY      group_id ( group_id ) REFERENCES  
                    SYS.SYSUSERPERM ( user_id ),  
    FOREIGN KEY      group_member ( group_member )  
                    REFERENCES SYS.SYSUSERPERM ( user_id )  
)
```

There is one row in SYSGROUP for every member of every group. This table describes a many-to-many relationship between groups and members. A group may have many members and a user may be a member of many groups.

### **group\_id**

The user number of group.

### **group\_member**

The user number of a member.

## **SYS.SYSFILE**

```
CREATE TABLE SYS.SYSFILE (  
    file_id          SMALLINT NOT NULL,  
    file_name        CHAR(80) NOT NULL,  
    dbspace_name     CHAR(128) NOT NULL,  
    PRIMARY KEY      ( file_id )  
)
```

Every database consists of one or more operating system files. Each file is recorded in SYSFILE.

### **file\_id**

Each file in a database is assigned a unique number. This file identifier is the primary key for SYSFILE. All system tables are stored in file\_id 0.

### **file\_name**

The database name is stored when a database is created. This name is for informational purposes only.

### **dbspace\_name**

Every file has a **dbspace name** that is unique. It is used in the CREATE TABLE command.

# SYS.SYSTABLE

```

CREATE TABLE SYS.SYSTABLE (
    table_id          SMALLINT NOT NULL,
    file_id           SMALLINT NOT NULL,
    count             INTEGER NOT NULL,
    first_page        INT NOT NULL,
    last_page         INT NOT NULL,
    primary_root      INT NOT NULL,
    creator           SMALLINT NOT NULL,
    table_name        CHAR(128) NOT NULL,
    table_type        CHAR(10) NOT NULL,
    view_def          LONG VARCHAR,
    remarks           LONG VARCHAR,
    PRIMARY KEY      ( table_id ),
    UNIQUE           ( table_name, creator ),
    FOREIGN KEY      ( creator ) REFERENCES
        SYS.SYSUSERPERM ( user_id ),
    FOREIGN KEY      REFERENCES SYS.SYSFILE
)

```

Each row of SYSTABLE describes one table or view in the database.

## **table\_id**

Each table or view is assigned a unique number (the **table number**) that is the primary key for SYSTABLE.

## **file\_id**

The file number indicates which database file contains the table. The `file_id` is a FOREIGN KEY for SYSFILE.

## **count**

The number of rows in the table is updated during each successful CHECKPOINT. This number is used by Watcom SQL when optimizing database access. The count is always 0 for a view.

## **first\_page**

Each Watcom SQL database is divided into a number of fixed size pages. This value identifies the first page containing information for this table, and is used internally to find the start of this table. The `first_page` is always 0 for a view.

## **last\_page**

The last page containing information for this table. The `last_page` is always 0 for a view.

**primary\_root**

Primary keys are stored in the database as B-trees. The primary\_root locates the root of the B-tree for the primary key for the table. It will be 0 for a view and for a table with no primary key.

**creator**

This user number identifies the owner of the table or view. The name of the user can be found by looking in SYSUSERPERM.

**table\_name**

The name of the table or view. One creator cannot have two tables or views with the same name.

**table\_type**

This column will be "BASE" for base tables and "VIEW" for views. It will be "GBL TEMP" for global temporary tables and "LCL TEMP" for local temporary tables.

**view\_def**

For a view, this column contains the CREATE VIEW command used to create the view. For a table, this column will contain any CHECK constraints for the table.

**remarks**

A comment string.

## SYS.SYSDOMAIN

```
CREATE TABLE SYS.SYSDOMAIN (  
    domain_id          SMALLINT NOT NULL,  
    domain_name       CHAR(128) NOT NULL,  
    PRIMARY KEY       ( domain_id )  
)
```

Each of the predefined data types (sometimes called a **domain**) in Watcom SQL is assigned a unique number. The SYSDOMAIN table is provided for informational purposes to show the association between these numbers and the appropriate data type. This table is never changed by Watcom SQL.

### **domain\_id**

The unique number assigned to each data type. These numbers cannot be changed.

### **domain\_name**

A string containing the data type normally found in the CREATE TABLE command, such as char or integer.

## SYS.SYSCOLUMN

```
CREATE TABLE SYS.SYSCOLUMN (  
    table_id          SMALLINT NOT NULL,  
    column_id        SMALLINT NOT NULL,  
    pkey             CHAR(1) NOT NULL,  
    domain_id        SMALLINT NOT NULL,  
    nulls            CHAR(1) NOT NULL,  
    width            SMALLINT NOT NULL,  
    scale            SMALLINT NOT NULL,  
    estimate         INT NOT NULL,  
    column_name      CHAR(128) NOT NULL,  
    remarks          LONG VARCHAR,  
    "default"       LONG VARCHAR,  
    PRIMARY KEY     ( table_id, column_id ),  
    FOREIGN KEY     REFERENCES SYS.SYSTABLE,  
    FOREIGN KEY     REFERENCES SYS.SYSDOMAIN  
)
```

Each column in every table or view is described by one row in SYSCOLUMN.

### **table\_id**

The table number uniquely identifies the table or view to which this column belongs.

### **column\_id**

Each table starts numbering columns at 1. The order of column numbers determines the order that columns are displayed in the command select \* from table.

### **pkey (Y/N)**

Indicate whether this column is part of the primary key for the table.

### **domain\_id**

Identify the data type for the column by the data type number listed in the SYSDOMAIN table.

### **nulls (Y/N)**

Indicate whether the NULL value is allowed in this column.

### **width**

This column contains the length of string columns, the precision of numeric columns, and the number of bytes of storage for all other data types.



**scale**

The number of digits after the decimal point for numeric data type columns, and zero for all other data types.

**estimate**

A self-tuning parameter for the optimizer. Watcom SQL will learn from previous queries by adjusting guesses that are made by the optimizer.

**column\_name**

The name of the column.

**remarks**

A comment string.

**default**

The default value for the column. This value is only used when an INSERT statement does not specify a value for this column.

## SYS.SYSINDEX

```
CREATE TABLE SYS.SYSINDEX (
  table_id          SMALLINT NOT NULL,
  index_id         SMALLINT NOT NULL,
  root             INT NOT NULL,
  file_id          SMALLINT NOT NULL,
  "unique"        CHAR(1) NOT NULL,
  creator          SMALLINT NOT NULL,
  index_name       CHAR(128) NOT NULL,
  remarks         LONG VARCHAR,
  PRIMARY KEY     ( table_id, index_id ),
  UNIQUE          ( index_name, creator ),
  FOREIGN KEY     REFERENCES SYS.SYSTABLE,
  FOREIGN KEY     REFERENCES SYS.SYSFILE,
  FOREIGN KEY     ( creator ) REFERENCES
  SYS.SYSUSERPERM ( user_id )
)
```

Each index in the database is described by one row in SYSINDEX. Each column in the index is described by one row in SYSIXCOL.

### **table\_id**

The table number uniquely identifies the table to which this index applies.

### **index\_id**

Each index for one particular table is assigned a unique index number.

### **root**

Indexes are stored in the database as B-trees. The **root** identifies the location of the root of the B-tree in the database file.

### **file\_id**

The index is completely contained in the file with this `file_id` (see **SYSFILE**). In the current implementation of Watcom SQL, this file is always the same as the file containing the table.

### **unique**

Indicate whether the index is a unique index ("Y"), a non-unique index ("N"), or a unique constraint ("U"). A unique index prevents two rows in the indexed table from having the same values in the index columns.

### **creator**

The user number of the creator of the index. In the current implementation of Watcom SQL, this user is always the same as the creator of the table identified by `table_id`.

**index\_name**

The name of the index. A user ID cannot have two indexes with the same name.

**remarks**

A comment string.

## SYS.SYSIXCOL

```
CREATE TABLE SYS.SYSIXCOL (  
  table_id          SMALLINT NOT NULL,  
  index_id         SMALLINT NOT NULL,  
  sequence         SMALLINT NOT NULL,  
  column_id       SMALLINT NOT NULL,  
  "order"         CHAR(1) NOT NULL,  
  PRIMARY KEY     ( table_id, index_id, sequence )  
  FOREIGN KEY     REFERENCES SYS.SYSINDEX,  
  FOREIGN KEY     REFERENCES SYS.SYSCOLUMN  
)
```

Every index has one row in SYSIXCOL for each column in the index.

### **table\_id**

Identifies the table to which the index applies.

### **index\_id**

Identifies in which index this column is used. Together, table\_id and index\_id identify one index described in SYSINDEX.

### **sequence**

Each column in an index is assigned a unique number starting at 0. The order of these numbers determines the relative significance of the columns in the index. The most important column has sequence number 0.

### **column\_id**

The column number identifies which column is indexed. Together, table\_id and column\_id identify one column in SYSCOLUMN.

### **order (A/D)**

Indicate whether this column in the index is kept in ascending or descending order.

# SYS.SYSFOREIGNKEY

```

CREATE TABLE SYS.SYSFOREIGNKEY (
    foreign_table_id  SMALLINT NOT NULL,
    foreign_key_id   SMALLINT NOT NULL,
    primary_table_id SMALLINT NOT NULL,
    root             INT NOT NULL,
    check_on_commit  CHAR(1) NOT NULL,
    nulls            CHAR(1) NOT NULL,
    role             CHAR(128) NOT NULL,
    remarks          LONG VARCHAR,
    PRIMARY KEY     ( foreign_table_id, foreign_key_id ),
    UNIQUE          ( role, foreign_table_id ),
    FOREIGN KEY     foreign_table ( foreign_table_id )
    REFERENCES SYS.SYSTABLE ( table_id ),
    FOREIGN KEY     primary_table ( primary_table_id )
    REFERENCES SYS.SYSTABLE ( table_id )
)

```

A foreign key is a **relationship** between two tables—the **foreign table** and the **primary table**. Every foreign key is defined by one row in SYSFOREIGNKEY and one or more rows in SYSFKCOL. SYSFOREIGNKEY contains general information about the foreign key while SYSFKCOL identifies the columns in the foreign key and associates each column in the foreign key with a column in the primary key of the primary table.

## foreign\_table\_id

The table number of the foreign table.

## foreign\_key\_id

Each foreign key has a **foreign key number** that is unique with respect to:

- ◆ The key number of all other foreign keys for the foreign table
- ◆ The key number of all foreign keys for the primary table
- ◆ The index number of all indexes for the foreign table

## primary\_table\_id

The table number of the primary table.

## root

Foreign keys are stored in the database as B-trees. The root identifies the location of the root of the B-tree in the database file.

## nulls (Y/N)

Indicate whether the columns in the foreign key are allowed to contain the

NULL value. Note that this setting is independent of the nulls setting in the columns contained in the foreign key.

**check\_on\_commit (Y/N)**

Indicate whether INSERT and UPDATE commands should wait until the next COMMIT command to check if foreign keys are valid. A foreign key is valid if, for each row in the foreign table, the values in the columns of the foreign key either contain the NULL value or match the primary key values in some row of the primary table.

**role**

The name of the relationship between the foreign table and the primary table. Unless otherwise specified, the role name will be the same as the name of the primary table. The foreign table cannot have two foreign keys with the same role name.

**remarks**

A comment string.

## SYS.SYSFKCOL

```

CREATE TABLE SYS.SYSFKCOL (
  foreign_table_id  SMALLINT NOT NULL,
  foreign_key_id    SMALLINT NOT NULL,
  foreign_column_id SMALLINT NOT NULL,
  primary_column_id SMALLINT NOT NULL,
  PRIMARY KEY      ( foreign_table_id,
                    foreign_key_id, foreign_column_id ),
  FOREIGN KEY      REFERENCES SYS.SYSFOREIGNKEY,
  FOREIGN KEY      ( foreign_table_id,
                    foreign_column_id ) REFERENCES
  SYS.SYSCOLUMN ( table_id, column_id )
)

```

Each row of SYSFKCOL describes the association between a foreign column in the foreign table of a relationship and the primary column in the primary table.

### **foreign\_table\_id**

The table number of the foreign table.

### **foreign\_key\_id**

The key number of the FOREIGN KEY for the foreign table. Together, `foreign_table_id` and `foreign_key_id` uniquely identify one row in SYSFOREIGNKEY, and the table number for the primary table can be obtained from that row.

### **foreign\_column\_id**

This column number, together with the `foreign_table_id`, identify the foreign column description in SYSCOLUMN.

### **primary\_column\_id**

This column number, together with the `primary_table_id` obtained from SYSFOREIGNKEY, identify the primary column description in SYSCOLUMN.

# SYS.SYSTABLEPERM

```

CREATE TABLE SYS.SYSTABLEPERM (
  stable_id          SMALLINT NOT NULL,
  grantee           SMALLINT NOT NULL,
  grantor           SMALLINT NOT NULL,
  ttable_id        SMALLINT NOT NULL,
  selectauth       CHAR(1) NOT NULL,
  insertauth       CHAR(1) NOT NULL,
  deleteauth       CHAR(1) NOT NULL,
  updateauth       CHAR(1) NOT NULL,
  updatecols       CHAR(1) NOT NULL,
  alterauth        CHAR(1) NOT NULL,
  referenceauth    CHAR(1) NOT NULL,
  PRIMARY KEY      ( stable_id, grantee, grantor ),
  FOREIGN KEY      ( stable_id )
                  REFERENCES SYS.SYSTABLE ( table_id ),
  FOREIGN KEY      ( future ( ttable_id )
                  REFERENCES SYS.SYSTABLE ( table_id ),
  FOREIGN KEY      ( grantee ( grantee ) REFERENCES
                  SYS.SYSUSERPERM ( user_id ),
  FOREIGN KEY      ( grantor ( grantor )
                  REFERENCES SYS.SYSUSERPERM ( user_id )
)

```

Permissions given by the GRANT command are stored in SYSTABLEPERM. Each row in this table corresponds to one table, one user ID granting the permission (**grantor**) and one user ID granted the permission (**grantee**).

There are several types of permission that can be granted. Each permission can have one of the following three values.

## N

No, the grantee has not been granted this permission by the grantor.

## Y

Yes, the grantee has been given this permission by the grantor.

## G

The grantee has been given this permission. In addition, the grantee can grant the same permission to another user.



**Permissions**

The grantee might have been given permission for the same table by another grantor. If so, this information would be recorded in a different row of SYSTABLEPERM.

**stable\_id**

The table number of the table or view to which the permissions apply.

**grantor**

The user number of the user ID granting the permission.

**grantee**

The user number of the user ID receiving the permission.

**ttable\_id**

In the current version of Watcom SQL, this table number is always the same as stable\_id.

**selectauth (Y/N/G)**

Indicate whether SELECT permission has been granted.

**insertauth (Y/N/G)**

Indicate whether INSERT permission has been granted.

**deleteauth (Y/N/G)**

Indicate whether DELETE permission has been granted.

**updateauth (Y/N/G)**

Indicate whether UPDATE permission has been granted for all columns in the table. (Only UPDATE permission can be given on individual columns. All other permissions are for all columns in a table.)

**updatecols (Y/N)**

Indicates whether UPDATE permission has only been granted for some of the columns in the table. If updatecols has the value Y, there will be one or more rows in SYSCOLPERM granting update permission for the columns in this table.

**alterauth (Y/N/G)**

Indicate whether ALTER permission has been granted.

**referenceauth (Y/N/G)**

Indicate whether REFERENCE permission has been granted.

## SYS.SYSCOLPERM

```

CREATE TABLE SYS.SYSCOLPERM (
  table_id          SMALLINT NOT NULL,
  grantee           SMALLINT NOT NULL,
  grantor           SMALLINT NOT NULL,
  column_id         SMALLINT NOT NULL,
  PRIMARY KEY      ( table_id, grantee,
                    grantor, column_id ),
  FOREIGN KEY      ( grantee ) REFERENCES
  SYS.SYSUSERPERM ( user_id ),
  FOREIGN KEY      ( grantor ) REFERENCES
  SYS.SYSUSERPERM ( user_id ),
  FOREIGN KEY      ( column_id ) REFERENCES
  SYS.SYSCOLUMN
)

```

The GRANT command can give UPDATE permission to individual columns in a table. Each column with UPDATE permission is recorded in one row of SYSCOLPERM.

### **table\_id**

The table number for the table containing the column.

### **grantee**

The user number of the user ID given UPDATE permission on the column. If the grantee is the user number for the special PUBLIC user ID, the UPDATE permission is given to all user IDs.

### **grantor**

The user number of the user ID granting the permission.

### **column\_id**

This column number, together with the table\_id, identifies the column for which UPDATE permission has been granted.

## **SYS.SYSOPTION**

```
CREATE TABLE SYS.SYSOPTION (  
    user_id          SMALLINT NOT NULL,  
    "option"        CHAR(128) NOT NULL,  
    "setting"       CHAR(80) NOT NULL,  
    PRIMARY KEY     ( user_id, "option" ),  
    FOREIGN KEY     REFERENCES SYS.SYSUSERPERM  
)
```

Options settings are stored in the SYSOPTION table by the SET command. Each user can have their own setting for each option. In addition, settings for the PUBLIC user ID define the default settings to be used for user IDs that do not have their own setting.

### **user\_id**

The user number to whom this option setting applies.

### **option**

The name of the option.

### **setting**

The current setting for the named option.

# SYS.SYSINFO

```
CREATE TABLE SYS.SYSINFO (
  page_size          SMALLINT NOT NULL,
  encryption        CHAR(1) NOT NULL,
  blank_padding     CHAR(1) NOT NULL,
  case_sensitivity  CHAR(1) NOT NULL,
  default_collation CHAR(10) NOT NULL,
  database_version  SMALLINT NOT NULL
)
```

This table indicates the database characteristics as defined when the database was created using DBINIT. It always contains only one row.

## **page\_size**

The page size specified to DBINIT. The default value is 1024.

## **encryption**

The value "Y" or "N" depending on whether the -e switch was used with DBINIT.

## **blank\_padding**

The value "Y" or "N" depending on whether the -b switch was used with DBINIT.

## **case\_sensitivity**

The value "Y" or "N" depending on whether the -c switch was used with DBINIT. Note that case sensitivity affects both value comparisons and table and column name comparisons. For example, if case sensitivity is enabled, the system catalog names such as SYSCATALOG must be specified in upper case since that is how the name was spelled when it was created.

## **default\_collation**

A string corresponding to the collation\_label in SYSCOLLATE corresponding to the collation sequence specified with DBINIT. The default value corresponds to the Multilingual collation sequence (code page 850), which was the default in Watcom SQL Version 3.1. The collation sequence is used for all string comparisons, including searches for character strings as well as column and table name comparison.

## **database\_version**

A small integer value indicating the database format. As newer versions of Watcom SQL become available, new features may require that the format of the database file change. The version number allows Watcom SQL software

to determine if this database was created with a newer version of the software and thus cannot be understood by the software in use.

**Earlier versions**

This table does not exist in databases created with Watcom SQL Version 3.1 or earlier.

# SYS.SYSCOLLATE

```
CREATE TABLE SYS.SYSCOLLATION (
    collation_id      SMALLINT NOT NULL,
    collation_name    CHAR(128) NOT NULL,
    collation_label   CHAR(10) NOT NULL,
    collation_order   BINARY(256) NOT NULL,
    PRIMARY KEY      ( collation_id )
)
```

This table contains the collation sequences available to Watcom SQL. There is no way to modify the contents of this table.

## **collation\_id**

A unique number identifying the collation sequence. The collation sequence with `collation_id` equal 2 is the sequence used in previous versions of Watcom SQL, and is the default when a database is created with DBINIT.

## **collation\_name**

The name of the collation sequence.

## **collation\_label**

A string identifying each of the available collation sequences. The collation sequence to be used is selected when the database is created by specifying the collation label with the `-z` option.

## **collation\_order**

An array of bytes defining how each of the 256 character codes are treated for comparison purposes. All string comparisons translate each character according to the collation order table before comparing the characters. For the different ASCII code pages, the only difference is how accented characters are sorted. In general, an accented character is sorted as if it were the same as the nonaccented character.

### **Earlier versions**

This table does not exist in databases created with Watcom SQL Version 3.1 or earlier and only exists if the `-z` option is specified when the database is created.

## **SYS.DUMMY**

```
CREATE TABLE SYS.DUMMY (  
    dummy_col          INT NOT NULL  
)
```

The DUMMY table is provided as a table that always has exactly one row. This can be useful for extracting information from the database, as in the following example that gets the current user ID and the current date from the database.

```
SELECT USER, today(*) FROM SYS.DUMMY
```

### **dummy\_col**

This column is not used. It is present because a table cannot be created with no columns.



## SYS.SYSPROCEDURE

```

CREATE TABLE SYS.SYSPROCEDURE (
  proc_id          SMALLINT NOT NULL,
  creator          SMALLINT NOT NULL,
  proc_name        CHAR(128) NOT NULL,
  proc_defn        LONG VARCHAR,
  remarks          LONG VARCHAR,
  PRIMARY KEY     ( proc_id ),
  UNIQUE          ( proc_name, creator ),
  FOREIGN KEY     ( creator ) REFERENCES
                  SYS.SYSUSERPERM ( user_id )
)

```

Each procedure in the database is described by one row in SYSPROCEDURE.

### **proc\_id**

Each procedure is assigned a unique number (the **procedure number**) that is the primary key for SYSPROCEDURE.

### **creator**

This user number identifies the owner of the procedure. The name of the user can be found by looking in SYSUSERPERM.

### **proc\_name**

The name of the procedure. One creator cannot have two procedures with the same name.

### **proc\_defn**

The command used to create the procedure.

### **remarks**

A comment string.

# SYS.SYSTRIGGER

```

CREATE TABLE SYS.SYSTRIGGER (
  trigger_id          SMALLINT NOT NULL,
  table_id           SMALLINT NOT NULL,
  event              CHAR(1) NOT NULL,
  trigger_time       CHAR(1) NOT NULL,
  trigger_order      SMALLINT,
  foreign_table_id   SMALLINT,
  foreign_key_id     SMALLINT,
  referential_action CHAR(1),
  trigger_name       CHAR(128),
  trigger_defn       LONG VARCHAR NOT NULL,
  remarks            LONG VARCHAR,
  PRIMARY KEY        ( trigger_id ),
  UNIQUE             ( trigger_name ),
  UNIQUE             ( table_id, event,
                      trigger_time, trigger_order ),
  UNIQUE             ( table_id, foreign_table_id,
                      foreign_key_id, event ),
  FOREIGN KEY        REFERENCES SYS.SYSTABLE,
  FOREIGN KEY        REFERENCES SYS.SYSFOREIGNKEY
)

```

Each trigger in the database is described by one row in SYSTRIGGER. The table also contains triggers automatically created by the database for foreign key definitions which have a referential triggered action (such as ON DELETE CASCADE).

## trigger\_id

Each trigger is assigned a unique number (the **trigger number**) that is the primary key for SYSTRIGGER.

## table\_id

The table number uniquely identifies the table to which this trigger belongs.

## event

The event that will cause the trigger to fire. This single character value corresponds to the trigger event that was specified when the trigger was created.

**D** DELETE  
**I** INSERT  
**U** UPDATE  
**C** UPDATE OF column-list

## trigger\_time

The time at which the trigger will fire. This single character value

corresponds to the trigger time that was specified when the trigger was created.

- A** AFTER
- B** BEFORE

**trigger\_order**

The order in which the trigger will fire. This determines the order that triggers are fired when there are triggers of the same type (insert, update, or delete) that fire at the same time (before or after).

**foreign\_table\_id**

The foreign table number identifies the table containing a foreign key definition which has a referential triggered action (such as ON DELETE CASCADE).

**foreign\_key\_id**

The foreign key number identifies the foreign key for the table referenced by foreign\_table\_id.

**referential\_action**

The action defined by a foreign key. This single character value corresponds to the action that was specified when the foreign key was created.

- C** CASCADE
- D** DELETE
- N** SET NULL
- R** RESTRICT

**trigger\_name**

The name of the trigger. One table cannot have two triggers with the same name.

**trigger\_defn**

The command used to create the trigger.

**remarks**

A comment string.

# SYS.SYSPROCPARM

```
CREATE TABLE SYS.SYSPROCPARM (  
    proc_id          SMALLINT NOT NULL,  
    parm_id          SMALLINT NOT NULL,  
    parm_type        SMALLINT NOT NULL,  
    parm_mode_in     CHAR(1) NOT NULL,  
    parm_mode_out    CHAR(1) NOT NULL,  
    domain_id        SMALLINT NOT NULL,  
    width            SMALLINT NOT NULL,  
    scale            SMALLINT NOT NULL,  
    parm_name        CHAR(128) NOT NULL,  
    remarks          LONG VARCHAR,  
    PRIMARY KEY      ( proc_id, parm_id ),  
    FOREIGN KEY      REFERENCES SYS.SYSPROCEDURE,  
    FOREIGN KEY      REFERENCES SYS.SYSDOMAIN  
)
```

Each parameter to a procedure in the database is described by one row in SYSPROCEDURE.

## **proc\_id**

The procedure number uniquely identifies the procedure to which this parameter belongs.

## **parm\_id**

Each procedure starts numbering parameters at 1. The order of parameter numbers corresponds to the order in which they were defined.

## **parm\_type**

The type of parameter will be one of the following:

- 0 - variable** normal parameter
- 1 - result** result variable - used with procedure that return result sets
- 2 - SQLSTATE** SQLSTATE error value
- 3 - SQLCODE** SQLCODE error value

## **parm\_mode\_in (Y/N)**

Indicate whether this parameter supplies a value to the procedure (IN or INOUT parameters).

## **parm\_mode\_out (Y/N)**

Indicate whether this parameter returns a value from the procedure (OUT or INOUT parameters).

**domain\_id**

Identify the data type for the parameter by the data type number listed in the SYSDOMAIN table.

**width**

This column contains the length of string parameters, the precision of numeric parameters, and the number of bytes of storage for all other data types.

**scale**

The number of digits after the decimal point for numeric data type parameters, and zero for all other data types.

**parm\_name**

The name of the parameter.

**remarks**

A comment string.

## **SYS.SYSPROCPERM**

```
CREATE TABLE SYS.SYSPROCPERM (  
  proc_id          SMALLINT NOT NULL,  
  grantee         SMALLINT NOT NULL,  
  PRIMARY KEY     ( proc_id, grantee )  
  FOREIGN KEY     ( grantee ) REFERENCES  
                  SYS.SYSUSERPERM ( user_id ),  
  FOREIGN KEY     REFERENCES SYS.SYSPROCEDURE  
)
```

Only users who have been granted permission can call a procedure. Each row of the SYSPROCPERM table corresponds to one user granted permission to call one procedure.

### **proc\_id**

The procedure number uniquely identifies the procedure for which permission has been granted.

### **grantee**

The user number of the user ID receiving the permission.

## APPENDIX F

# Watcom SQL System Views

### About this appendix

This appendix lists predefined views for the Watcom SQL system tables. The system tables described in "Watcom SQL System Tables" on page 413 use numbers to identify tables, userids, and so forth. While this is efficient for internal use by Watcom SQL, it makes these tables difficult for people to interpret. A number of predefined views are provided that present the information in the system tables in a more readable format.

The definitions for the system views are included with their descriptions. Some of these definitions are complicated, but need not be understood to use the views. They serve as good examples of what can be accomplished using the SELECT command and views.

### Contents

The views are listed alphabetically.

## SYS.SYSCATALOG

```
CREATE VIEW SYS.SYSCATALOG ( creator, tname, dbspacename,  
                             tabletype, ncols, primary_key, "check", remarks )  
AS SELECT ( SELECT user_name FROM SYS.SYSUSERPERM  
            WHERE user_id = SYSTABLE.creator ),  
          table_name,  
          ( SELECT dbspace_name from SYS.SYSFILE  
            WHERE file_id = SYSTABLE.file_id ),  
          IF table_type='BASE' THEN 'TABLE'  
          ELSE table_type ENDIF,  
          ( SELECT count(*) FROM SYS.SYSCOLUMN  
            WHERE table_id = SYSTABLE.table_id ),  
          IF primary_root = 0 THEN 'N' ELSE 'Y' ENDIF,  
          IF table_type <> VIEW'  
            THEN view_def ENDIF,  
          remarks  
FROM SYS.SYSTABLE
```

Lists all the tables and views from SYSTABLE in a readable format.



## SYS.SYSCOLUMNS

```
CREATE VIEW SYS.SYSCOLUMNS ( creator, cname, tname,
                             coltype, nulls, length, syslength,
                             in_primary_key, "colno", default_value, remarks )
AS SELECT ( SELECT user_name FROM SYS.SYSUSERPERM
           WHERE user_id = SYSTABLE.creator ),
         column_name, table_name,
         ( SELECT domain_name FROM SYS.SYSDOMAIN
           WHERE domain_id = SYSCOLUMN.domain_id ),
         nulls, width, scale, pkey, column_id,
         "default", SYSCOLUMN.remarks
FROM SYS.SYSCOLUMN == SYS.SYSTABLE
```

Presents a readable version of the table SYSCOLUMN. (Note the S at the end of the view name that distinguishes it from the SYSCOLUMN table.)

## **SYS.SYSVIEWS**

```
CREATE VIEW SYS.SYSVIEWS ( vcreator, viewname, viewtext )
AS SELECT user_name, table_name, view_def
FROM SYS.SYSTABLE KEY JOIN SYS.SYSUSERPERM
WHERE table_type = 'VIEW'
```

Lists views along with their definitions.

## SYS.SYSINDEXES

```

CREATE VIEW SYS.SYSINDEXES ( icreator, iname, fname, creator,
                             tname, indextype, colnames, interval, level )
AS SELECT ( SELECT user_name FROM SYS.SYSUSERPERM
            WHERE user_id = SYSINDEX.creator ),
          index_name,
          ( SELECT file_name FROM SYS.SYSFILE
            WHERE file_id = SYSINDEX.file_id ),
          ( SELECT user_name FROM SYS.SYSUSERPERM
            WHERE user_id = SYSINDEX.creator ),
          table_name,
          IF "unique" = 'Y' THEN 'Unique'
            ELSE 'Non-unique' ENDIF,
          ( SELECT list( string( column_name,
                                IF "order" = 'A' THEN ' ASC' i
                                  ELSE ' DESC' ENDIF ) )
            FROM SYS.SYSIXCOL == SYS.SYSCOLUMN
            WHERE index_id = SYSINDEX.index_id ), 0, 0
FROM SYS.SYSTABLE KEY JOIN SYS.SYSINDEX

```

Presents index information from SYSINDEX and SYSIXCOL in a more readable format.

## SYS.SYSFOREIGNKEYS

```
CREATE VIEW SYS.SYSFOREIGNKEYS ( foreign_creator,
                                foreign_tname, primary_creator,
                                primary_tname, role, columns )
AS SELECT ( SELECT user_name FROM
            SYS.SYSUSERPERM == SYS.SYSTABLE
            WHERE table_id = foreign_table_id ),
          ( SELECT table_name FROM SYS.SYSTABLE
            WHERE table_id = foreign_table_id ),
          ( SELECT user_name
            FROM SYS.SYSUSERPERM == SYS.SYSTABLE
            WHERE table_id = primary_table_id ),
          ( SELECT table_name FROM SYS.SYSTABLE
            WHERE table_id = primary_table_id ), role,
          ( SELECT list( string( FK.column_name,
                                ' IS ', PK.column_name ) )
            FROM SYS.SYSFKCOL KEY JOIN
            SYS.SYSCOLUMN FK, SYS.SYSCOLUMN PK
            WHERE foreign_table_id =
            SYSFOREIGNKEY.foreign_table_id
            AND foreign_key_id = SYSFOREIGNKEY.foreign_key_id
            AND PK.table_id = SYSFOREIGNKEY.primary_table_id
            AND PK.column_id = SYSFKCOL.primary_column_id )
          FROM SYS.SYSFOREIGNKEY
```

Presents foreign key information from SYSFOREIGNKEY and SYSFKCOL in a more readable format.

## SYS.SYSUSERAUTH

```
CREATE VIEW SYS.SYSUSERAUTH ( name, password, resourceauth,  
                             dbaauth, scheduleauth, user_group )  
AS SELECT user_name, password, resourceauth,  
         dbaauth, scheduleauth, user_group  
FROM SYS.SYSUSERPERM
```

Displays all the information in the table SYSUSERPERM except for user numbers. Since this view shows passwords, this system view does not have PUBLIC select permission. (All other system views have PUBLIC select permission.)

## **SYS.SYSUSERPERMS**

```
CREATE VIEW SYS.SYSUSERPERMS
AS SELECT user_id, user_name, resourceauth, dbaauth,
         scheduleauth, user_group, remarks
FROM SYS.SYSUSERPERM
```

Contains exactly the same information as the table **SYS.SYSUSERPERM** except the password is omitted. All users have read access to this view, but only the DBA has access to the underlying table (**SYS.SYSUSERPERM**).

## **SYS.SYSUSERLIST**

```
CREATE VIEW SYS.SYSUSERLIST ( name, resourceauth,  
                             dbaauth, scheduleauth, user_group )  
AS SELECT user_name, resourceauth,  
         dbaauth, scheduleauth, user_group  
FROM SYS.SYSUSERPERM
```

Presents all information in SYSUSERAUTH except for passwords.

## **SYS.SYSGROUPS**

```
CREATE VIEW SYS.SYSGROUPS ( group_name, member_name )
AS SELECT g.user_name, u.user_name
FROM SYS.SYSGROUP, SYS.SYSUSERPERM g, SYS.SYSUSERPERM u
WHERE group_id = g.user_id AND group_member = u.user_id
```

Presents group information from SYSGROUP in a more readable format.



## SYS.SYSTABAUTH

```

CREATE VIEW SYS.SYSTABAUTH ( grantor, grantee,
                             screator, stname, tcreator, tname,
                             selectauth, insertauth, deleteauth,
                             updateauth, updatecols, alterauth, referenceauth )
AS SELECT ( SELECT user_name FROM SYS.SYSUSERPERM
            WHERE user_id = SYSTABLEPERM.grantor ),
          ( SELECT user_name FROM SYS.SYSUSERPERM
            WHERE user_id = SYSTABLEPERM.grantee ),
          ( SELECT user_name
            FROM SYS.SYSUSERPERM == SYS.SYSTABLE
            WHERE table_id = SYSTABLEPERM.stable_id ),
          ( SELECT table_name FROM SYS.SYSTABLE
            WHERE table_id = SYSTABLEPERM.stable_id ),
          ( SELECT user_name FROM
            SYS.SYSUSERPERM == SYS.SYSTABLE
            WHERE table_id = SYSTABLEPERM.ttable_id ),
          ( SELECT table_name FROM SYS.SYSTABLE
            WHERE table_id = SYSTABLEPERM.ttable_id ),
          selectauth, insertauth, deleteauth,
          updateauth, updatecols,
          alterauth, referenceauth
FROM SYS.SYSTABLEPERM

```

Presents table permission information in SYSTABLEPERM in a more readable format.

## **SYS.SYSCOLAUTH**

```
CREATE VIEW SYS.SYSCOLAUTH ( grantor, grantee, creator,  
                             tname, colname )  
AS SELECT ( SELECT user_name FROM SYS.SYSUSERPERM  
            WHERE user_id = SYSCOLPERM.grantor ),  
          ( SELECT user_name FROM SYS.SYSUSERPERM  
            WHERE user_id = SYSCOLPERM.grantee ),  
          ( SELECT user_name  
            FROM SYS.SYSUSERPERM == SYS.SYSTABLE  
            WHERE table_id = SYSCOLPERM.table_id ),  
          ( SELECT table_name FROM SYS.SYSTABLE  
            WHERE table_id = SYSCOLPERM.table_id ),  
          ( SELECT column_name FROM SYS.SYSCOLUMN  
            WHERE table_id = SYSCOLPERM.table_id  
            AND column_id = SYSCOLPERM.column_id )  
FROM SYS.SYSCOLPERM
```

Presents column update permission information in SYSCOLPERM in a more readable format.

## SYS.SYSOPTIONS

```
CREATE VIEW SYS.SYSOPTIONS ( user_name, "option", "setting" )
AS SELECT ( SELECT user_name FROM SYS.SYSUSERPERM
            WHERE user_id = SYSOPTION.user_id ),
            "option", "setting"
FROM SYS.SYSOPTION
```

Displays option settings contained in the table SYSOPTION in a more readable format.

## **SYS.SYSUSEROPTIONS**

```
CREATE VIEW SYS.SYSUSEROPTIONS ( "user_name",  
                                "option", "setting" )  
as SELECT u.name, "option",  
         isnull( ( SELECT "setting" FROM sys.sysoptions s  
                   WHERE s.user_name = u.name  
                   AND s."option" = o."option" ),  
                "setting" )  
FROM SYS.SYSOPTIONS o, SYS.SYSUSERAUTH u  
WHERE o.user_name = 'PUBLIC'
```

Display effective permanent option settings for each user. If a user has no setting for an option, this view will display the public setting for the option.

## SYS.SYSTRIGGERS

```
CREATE VIEW SYS.SYSTRIGGERS ( owner, trigname, tname,
                             event, trigttime, trigdefn )
AS SELECT ( SELECT user_name FROM SYS.SYSUSERPERM
            WHERE user_id = SYSTABLE.creator ),
         trigger_name, table_name,
         IF event = 'I' THEN 'INSERT'
         ELSE IF event = 'U' THEN 'UPDATE'
         ELSE IF event = 'C' THEN 'UPDATE'
         ELSE 'DELETE' ENDIF ENDIF ENDIF,
         IF trigger_time = 'B' THEN 'BEFORE'
         ELSE 'AFTER' ENDIF,
         trigger_defn
FROM SYS.SYSTRIGGER == SYS.SYSTABLE
WHERE foreign_table_id IS NULL
```

Lists all the triggers from SYSTRIGGER in a readable format.

## **SYS.SYSPROCPARMS**

```
CREATE VIEW SYS.SYSPROCPARMS ( creator, parmname, procname,  
                               parmtype, parmmode, parmdomain, length, remarks )  
AS SELECT ( SELECT user_name FROM SYS.SYSUSERPERM  
            WHERE user_id = SYSPROCEDURE.creator ),  
          parm_name, proc_name, parm_type,  
          IF parm_mode_in = 'Y' AND  
            parm_mode_out = 'N' THEN 'IN'  
          ELSE IF parm_mode_in = 'N'  
            AND parm_mode_out = 'Y' THEN 'OUT'  
          ELSE 'INOUT' ENDIF ENDIF,  
          ( SELECT domain_name FROM SYS.SYSDOMAIN  
            WHERE domain_id = SYSPROCPARM.domain_id ),  
          width, SYSPROCPARM.remarks  
FROM SYS.SYSPROCPARM == SYS.SYSPROCEDURE
```

Lists all the procedure parameters from SYSPROCPARM in a readable format.

## SYS.SYSPROCAUTH

```
CREATE VIEW SYS.SYSPROCAUTH ( grantee, creator, procname )
AS SELECT ( SELECT user_name FROM SYS.SYSUSERPERM
           WHERE SYSPROCPERM.grantee =
             SYSUSERPERM.user_id ),
         ( SELECT user_name FROM SYS.SYSUSERPERM
           WHERE SYSPROCEDURE.creator =
             SYSUSERPERM.user_id ),
         proc_name
FROM SYS.SYSPROCEDURE == SYS.SYSPROCPERM;
```

Lists all the procedure permissions from SYSPROCPERM in a readable format.





# Index

## A

- abort a command 46
- ABS function 76
- ACOS function 76
- actions 63
- active connection 52
- aggregate function 73
  - AVG 74
  - COUNT 73, 144
  - LIST 74
  - MAX 74
  - MIN 74
  - SUM 74
- alias (column) 100
- ALL 93, 100
- ALTER DBSPACE 186
- ALTER TABLE 187
- AND 93
- ANY 92
- applying updates 112
- ARGN function 75
- arithmetic 85
- ASCII 318
- ASCII file format 293, 296
- ASCII function 77
- ASIN function 76
- assertion failed 409
- ATAN function 76
- ATOMIC 154, 199
- audit trail 119
- auto\_commit, ISQL option 293
- automatic join 101, 353
- average 74
- AVG function 74

## B

- B-tree 420, 427
- backup 115, 120, 308

- base table 210
- beep 293
- BEGIN 199
- bell, ISQL option 293
- BETWEEN 91
- BINARY data type 67
- blocking 107, 285
- bulk operations 325

## C

- C 25, 307
- cache 127, 324-325
  - size 325
- CALL 192, 147, 151
- CASCADE 63, 214
- CASE 193
- case sensitive 42, 182, 317
- CAST 87
- CEILING function 76
- change column definitions 188
- change table definitions 188
- CHAR data type 66
- CHAR function 77
- CHARACTER data type 67
- character set 69
- CHARACTER VARYING data type 67
- CHECK 215
- CHECKPOINT 194, 117
- checkpoint log 117
- client 28
- CLOSE 167, 195
- COALESCE function 75
- code page 69, 173, 318
- collation sequence 173, 311, 318, 437
- column 422, 447
  - alias 100
  - changing heading name 100
  - constraints 212
  - list 46

# Index

- name 86, 183
- permission 433
- comma delimited files 293, 296
- command echo 293
- command files 272
  - parameters 272
- command line summary 307
- command recall 44
- Command window 38
  - Command 38
- commands 181
- COMMENT 196
- COMMIT 197
- commit on exit, ISQL option 293
- comparison 90
- components 25, 307
- Compound statement 199
- concatenation 87
- concurrency 107
- conditions 90
- configuration 283
- CONFIGURE 201
- conflict resolution 112
- connecting 40
- connection 52
- connection parameters 53
- consistency 104
- constant 85
- constraints 212, 189
- control statement 147
- COS function 76
- cost based optimizer 133
- COT function 76
- COUNT function 73, 144
- CREATE DBSPACE 204
- CREATE INDEX 205, 130
- CREATE PROCEDURE 207, 150
- CREATE TABLE 209
  - examples 413

- CREATE TRIGGER 217, 158
- CREATE VARIABLE 219
- CREATE VIEW 221, 144-146
  - examples 445
- creating a new database 317
- creator 183
- CURRENT DATE 85
- current query 188, 267
- CURRENT TIME 85
- CURRENT TIMESTAMP 85
- cursor 167, 238, 265
- cursor stability 105

## D

- data dictionary
  - see system tables
- data source 32
- data types 66, 183, 421
- data window 38
- database
  - cache 324
  - creating a new 317
  - engine 25, 307, 324
  - erasing 313
  - files 204, 418
  - information 315
  - starting 324
  - stopping 330
  - structure 413
  - unload 333
  - write file 339
- database administrator 138
- database information 315
- DATE data type 68, 82
- date format, database option 286
- DATE function 82
- date order, database option 287
- DATE\_ORDER 88
- DATEFORMAT function 82

# Index

- dates 70, 79-83
    - formatting 82
  - DATETIME function 83
  - DAY function 81
  - DAYS function 81
  - DB32 324
  - DBA
    - see database administrator
  - DBA authority 415
  - dBASE file format 293, 296
  - DBBACKUP 120, 308
  - DBCOLLAT 311
  - DBERASE 313
  - DBEXPAND 314
  - DBINFO 315
  - DBINIT 317, 435
  - DBLOG 322
  - DBSHRINK 323
  - DBSPACE 204, 210, 234, 418
  - DBSTART 25, 307
  - DBSTOP 330
  - DBTOOL 223
  - DBTRAN 119, 331
  - DBUNLOAD 333
  - DBUPGRAD 336
  - DBVALID 120, 338
  - DBWRITE 339
  - deadlock 107
  - DECIMAL data type 67
  - decimal precision, database option 288
  - DECLARE 167, 199
  - DELETE 231
  - descriptor 238, 265, 304
  - device failure 115
  - DIF file format 296
  - digits, maximum number 288
  - dirty read 104
  - disk failure 115
  - DISTINCT 100
  - domain
    - see data types
  - DOUBLE data type 68
  - DOW function 81
  - DROP DBSPACE 234
  - DROP INDEX 234
  - DROP OPTIMIZER STATISTICS 235
  - DROP PROCEDURE 234
  - DROP TABLE 234
  - DROP TRIGGER 234, 160
  - DROP VARIABLE 236
  - DROP VIEW 234, 144
  - DYNAMIC SCROLL cursor 227
- ## E
- EBCDIC 318
  - echo, ISQL option 293
  - editing commands 44
  - efficiency
    - see improving performance
  - ELSE 87
  - Embedded SQL
    - cursors 167
  - encryption 317
  - END 199
  - ENDIF 87
  - engine 25, 307
  - entity integrity 352
  - environment variables 346
  - erasing a database 313
  - error
    - codes 357
    - handling in ISQL 295
    - messages 357
  - establish a connection 52
  - estimates 135
  - EXCEPTION 199
  - exclusive lock 106
  - executable files 307

# Index

execute 151  
executing commands 42  
EXISTS 92  
EXP function 76  
exporting data 99, 267, 296  
expressions 84-85, 264  
    NULL 264

## F

failure 115  
FALSE condition 91  
FETCH 238, 167  
files 204, 418  
finding procedures 140  
finding tables 140  
FIXED file format 293, 296  
FLOAT data type 68  
FLOOR function 76  
FOR 241  
for update 239  
foreign key 62, 129, 213, 352, 427, 429, 450  
foreign table 427  
format 293, 296  
forward log 118  
FoxPro file format 293, 296  
fragmentation 127  
FROM 101, 243  
functions 71  
    ABS 76  
    ACOS 76  
    ARGN 75  
    ASCII 77  
    ASIN 76  
    ATAN 76  
    AVG 74  
    CEILING 76  
    CHAR 77  
    COALESCE 75  
    COS 76  
    COT 76  
    COUNT 73  
    DATE 82  
    DATEFORMAT 82  
    DATETIME 83  
    DAY 81  
    DAYS 81  
    DOW 81  
    EXP 76  
    FLOOR 76  
    HOUR 81  
    HOURS 81  
    IFNULL 75  
    ISNULL 75  
    LCASE 77  
    LEFT 77  
    LENGTH 77  
    LIST 74  
    LOCATE 77  
    LOG 76  
    LOG10 76  
    LTRIM 78  
    MAX 74  
    MIN 74  
    MINUTE 81  
    MINUTES 81  
    MOD 76  
    MONTH 80  
    MONTHS 80  
    NOW 83  
    NUMBER 75  
    PI 76  
    PLAN 78  
    REMAINDER 77  
    RIGHT 78  
    RTRIM 78  
    SECOND 82  
    SECONDS 82  
    SIGN 77

# Index

SIMILAR 78  
SIN 77  
SOUNDEX 78  
SQRT 77  
STRING 78  
SUBSTR 79  
SUM 74  
TAN 77  
TODAY 83, 438  
TRACEBACK 79  
TRIM 79  
UCASE 79  
WEEKS 80  
YEAR 79  
YEARS 79  
YMD 83

## G

getting commands 42  
GLOBAL 210  
GRANT 249, 138-139  
group (user group) 140  
GROUP BY 101, 144  
group tables 140

## H

HAVING 101  
heading name 100  
heading, ISQL option 293  
help 253, 39  
heuristic 133  
HOLD 265  
HOUR function 81  
HOURS function 81

## I

I/O estimates 135

I/O operations 328  
identifier 183  
IF 87  
IF statement 254  
IFNULL function 75  
importing data 255  
improving performance 129-135  
    estimates 135  
    temporary tables 134  
IN 92  
index 130, 205, 424, 426, 449  
    creating 130  
inherit 140  
initializing a database 317  
inner join 246  
INPUT 255  
input format, ISQL option 293  
INSERT 259  
INT data type 67  
INTEGER data type 67  
integrity 212  
Interactive SQL  
    see ISQL  
interrupt 46  
INTO 100  
invoke 151  
IS FALSE 94  
IS NULL 91  
IS TRUE 94  
IS UNKNOWN 94  
ISNULL function 75  
isolation level 105, 109, 265  
ISQL 37, 307, 341  
    error handling 295  
    executing commands 42  
    getting commands 42  
    loading commands 42  
    Microsoft Windows 37  
    Microsoft Windows NT 37

# Index

options 292  
saving commands 42  
ISQL\_LOG, ISQL option 295

## J

join 244, 101, 131, 353

## K

key 129  
key join 245, 131  
keyboard 40  
keyword 411

## L

label 184  
LAN 28  
large databases 112  
LCASE function 77  
LEAVE 261  
LEFT function 77  
left, moving screen 43  
LENGTH function 77  
ligatures 174  
LIKE 92  
LIST function 74  
list of columns 46  
list of tables 46  
literal string 85, 184  
loading commands 42  
LOCATE function 77  
locking 103  
log file 117-118, 322  
LOG function 76  
LOG10 function 76  
logging on 40  
see also connecting

logical unit of work 55  
see also transaction  
LONG BINARY data type 67  
LONG VARCHAR data type 67  
LOOP 262  
LOTUS file format 293, 296  
LTRIM function 78

## M

mathematics 85  
MAX function 74  
maximum 74  
media failure 115  
membership 140  
Microsoft Windows 37  
Microsoft Windows NT 37  
MIN function 74  
minimum 74  
MINUTE function 81  
MINUTES function 81  
MOD function 76  
modify column definitions 188  
modify table definitions 188  
MONTH function 80  
MONTHS function 80  
moving screen 43  
multi-user 28  
multiple row queries 167

## N

named connection 52  
natural join 245  
network 28  
new database 317  
NO SCROLL cursor 227  
non-repeatable read 104  
normal function  
see functions

# Index

NOT 94  
not found warning 167  
NOW function 83  
NULL value 263, 60, 75, 295  
NULLS, ISQL option 295  
NUMBER function 75, 302  
number of rows 419  
numbering columns 75  
numbers 85  
NUMERIC data type 68  
numeric precision, database option 288

## O

occasionally connected 112  
ODBC 31  
    data source 32  
offline backup 120  
on error, ISQL option 295  
online backup 120, 308  
OPEN 265, 167  
Open Database Connectivity  
    see ODBC  
operators  
    expression 86  
    join 243  
optimization  
    see improving performance  
optimizer 133  
options 283, 201, 223, 434, 457-458  
    AUTO\_COMMIT 293  
    AUTO\_REFETCH 293  
    BELL 293  
    BLOCKING 285  
    CHECKPOINT\_TIME 285  
    COMMAND\_DELIMITER 286  
    COMMIT\_ON\_EXIT 293  
    CONVERSION\_ERROR 286  
    DATE\_FORMAT 286  
    DATE\_ORDER 287

ECHO 293  
HEADINGS 293  
INPUT\_FORMAT 293  
ISOLATION\_LEVEL 288  
ISQL\_LOG 295  
NULLS 295  
ON\_ERROR 295  
OUTPUT\_FORMAT 296  
OUTPUT\_LENGTH 297  
PRECISION 288  
RECOVERY\_TIME 288  
ROW\_COUNTS 289  
SCALE 289  
STATISTICS 297  
THREAD\_COUNT 289  
TIME\_FORMAT 290  
TIMESTAMP\_FORMAT 290  
TRUNCATION\_LENGTH 297  
WAIT\_FOR\_COMMIT 291

OR 93  
ORDER BY 101, 132  
outer join 246, 86, 247  
OUTPUT 267  
output format, ISQL option 296  
output length, ISQL option 297  
overflow 288

## P

page size 317  
PARAMETERS 270  
parameters to command files 272  
password 40, 138, 415  
    changing 250  
pattern matching 92  
performance 118, 127  
permission 249, 138, 146, 277, 430, 433,  
    455-456  
    see also GRANT  
CONNECT authority 250

# Index

- DBA authority 250
- EXECUTE 251
- GROUP authority 250
- MEMBERSHIP 251
- RESOURCE authority 250
- phantom lock 106
- phantom row 104
- PI function 76
- PLAN function 78
- portable computers 112
- power failure 115
- PREPARE TO COMMIT 271
- prepared statement 420
- previous commands 44
- primary key 75, 110, 129, 131, 213, 352, 420, 422, 427
- primary table 427
- procedure 147
- procedures
  - see CREATE PROCEDURE
- projection 100
- PUBLIC userid 139, 416

## R

- READ 272
- read from a cursor
  - see fetch
- read lock 106
- REAL data type 68
- recalling commands 44
- recovery 115
- referential integrity 243, 352
  - see also foreign key
- relationship 427
- release a connection 52
- RELEASE SAVEPOINT 274
- REMAINDER function 77
- rename a column 190
- rename a table 190

- repeatable read 104
- resident program 324
- RESIGNAL 275
- resource authority 415
- RESTRICT 63, 214
- RESULT 207
- RESUME 276
- return codes 349
- REVOKE 277
- RIGHT function 78
- right, moving screen 43
- role name 184, 213
- roll forward 118
- ROLLBACK 279, 118
- rollback log 118
- ROLLBACK TO SAVEPOINT 280
- rounding 289
- row counts 289
- row not found 167
- RTRIM function 78
- RTSQL 341

## S

- SAVEPOINT 281, 58, 184, 274, 280
- saving commands 42, 295
- SCROLL cursor 227
- search conditions 90
- SECOND function 82
- SECONDS function 82
- security 40
- SELECT 99, 42, 129-130, 131-135
  - examples 445
- select list 100
- server 28
- SET DEFAULT 63, 214
- SET NULL 63, 214
- SET OPTION 283
- SET variable 298
- share lock 106



# Index

- SIGN function 77
- SIGNAL 299
- signing on
  - see connecting
- SIMILAR function 78
- SIN function 77
- single row queries 166
- single row table 438
- SMALLINT data type 67
- software
  - DB32 324
  - DBBACKUP 308
  - DBCOLLAT 311
  - DBERASE 313
  - DBEXPAND 314
  - DBINFO 315
  - DBINIT 317
  - DBLOG 322
  - DBSHRINK 323
  - DBSTART 324
  - DBSTOP 330
  - DBTRAN 331
  - DBUNLOAD 333
  - DBUPGRAD 336
  - DBVALID 338
  - DBWRITE 339
  - ISQL 341
  - return codes 349
  - RTSQL 341
- software components 307
- SOME 93
- sort
  - see ORDER BY
- sorting 132, 437
- SOUNDEX function 78
- special tables 413
- speed 127
- SQL 181
- SQL file format 296
- SQL keywords
  - ALL 93, 100
  - ALTER DBSPACE 186
  - ALTER TABLE 187
  - AND 93
  - ANY 92
  - BETWEEN 91
  - BYE 237
  - CALL 192
  - CASE 193
  - CHECKPOINT 194
  - CLOSE 195
  - COMMENT 196
  - COMMIT 197
  - Compound statement 199
  - Conditions 90
  - CONFIGURE 201
  - CONNECT 202, 138
  - CREATE DBSPACE 204
  - CREATE INDEX 205, 130
  - CREATE PROCEDURE 207
  - CREATE TABLE 209
  - CREATE TRIGGER 217
  - CREATE VARIABLE 219
  - CREATE VIEW 221, 144-146
  - Data types 66
  - DBTOOL 223
  - DECLARE CURSOR 227
  - DECLARE TEMPORARY TABLE 230
  - DELETE 231, 232
  - DISCONNECT 233
  - DISTINCT 100
  - DROP DBSPACE 234
  - DROP INDEX 234
  - DROP OPTIMIZER STATISTICS 235
  - DROP PROCEDURE 234
  - DROP TABLE 234
  - DROP TRIGGER 234
  - DROP VARIABLE 236

# Index

- DROP VIEW 234, 144
- EXISTS 92
- EXIT 237
- expressions 84
- FETCH 238
- FOR 241
- FROM 101, 243
- Functions 71
- GRANT 249, 138-139
- GROUP BY 101, 144
- HAVING 101
- HELP 253
- IF statement 254
- IN 92, 92
- INPUT 255
- INSERT 259
- INTO 100
- IS NULL 91
- LEAVE 261
- LIKE 92
- LOOP 262
- NOT 94
- NULL value 263
- OPEN 265
- OR 93
- ORDER BY 101
- OUTPUT 267
- PARAMETERS 270
- PREPARE TO COMMIT 271
- QUIT 237
- READ 272
- RELEASE SAVEPOINT 274
- RESIGNAL 275
- RESUME 276
- REVOKE 277
- ROLLBACK 279
- ROLLBACK TO SAVEPOINT 280
- SAVEPOINT 281
- SELECT 99, 42, 129-130, 131-135
- SET CONNECTION 282
- SET OPTION 283
- SET variable 298
- SIGNAL 299
- SOME 93
- UNION 300
- UPDATE 302, 304
- USER 85, 438
- VALIDATE TABLE 305
- WHERE 101
- WHILE 262
- SQL variable 219, 236, 298
- SQLCODE 357, 85, 161
- SQLCONNECT environment variable 346
- SQLDriverConnect 53
- SQLPP
- SQLSTART environment variable 347
- SQLSTATE 363, 85, 161
- SQRT function 77
- starting the software
  - in Windows 38
  - in Windows NT 38
- statement label 184
- Statistics window 38
  - statistics 38
- statistics, ISQL option 297
- stopping the database 330
- stored procedure 147, 439
- stored procedures
  - see CREATE PROCEDURE
- string concatenation 87
- string constant 85, 184
- STRING function 78
- structure of a database 413
- subquery 86, 92
- SUBSTR function 79
- subtransaction 58
- SUM function 74
- syntax rules 181

# Index

SYS userid 416  
system catalogue 413, 446  
system failure 115  
system tables 413  
system views 445  
    see also system tables

## T

table constraints 212  
table list 243, 46, 184  
table name 184  
table number 419  
tables 419  
    DUMMY 438  
    Employee 130-131  
    looking at 42  
    Mark 138  
    Register 131, 138, 144  
    Section 131  
    Student 129, 138  
    SYSCATALOG 146, 446  
    SYSCOLLATE 437  
    SYSCOLPERM 433  
    SYSCOLUMN 422  
    SYSCOLUMNS 146, 447  
    SYSDOMAIN 421  
    SYSFILE 418  
    SYSFKCOL 429  
    SYSFORIGNKEY 427  
    SYSFORIGNKEYS 450  
    SYSGROUP 417  
    SYSGROUPS 454  
    SYSINDEX 424  
    SYSINDEXES 449  
    SYSINFO 435  
    SYSIXCOL 426  
    SYSOPTION 434  
    SYSOPTIONS 457  
    SYSPROCAUTH 461

    SYSPROCEDURE 439  
    SYSPROCPARM 442  
    SYSPROCPARMS 460  
    SYSPROCPERM 444  
    SYSTABAUTH 455-456  
    SYSTABLE 419  
    SYSTABLEPERM 246, 430  
    SYSTRIGGER 440  
    SYSTRIGGERS 459  
    SYSUSERAUTH 451  
    SYSUSERLIST 453  
    SYSUSEROPTIONS 458  
    SYSUSERPERM 246, 415  
    SYSUSERPERMS 452  
    SYSVIEWS 448  
TAN function 77  
TEMPORARY 210  
temporary tables 134  
TEXT file format 296  
THEN 87  
three valued logic 94, 263  
time 68, 70  
time format, database option 290  
timestamp 70  
timestamp format, database option 290  
TMP environment variable 347  
TODAY function 83, 438  
total  
    see SUM function  
TRACEBACK function 79  
transaction 55, 103, 197, 279  
transaction log 118, 127, 322, 331  
translate 331, 338  
trigger 147, 158, 440  
triggers  
    see CREATE TRIGGER  
TRIM function 79  
TRUE condition 91  
truncation length, ISQL option 297

# Index

two-phase commit 59, 271  
type conversions 87  
types 66

## U

UCASE function 79  
UNION 300  
UNIQUE 205, 212  
UNKNOWN condition 91, 87  
UPDATE 302  
update column permission 433  
USER 85, 438  
user group 140  
user number 415  
userid 40, 138, 185, 415, 420, 451  
    changing passwords 250  
    creation 250

## V

VALIDATE TABLE 305  
validation 120  
validity checking 129  
VARCHAR data type 67  
variable 185, 219, 236, 298  
variable names 86  
view 221, 143-146, 448  
    creating 144  
    deleting 144  
    examples 445  
    permission 146

## W

WATFILE file format 293, 296  
WEEKS function 80  
WHERE 101  
WHILE 262  
windows 37

    Connect to Database 138  
    Statistics 131, 297  
WITH HOLD 265  
write file 339  
write lock 106  
WSQL environment variable 348

## Y

YEAR function 79  
YEARS function 79  
YMD function 83